Automated Software Engineering (2024) 31:28
https://doi.org/10.1007/s10515-024-00423-2

Comparing apples and oranges? Investigating the consistency of CPU and memory profiler results across multiple java versions

Myles Watkinson[1] · **Alexander E. I. Brownlee**[2]

Received: 31 July 2023 / Accepted: 1 February 2024

Abstract
Profiling is an important tool in the software developer's box, used to identify *hot* methods where most computational resources are used, to focus efforts at improving efficiency. Profilers are also important in the context of Genetic improvement (GI) of software. GI applies search-based optimisation to existing software with many examples of success in a variety of contexts. GI generates variants of the original program, testing each for functionality and properties such as run time or memory footprint, and profiling can be used to target the code variations to increase the search efficiency. We report on an experimental study comparing two profilers included with different versions of the Java Development Kit (JDK), HPROF (JDK 8) and Java Flight Recorder (JFR) (JDK 8, 9, and 17), within the GI toolbox Gin on six open-source applications, for both run time and memory use. We find that a core set of methods are labelled *hot* in most runs, with a long tail appearing rarely. We suggest five repeats enough to overcome this noise. Perhaps unsurprisingly, changing the profiler and JDK dramatically change the *hot* methods identified, so profiling must be rerun for new JDKs. We also show that using profiling for test case subset selection is unwise, often missing relevant members of the test suite. Similar general patterns are seen for memory profiling as for run time but the identified *hot* methods are often quite different.

Keywords Profiling · Runtime · Memory use · Genetic improvement · Java · Empirical study

✉ Alexander E. I. Brownlee
alexander.brownlee@stir.ac.uk

Myles Watkinson
myleswatkinson1@gmail.com

[1] School of Computer Science, University of Adelaide, Adelaide, Australia

[2] Computing Science and Mathematics, University of Stirling, Stirling, Scotland, UK

Published online: 22 March 2024

1 Introduction

Profiling tools are a useful aid to software developers in general, helping to identify areas of existing code for improvement. As systems become more complex and greater reliance is placed on automated tools to support developers, better understanding of profilers' behaviour is of great importance. Profilers are also relevant to Genetic Improvement (GI) of software (Petke et al. 2017). GI aims to automatically improve code by applying computational search methods, with impressive results being demonstrated for both functional improvements like bug fixes (e.g., GenProg (Le Goues et al. 2012)), and non-functional improvements, like run time (e.g., (Langdon and Harman 2015)). A critical, and yet underexplored, part of the GI process is the identification of *hot* methods or procedures, regions of code found to be bottlenecks that are targeted for mutation. *Hot* methods are usually identified by a profiling tool. Variations in a profiler's output will greatly impact the insights a developer can make or the performance of the GI search, motivating us to seek better understanding of the consistency and behaviour of profilers. The Gin open source toolbox for GI research in Java, originally proposed by White (2017) and further developed by Brownlee et al. (2019), was recently upgraded by us (Watkinson and Brownlee 2023) to support the Java Development Kit (JDK) 17 and part of that process switched from the retired HPROF profiler to the more recent Java Flight Recorder (JFR).

We (Watkinson and Brownlee 2023) outlined the initial process of changing the profiler in Gin and explored the difference in profiling outputs between HPROF under JDK 8 and JFR under JDK 9. We now recap and expand that work with a broader series of experiments, covering additional profiler/JDK combinations (JFR and JDK 8/17), five additional open-source target projects, and further analysis including variation within the results of each profiler and extension to memory profiling.

To measure the effectiveness of a profiler there needs to be some baseline or ground truth. Mytkowicz et al. (2010) faced a similar challenge when comparing the accuracy of Java profilers. They reached the conclusion that accuracy should be judged in terms of actionability. An actionable profile relates to the practical nature of the use of a programs profile. The profile is used to identify *hot* methods that are then altered to improve the program. If a profiler identifies ten slow methods and, when these are changed, the program speeds up considerably, the profiler has created an actionable profile. The profiler could be deemed inaccurate if its results were not actionable, that is if it returned 10 *hot* methods and when changed these did not actually speed up the program.

Actionable can be generalised to: "able to be used in the context needed". In our case, if the profiler finds *hot* methods that can be altered to improve the program, it is accurate. While the true 'hotness' of every method is not known, it can be assumed that if a profiler returns the same 10 methods every profiling run, and its method of determining hotness is correct, that it is accurate. Conversely, if a profiler returns an inconsistent list of *hot* methods each run it can be deemed inaccurate. Or, alternatively, the relative hotness of all methods is similar enough that there is little

value in identifying any subset of methods as *hot*, though as we will see, this is not the case for the projects in the present study (as demonstrated by the histograms in Figure 4 in Section 4.1, where around 50 methods are consistently identified as *hot*, and over 100 are identified by profiler in only 5–10% of repeat runs). Throughout the rest of this paper, we will describe methods as *hot* when they have been identified as such by a profiler; though in practice *hot* is really a property of the software and its typical usage.

In light of the above, our study seeks to answer the following research questions.

RQ1 How consistently does a single CPU profiler identify *hot* methods between repeat runs on a given JDK? How much confidence can we have in the profiling results? How many repeat runs are needed to capture most *hot* methods?

RQ2 How much do the *hot* methods change as we move from one CPU profiler and JDK to another? Will it be necessary to repeat the profiling stage for a new context?

RQ3 Is it possible to identify a subset of the test suite calling a target method (i.e., test case subset selection (Yoo and Harman 2012)) during the profiling process? As Gin samples the call stack during profiling, it also walks up the stack to identify unit tests calling each *hot* method. The aim is to reduce evaluation time for patches by only needing to run the relevant tests rather than the whole suite. How much do those tests identified vary?

RQ4 Does memory profiling follow a similar pattern to CPU profiling? Over the years GI has grown to target other non-functional properties of code such as memory consumption. We are unaware of any work exploring whether the *hot* methods are different for such properties, so what issues might appear if we apply a similar profiling approach to memory use?

We have focused on the context of GI, but investigation of profilers should also be of interest to Java developers more generally. Profiling already forms an important part of a developer's tool kit, allowing for performance bottlenecks to be identified for optimisation once the software's implementation has met the required specification. RQ1 explores how reliable the results of profiling are. RQ2 extends this to look at the impact of changing profiler and Java version. Migrating software to a different Java version is becoming more frequent as Oracle's release cycle switched to every six months so RQ2 is of increasing importance: do we need to reprofile for every upgrade? Our results suggest that we do. RQ3 investigates the automatic identification of tests calling *hot* methods, which is of relevance to automatic program repair researchers, and potentially relevant to anyone interested in test-driven development of software. RQ4 is also of wider relevance to Java developers. As cloud and mobile apps continue to grow in importance, developers strive to make software less resource-heavy, motivating investigation of non-CPU properties such as memory.

Our experimental results lead to several conclusions. We use Weighted Rank Biased Overlap (WRBO) (Webber et al. 2010) to compare the ranked lists of *hot* methods produced by profiler runs, giving a scale of 0 to 1 for similarity in the rankings. We show that each of the profilers is largely consistent in the *hot* methods that it identifies during multiple repeat runs, with a WRBO of over 0.65. In practice, five repeat runs are enough to identify *hot* methods with 95% confidence. However, changing profiler and JDK leads to dramatically different profiling results, with a typical WRBO of 0.2 when switching profiler on the same JDK, and between 0.2–0.7 for the same profiler on different JDKs. This second point is, perhaps, unsurprising, but to our knowledge has not been tested experimentally in this way before. We also show that using a profiler for test case subset selection as described in RQ3 should generally be avoided; even in the context of GI whereby the full test suite is used to confirm functionality of the final results, the selected tests vary so much that it is likely the search will find sub-optimal results that break functionality, requiring further rerunning of the search. In the example highlighted in Sect. 4.3, for only 9 of 57 identified *hot* methods did the same test cases appear in nearly all repeat runs. From this we might also conclude that testing is itself suboptimal, and observing the behaviour of programs "in the real world" would better serve to identify where optimisation is needed. We also show that memory profiling follows broadly similar patterns to those seen for CPU profiling, albeit identifying rather different sets of *hot* methods. The Spearman correlation between ranks assigned to *hot* methods for CPU and memory varies between −1.0 (i.e., the most-*hot* methods for CPU were the least-*hot* for memory), and +0.751 (i.e., *hot* methods for CPU and memory were largely the same).

Section 2 describes the approach to integrating the profiling tools within Gin. Section 3 describes our experimental study and summarises the results, with further discussion in Sect. 5. Sections 6 and 7 summarise the threats to validity of our results, and relevant related work. In Sect. 8 we give our conclusions and suggestions for future work.

2 Profilers and integration with Gin

Our experiments focus on profiling in the context of the Java GI toolkit Gin, which provides several utilities to ease our experimental pipeline including a consistent API for integration with large projects built with both the Maven and Gradle build tools, and an existing profiling framework. There are many profiling tools available for Java. We investigate the profilers HPROF (Oracle 2011) and Java Flight Recorder (JFR) (Oracle 2014). We now summarise the motivation behind focusing on these two. Key to an automated experimental pipeline is the ability to call the profiler programatically without manual intervention. We also wish to limit any additional overhead or dependencies and, in the interests of open science, avoid anything requiring a commercial licence. Ideally our profilers should also be able to capture both CPU and memory usage. We previously summarised (Watkinson and Brownlee 2023) these desirable qualitative properties of several current profiling tools with a view to upgrading Gin for newer versions of Java. The third-party

tool JProfiler (EJ-Technologies 2020) was ruled out due to its commercial licence. The profilers integrated within IDEs, such as NetBeans (Apache 2020) and Eclipse (Eclipse Foundation 2019), were ruled out due to the overhead associated with running the host IDE. VisualVM (Sedlacek and Hurka 2022), Java Mission Control (Oracle 2018), and JConsole (Oracle 2023) all require some element of user interaction with a visual interface, preventing their use in an automated experimental pipeline. This leaves HPROF and JFR, both of which are free and can be called programatically. HPROF was included with JDK versions up to 8, and was the original profiling tool integrated within Gin up to v2.0 (White 2017; Brownlee et al. 2019). This led to its use in identifying code for improvement in several GI studies (Petke and Brownlee 2019; Petke et al. 2023; Brownlee et al. 2020), so we include it for comparison within our study. JFR has been bundled with the JDK since version 7, being officially supported since version 9. We first described the integration of both profilers within Gin in (Watkinson and Brownlee 2023), and recap the details here for convenience.

Both HPROF and JFR profilers rely on sampling, whereby a trace of the stack is made at intervals to determine which methods appear most often. The basic idea is that the method on the top of the call stack is recorded at intervals to determine where CPU time is spent or changes to memory use occur. Sampling has the inherent potential to introduce noise to the results by identifying different methods as *hot* on different runs. Both HPROF and JFR also add a small amount of randomness to the profiling interval to avoid bias towards functions that run at the same rate as the sampling. Further, HPROF and JFR take slightly different approaches to sampling that may also impact results so we briefly explain this difference. The interested reader can refer to the official documentation for HPROF (Oracle 2011) and JFR (Oracle 2014) for further detail. JFR records information on events that occur in the Java Virtual Machine (JVM). To minimise overhead, rather than stopping all threads at once to take a stack trace of everything, JFR stops threads individually. Each trace produces an *event* that is recorded, and these events are output to a .jfr file after a recording ends (Oracle Corporation 2022). A consequence of JFR's approach is that only stack traces ending in Java code (rather than JDK library code) are recorded. HPROF differs by profiling all threads inside the JVM at intervals. This differs from the event based profiling of JFR as HPROF can profile sleeping, waiting or blocked threads and I/O calls that JFR can not see. In the context of GI, we are only interested in identifying the Java code that can be modified by the search. For API and library calls that means identifying the calling code in the project being profiled; which is not always possible through HPROF's approach. So, a filtering step is introduced with the net result that the results of both profilers identify *hot* methods within the target application's Java source only.

2.1 Integration with Gin

As both tools are built-in to Java, integrating HPROF and JFR with Gin is relatively simple. For the profiling tool, Gin runs a target application with a separate Java

virtual machine instance, adding flags to the Java command that cause it to invoke the profiler. In Gin versions up to v2.1, the command to invoke HPROF is:

```
-agentlib:hprof=cpu=samples,lineno=y,depth=1,interval=$hprofInterval,file=
```

where, by default, the interval in milliseconds is 10. The initial upgrade to JFR for JDK 8 and 9 proposed by Watkinson and Brownlee (2023) used the following:

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
-XX:StartFlightRecording=name=Gin,dumponexit=true,settings=profile,
    filename=
```

where the "profile" option for `settings` makes samples every 10ms (currently this rate is not user-configurable). Current versions of the JDK no longer require the first two options so Gin for JDK 17 uses this:

```
-XX:StartFlightRecording=name=Gin,dumponexit=true,settings=profile,filename=
```

HPROF and JFR each output respective files that are used to extract results. HPROF outputs results to a text file that can be easily read and translated. JFR outputs to a .jfr file with a binary format that requires specific parsing tools provided with the Java API to view: these are provided officially from JDK 9 and up (GitHub 2020); in JDK 7–8 the JFR parsing API is radically different, and undocumented[1]. Programmatically, this file is broken into `RecordedEvent` objects, a data structure from the JFR library, by first reading the .jfr file to a JFR `RecordingFile` object. Gin is concerned with stack traces, so each `RecordedEvent` with the type `ExecutionSample` is collected. Each `ExecutionSample` has a list of stack frames that are iterated over to identify methods and function calls that occurred in order. Though it does not fit the context of Gin, the free tool Java Mission Control[2] can easily visualise JFR files to debug.

Figure 2 shows an example HPROF output file from a profiling run of JCodec (See Sect. 3.1 for projects in our study). The top ranked method, a Java native I/O function is found 112 times. As discussed before, JFR can not profile I/O calls so this function can only be found by HPROF. However, this is actually a negative for HPROF if the calling function cannot be identified as both GI and human developers aim to target editable source code rather than internal Java API functions. Moving down the list, methods 2, 4, 6–8, and 10 are all Java native calls, Gin ignores these as well. For these functions, JFR would jump past them in the call stack to find a function part of JCodec. By doing this, in the example JFR could potentially return an extra 132 samples of JCodec methods.

[1] http://hirt.se/blog/?p=446.

[2] https://www.oracle.com/java/technologies/jdk-mission-control.html.

Fig. 1 Example of a call stack: three methods in a call stack showing the bottom of the stack, found by JFR and the top, found by HPROF. This is a fundamental difference in the way HPROF and JFR approach sampling. JFR will stop walking up the trace at the code that is part of the project being profiled (`MainProgram.begin()`) whereas HPROF will pick up the API function `Java.vector.indexOf()` . This API function is not editable; so we walk back up the trace to find the calling function `MainProgram.begin()`. In some cases, this trace back to the target project is not available. Figure taken from (Watkinson and Brownlee 2023)

```
CPU SAMPLES BEGIN (total = 229) Tue Dec 12 19:30:33 2023
rank self accum count trace method
   1 48.91% 48.91% 112 300154 java.io.FileInputStream.readBytes
   2 2.62% 51.53% 6 300062 java.lang.ClassLoader.defineClass1
   3 2.62% 54.15% 6 300462 org.jcodec.codecs.h264.decode.PredictionMerger.
       weightPrediction
   4 1.75% 55.90% 4 300380 java.lang.Thread.isInterrupted
   5 1.75% 57.64% 4 300453 org.jcodec.codecs.h264.decode.MBlockDecoderBase.
       predictChromaInter
   6 1.31% 58.95% 3 300140 java.util.zip.ZipFile.getEntry
   7 1.31% 60.26% 3 300171 java.io.UnixFileSystem.getBooleanAttributes0
   8 0.87% 61.14% 2 300065 java.lang.ClassLoader.findBootstrapClass
   9 0.87% 62.01% 2 300404 org.jcodec.containers.mp3.MPEGAudioDemuxer.
       skipJunkBB
  10 0.87% 62.88% 2 300433 java.lang.System.arraycopy
  11 0.87% 63.76% 2 300437 org.jcodec.codecs.h264.decode.PredictionMerger.
       mergeWeight
  12 0.87% 64.63% 2 300439 org.jcodec.codecs.h264.decode.DeblockerInput.<
       init>
```

Fig. 2 Example of an HPROF output file: The methods found on top of the call stack in each sample taken by HPROF. Each row corresponds to a unique method, sorted in descending order of appearances. Rank 1 in this case is a Java file reading method that appeared on the top of the stack 112 times. 'Trace' is a unique ID identifying the stack trace (elsewhere in the file) that shows how this method was called. In this example, many Java language and util functions are found on top of the call stack alongside the JCodec core methods

Both profilers count the appearances of methods in the stack. HPROF records and returns the number of times each method is found at the top of the call stack. The methods seen most often are filtered to only those that belong to the target program; these are then returned as *hot* methods. JFR traverses the call stack until it discovers a method that is part of the program being profiled, then, this method's number of samples is incremented. Figure 1 gives a sketch of a situation. This means, a method that is part

of the program being profiled is found in every call stack, whereas, in HPROF samples there may be Java language or I/O operations that are found and not returned as *hot* methods. This is why, as noted by Watkinson and Brownlee (2023), JFR consistently finds more *hot* methods and finds them more times in samples.

These differing approaches impact the final profiling result. Every JFR call stack will be traversed to find a function relating to the program being profiled, this is not the same for HPROF. If two programs call a number of I/O functions and Java language functions HPROF and JFR will return different results. JFR will search past the unrelated functions and identify that the method from the program being profiled is *hot*. HPROF will only identify the function as a *hot* method if it is profiled when it is not running any of the unrelated functions. Even though GI can not edit functions in libraries rather than the program being profiled, manipulating how library functions are called could improve the run time. Part of the motivation for our experiments is to determine how much of a difference to the profiling results there is in practice, within repeat runs of one profiler on one JDK (Sect. 4.1) and as profiler and JDK are changed (Sect. 4.2).

2.2 Memory profiling

Both HPROF and JFR are also able to profile memory use. Gin was extended to include memory profiling by Callan and Petke (2022), as part of work studying multi-objective genetic improvement of software (in that case, studying the trade-off of programs ranging from low CPU time to low memory use). Integration just requires a slight change to the arguments passed to the JVM. For HPROF:

```
-agentlib:hprof=heap=sites,lineno=y,depth=1,interval=$hprofInterval,file=";
```

and for JFR:

```
-XX:+FlightRecorder -XX:StartFlightRecording:jdk.ObjectAllocationInNewTLAB
    #enabled=true,name=Gin,dumponexit=true,settings=profile,filename=
```

The profiling works by tracking objects being allocated to the heap. Each allocation is annotated with a stack trace identifying the original Java code location where the item was allocated. This includes temporary local objects including arrays.

HPROF then outputs a table of samples and corresponding traces, similar to that for CPU time, and are parsed in the same way. The recording file from JFR contains similar data in binary format. The samples and corresponding traces from JFR are embedded in `jdk.ObjectAllocationInNewTLAB` events. The JFR traces are parsed and ranked in order of those responsible for the largest memory allocation.

Table 1 The machines used for the experiments

	Machine A	Machine B
CPU	two 16-core Intel Xeon E5-2620v4 CPUs@2.1GHz	one 8-core Intel i7-11390H CPU@3.40GHz
Memory	32GB DDR4 2133MHz ECC	16GB DDR3 1600MHz non-ECC
OS	Debian 5.10	Ubuntu 22.04

3 Experiments

We (Watkinson and Brownlee 2023) previously profiled Perwendel Spark and reported that, while there was some positive correlation between the results, there was considerable noise and the number of samples for *hot* methods found by the two profilers was also different. JFR found a mean of 26.2 *hot* methods, whereas HPROF found only 21.6. Some of this can be accounted for by the change in JDK as well as the change in profiler. Some can also be attributed simply to noise in the system: run time is notoriously difficult to measure consistently, and at the resolution required to sample methods appearing frequently in the call stack, the problem is worse still. Thus, in the present study, we investigate just how consistent the results of profiling are.

We compare both HPROF and JFR in experiments, with HPROF on JDK 8 and JFR on JDK 8, JDK 9, and JDK 17. We include 9 for its closeness to 8, and 17 as it is the version currently used by Gin, and (until September 2023) the current Long Term Support version of Java. For brevity, we will refer to HPROF/JDK8 as **HPROF8**, JFR/JDK8 as **JFR8**, JFR/JDK9 as **JFR9**, and JFR/JDK17 as **JFR17**.

Note that the ground truth is unknown and, in practice, probably does vary from one JDK to the next, and more so with changes in underlying OS, hardware, and so on. However, we keep these fixed for a given project in our study. Our goal here is simply to understand how much variation there is in different profiling contexts.

3.1 Experimental setup and pipeline

We wished to focus on the default profiling tools that come with Java. This is motivated by the desire to use the profiling tools explored with the Gin GI toolkit, which avoids the use of additional external tools as much as possible. HPROF is freely available with Java Development Kit (JDK) versions 8 and below, whereas JFR is freely available with versions 7 and up. More specifically, our experiments used Oracle JDK 8.0.341 for HPROF and Oracle JDK 1.8.0_202, Oracle JDK 9.0.4, and Oracle JDK 17.0.6 for JFR. OpenJDK 8–9 does not include JFR, so we were limited to the closed-source Oracle JDK. In combination, these will allow us to measure the impact of changing from HPROF to JFR under JDK 8, and changing JDK while running JFR. All experiments were run on the machines listed

Table 2 Which projects were studied with which profilers in our experiments. *The URL for each is prefixed with github.com. X: OpenNLP results with JFR8 for CPU only

Project	GitHub* URL	Version	Machine	HPROF8	JFR8	JFR9	JFR17
Disruptor	/LMAX-Exchange/disruptor	3.4.2	A	•		•	
GSON	/google/gson	2.8.4	A	•	•	•	•
JCodec	/jcodec/jcodec	0.2.0	A	•	•	•	•
JUnit4	/junit-team/junit4	4.13.2	A	•	•	•	•
OpenNLP	/apache/opennlp	1.9.4	A	•	X	•	•
Perwendel Spark	/perwendel/spark	2.9.3	B	•	•	•	

in Table 1, with Table 2 showing which project was run on which machine. The experiments were run sequentially (one profiling run of one project at a time), and no other computationally intensive processes were run in parallel with the experiments.

All profiling is done inside of Gin using its two profiling tools (`gin.util.Profiler` for CPU time and `gin.util.MemoryProfiler` for memory use). These tools output CSV files listing the *hot* methods in descending order, with the sample counts and any associated unit tests. The CSVs generated by our experimental runs are all available in the artefact published with the paper (Brownlee and Watkinson 2024). Gin itself is available from https://github.com/gintool/gin. The specific builds used in our experiments were tag `v2.1` for HPROF8, branch `jdk8-jfr` for JFR8, branch `jdk9` for JFR9, and commit `2359f57` from the current trunk for JFR17.

We considered six open source Java projects, listed in Table 2, with their GitHub repository URL and the specific build we worked with. The projects selected were drawn from those in two previous studies with Gin (Petke et al. 2023; Watkinson and Brownlee 2023), and were originally chosen using the criteria that they used the Maven or Gradle build tools, have a non-trivial test suite (taking minutes to hours to run during profiling) with only passing tests, are reasonably popular (> 1000 stars (i.e., bookmarked by over 1000 users) and > 300 forks), open-source with a permissive licence, and crucially, compatible with Java 8 to allow comparisons with HPROF. As we wanted to keep the comparisons as fair as possible, no changes were made by us to the projects to make them compatible with particular Java versions. Consequently, in some cases older builds were chosen to allow compatibility with Java 8 for comparisons and, even then, not all builds worked with all Java versions in the study without modification. The Java versions that each project was tested on are indicated in Table 1. All projects used the Maven build tool, except Disruptor, which used Gradle.

The experimental pipeline for each project listed in Table 2 was as follows:

1. reboot the host machine
2. switch the default Java and `JAVA_HOME` to the appropriate JDK
3. clone the project from GitHub
4. build and test the project using one of Maven or Gradle

5. run shell script calling the Gin profiler 20 times (running the test suite for the target application); each run of the test suite invokes a separate, fresh Java virtual machine instance
6. analyse resulting CSVs

Profiling was carried out on runs of the test suite included with each project. While not necessarily reflecting the real-world usage patterns of the projects, for the purpose of testing profiling these exercise large portions of the projects and so represent a reasonable way to compare the results of profiling itself. Inevitably, there will be some changes to the profiles over the course of the repeat runs due to cache filling, but the same approach was used for all projects and all profiler configurations. For any one project the impact should be comparable across the profilers tested. Analysis of profiling consistency is given in Section 4.1.

3.2 Approach to analysis

The profilers return ranked lists of methods in a CSV (20 lists per profiler per project). We are interested in the similarity of these lists: both within the 20 lists produced by one profiler (RQ1 and RQ4), and comparing the 20 lists produced by one profiler and the 20 produced by another (RQ2). To do this, we consider each pair of lists drawn from the 20. That is, when comparing the 20 lists $(l_1, l_2, \ldots l_{20})$ produced by one profiler, we measure the similarity of the pair l_1 vs l_2, the pair l_1 vs l_3, the pair l_1 vs l_4 ... etc, giving us 190 (20 choose 2) comparisons. We then report summary statistics over these 190 measurements. When comparing two profilers, we have two sets of 20 ranked lists to compare $(l_1, l_2, \ldots l_{20})$ and $(m_1, m_2, \ldots m_{20})$. We measure the similarity of all combinations of this: i.e., the pair l_1 vs m_1, the pair l_1 vs m_2, the pair l_1 vs m_3...etc., giving us 400 (20×20) comparisons. Again we report summary statistics over all 400 comparisons.

The specific similarity measures used need to consider that the rank order of methods varies between repeat runs and the lists rarely contain precisely the same set of methods. It is trivial to filter the sets of methods to only those that appear a certain number of times, and this could be a way to reduce noise in the results. Consequently, where we consider metrics calculated across the repeat runs for a profiler, we filter the list of *hot* methods found by each run according to the following three sets. UNION is the set of methods identified as *hot* in any of the repeat runs of a profiler, that is, the case where no filtering is applied. MEDIAN is the set of methods identified as *hot* in at least half (i.e., 10) of the repeat runs of a profiler. INTERSECTION is the set of methods identified as *hot* in all 20 of the repeat runs of a profiler. While we apply the above filtering, we still keep the rank orderings associated with each method in the set, in order to apply the metrics described in the remainder of this section.

The variation in rank order and methods present across repeat runs also leads us to use a method known as Weighted Rank Biased Overlap (WRBO) (Webber et al. 2010) as the principal means to compare the rankings generated by the profilers. The ability of WRBO to both handle lists of varying length and not require that the

lists contain the same elements makes it preferable to well-known rank-comparison methods such as Spearman Rank Correlation or Kendall Tau. WRBO has previously been used to compare feature importance rankings generated by Explainable AI algorithms (Sarica et al. 2022). WRBO can also assign a higher weight to the first few elements in a list; we use a weighting of 0.9 which results in the top 10 *hot* methods being responsible for 85.56% to the total scoring. The method returns a score in the range [0, 1], with 1 being a perfect overlap of identical lists and 0 being no similarity between both the order of the list elements, and the number of shared elements. We used a Python implementation of the WRBO function provided by Raikar (2023).

It is also possible to further filter the UNION, MEDIAN, and INTERSECTION sets to include only those methods identified by both profilers being compared. This allows us to compute a simple arithmetic mean ranking for each member of the sets across all repeat runs, then use scatterplots and the more well-known Spearman Rank Correlation for comparisons, at the cost of significant loss of information through aggregation and exclusion of methods not common to both sets. This latter approach was taken in our previous paper (Watkinson and Brownlee 2023). These results appear later in the discussion (Sects. 4.2 and 4.4).

4 Results

We now present the results of our experiments and answers to the research questions. The following sections report summary statistics and a subset of examples reflecting the trends we have observed across all the experiments. The full set of results and visualisations is available from https://github.com/MylesWatkinson/replication_package and (Brownlee and Watkinson 2024).

4.1 RQ1. How consistently does each profiler identify *hot* methods between repeat runs on a given JDK?

GI pipelines generally use profiling to identify *hot* methods, i.e., the methods where the CPU spends most time and so are most fertile for improvement. These are the methods to be targeted by mutations. Yet measurement of CPU time is notoriously noisy so we begin by asking how consistently any of the profilers in our study identify *hot* methods for a given project over multiple runs.

Table 3 reports the counts of *hot* methods in each of the four sets for each project and profiler. The MEDIAN lists contain 10–30% of the methods in UNION, and INTERSECTION contains around 10–60% of MEDIAN. In summary, well over half the methods identified across all repeat runs appear in fewer than half of them, and a smaller fraction still appear in all repeat runs. Thus we confirm that there is considerable noise in the results for each profiler and JDK combination on each project. The histograms in Fig. 4 show the distribution of appearances for *hot* methods for JUnit4. These reflect the trend seen across all projects in the study. The figures show how often unique methods are identified as *hot* over the 20 repeat runs, each bar being the number of *hot* methods that

Table 3 Count of *hot* methods identified by each profiler and JDK combination within the UNION, MEDIAN, and INTERSECTION sets for each project. The percentages show what portion of UNION were in MEDIAN, and what portion of MEDIAN were in INTERSECTION for each profiler

	Project	Hot method count			
		HPROF8	JFR8	JFR9	JFR17
UNION	Disruptor	56	–	109	–
	Gson	146	94	109	104
	Jcodec	845	830	830	635
	Junit4	470	287	285	311
	OpenNLP	587	442	483	499
	Perwendel Spark	102	113	95	–
MEDIAN	Disruptor	5 (9%)	–	39 (36%)	–
	Gson	32 (22%)	27 (29%)	27 (25%)	17 (16%)
	Jcodec	191 (23%)	257 (31%)	244 (29%)	157 (25%)
	Junit4	154 (33%)	95 (33%)	102 (36%)	105 (34%)
	OpenNLP	158 (27%)	156 (35%)	169 (35%)	170 (34%)
	Perwendel Spark	16 (16%)	25 (22%)	20 (21%)	–
INTERSECTION	Disruptor	3 (60%)	-	13 (33%)	–
	Gson	7 (22%)	5 (19%)	4 (15%)	2 (12%)
	Jcodec	50 (26%)	72 (28%)	74 (30%)	39 (25%)
	Junit4	53 (34%)	42 (44%)	49 (48%)	50 (48%)
	OpenNLP	61 (39%)	80 (51%)	78 (46%)	81 (48%)
	Perwendel Spark	4 (25%)	14 (56%)	10 (50%)	–

appeared $1, 2, \ldots, 20$ times. There is a peak at 20: a core set of *hot* methods that are identified by the profiler in every repeat run. There is then a long tail tending towards a large number of methods that are identified as *hot* in only one of the repeat runs. The key point here is that even a limited number of repeat runs will be able to identify the core set of *hot* methods and filter out those methods identified as *hot* purely as a result of sampling artefacts.

How many repeats might be necessary in practice? This will depend on the confidence we wish to have that the methods identified by profiling are truly *hot*, and what our definition of *hot* is in the first place. If we assume that we wish to detect all methods that would be identified as *hot* in at least half of all profiling runs, then we can say that the associated probability of *hot* ness $p = 0.5$. The binomial distribution then tells us the probability $P(x)$ that the method will be identified as *hot* x times in a set of n profiling runs is:

$$P(x) = \binom{n}{x} p^x q^{n-x} \tag{1}$$

We can rearrange this inequality to tell us the minimum number of repeat runs required to identify the method as *hot* with a 95% probability or, in practice, that it will not be identified as *hot* in any repeat runs with a probability of < 0.05. i.e.,:

$$\binom{n}{0} 0.5^0 0.5^{n-0} < 0.05 \tag{2}$$

Solving Eq. (2) for n, we find that $n \geq 4.32$. So five repeat runs of the profiler gives us a 95% confidence that we will detect all *hot* methods by taking the UNION across those five repeat runs. Of course, the sample size will change if the probability of detection that we consider to be *hot* is different.

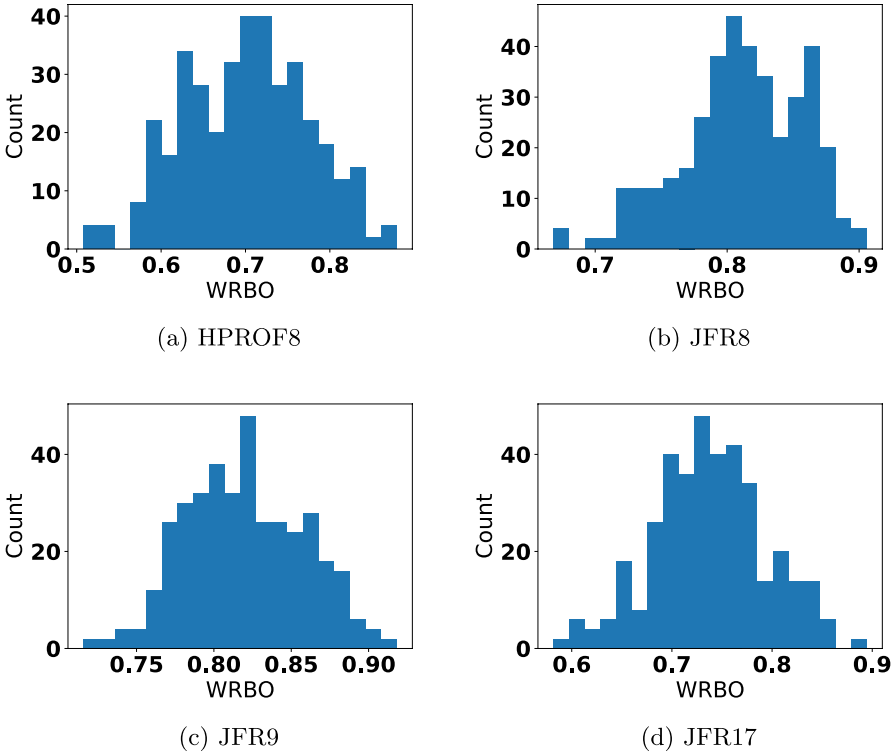
How much do the ranked lists of *hot* methods generated by repeat runs of the same profiler on the same JDK overlap? That is, how much noise is there among the runs of a single profiler? We calculated the WRBO between all pairs of ranked lists within the 20 produced by repeat runs of a given profiler, the lists being filtered to match UNION (in practice, no filtering) and INTERSECTION sets. The median and interquartile range of these WRBO figures are given in Table 4.

For most projects, the median WRBO is over 0.65 with a low Interquartile Range (IQR), indicating that all pairs of profiling results among the repeat runs of one profiler give similar ranks to the *hot* methods. Only Gson and Perwendel Spark differ, with Gson having a low WRBO for JFR17 and Perwendel Spark having a high IQR for the UNION results with HPROF8. Closer inspection revealed that a small number of repeat runs identified dramatically different orderings of the top few *hot* methods, skewing the results for these specific project/profiler combinations. The top-ranked *hot* methods for these two projects tended to have only a small (fewer than 10) number of samples during the profiling run so only a small amount of noise had a large impact on the overall ranking. To avoid this issue causing uncertainty in profiling results in practice, a mitigation could be to repeat runs of each unit test during the profiling.

In Table 4 it can be seen that JFR versions typically achieve higher WRBO between repeat runs on one JDK than HPROF. This is true for both UNION methods and INTERSECTION methods. Further, JFR results generally have lower IQR than those of HPROF, meaning that in most cases all pairings of profiling results for a given profiler have a high WRBO (having distributions like those in Fig. 3). Thus we can conclude that JFR produces more consistent rankings of *hot* methods than HPROF does.

WRBO is higher for the INTERSECTION sets. These are filtered to only the methods identified *hot* by every repeat of the profiler, so the random noise from the tail end of the results is removed. Among these *hot* methods, the rank order is also generally more consistent. The highest value of 1.0 is seen for Disruptor on HPROF8. Here, there are only three methods in INTERSECTION, that is, appearing in every repeat run. These three appear in the same order in every repeat run, leading to a WRBO of 1.0. For UNION and MEDIAN, WRBO is reduced because it then includes the many other methods appearing in only some runs, often in a differing orders.

Impact of rebooting Our experimental pipeline did not feature a reboot between repeat runs of each compiler. This could result in some drift in the results as CPU caches become filled. To determine the impact of this issue, we looked at the ranks allocated to each *hot* method in each repeat run. Figure 5 shows these ranks as a heat map. We show the results for Gson as it was run on all profilers and had the fewest *hot* methods, making the visualisation a reasonable size, but the results are similar



(a) HPROF8 (b) JFR8

(c) JFR9 (d) JFR17

Fig. 3 Distribution of WRBO values for all 190 pairs of ranked lists from 20 repeat runs when profiling JCodec with each profiler/JDK combination

for all projects. The methods (rows) are sorted by order of first appearance, so we see those appearing in higher ranks in the first profiler runs near the top. Near the bottom we see methods that were not ranked at all in the first 18–19 repeats, and only appeared on the last runs of the profiler. If there was a large impact from caching on the results, we would expect to see many rows that are either: dark on the left, steadily getting lighter to the right, as particular methods start out being identified as *hot* then gradually dropping in the rankings, or light on the left changing to dark on the right, as new methods start to be identified as *hot*. Instead we see some methods being consistently dark across the whole row, or a few dark points spread uniformly (subject to statistical noise) across the rows. The same figures for the other projects are in our artefact (Brownlee and Watkinson 2024).

Overall, these results give some confidence that there is not much drift over the course of the repeat runs.

In summary, there is a high degree of consistency between the ranked lists of *hot* methods identified by a given profiler on a single JVM, with JFR on any of the three JDKs tested producing more consistent results than HPROF. While there is noise among the repeat runs of any one profiler, this can be filtered by removing the tail of methods that only appear once across all repeat runs.

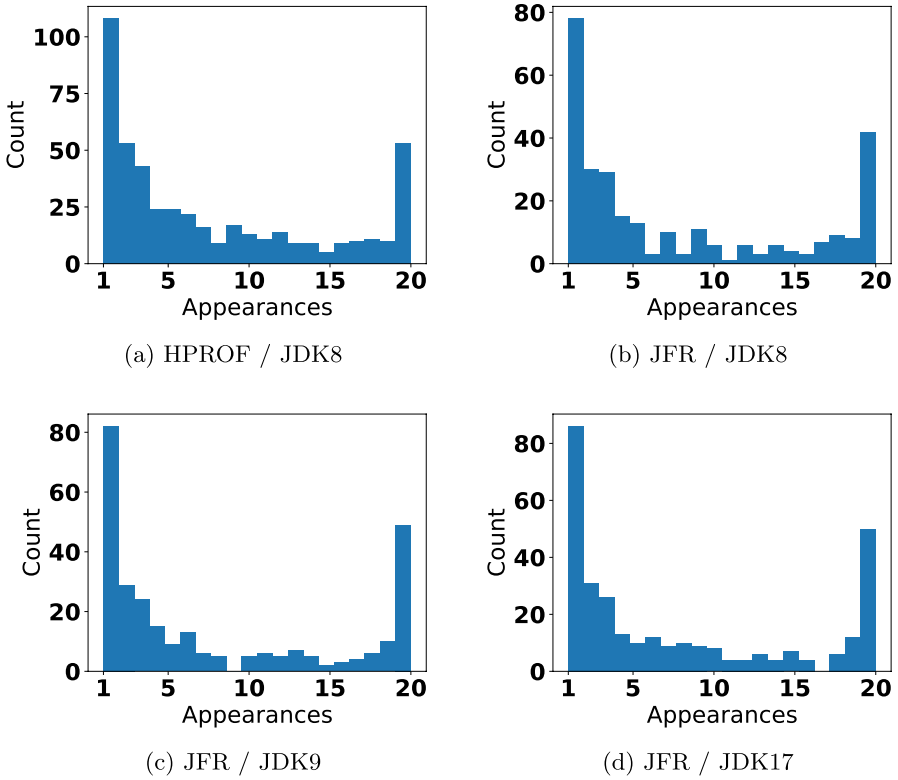
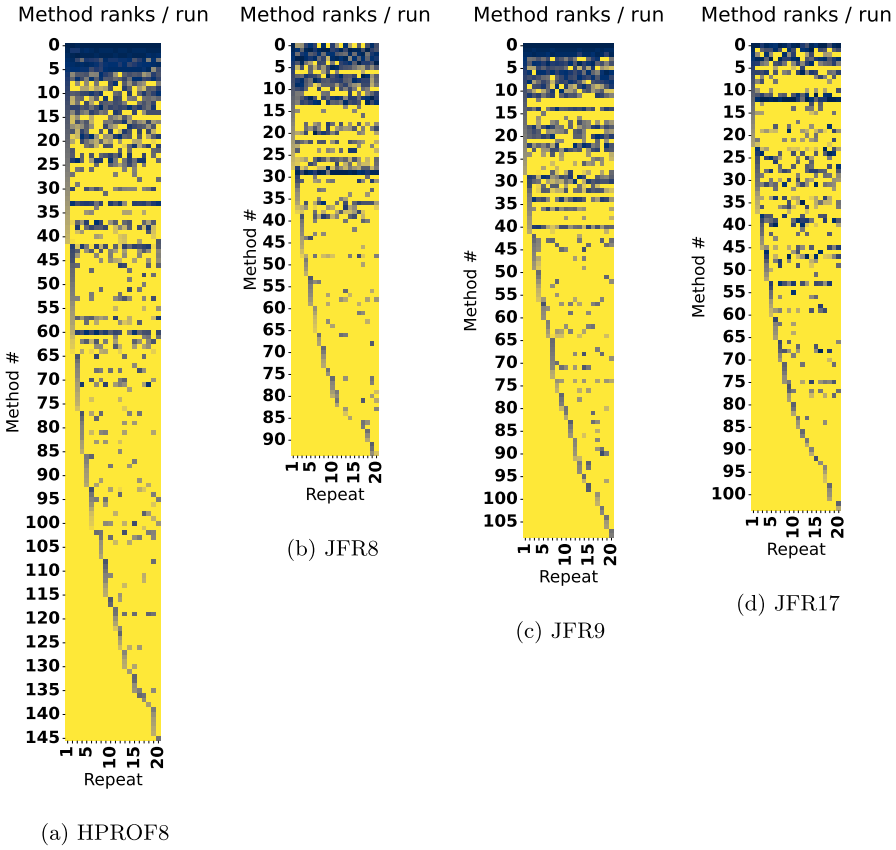


Fig. 4 Distribution of *hot* method appearances across repeat runs of profilers for JUnit4, i.e., how often we see methods appearing only once, twice, up to 20 times

Table 4 WRBO within repeat runs of each profiler. Each figure is the median with interquartile range in brackets. Values closer to 1 suggest that the set of methods identified as *hot* and their rank ordering are highly consistent across all repeat runs of the profiler

	Project	Median WRBO (IQR)			
		HPROF8	JFR8	JFR9	JFR17
UNION	Disruptor	0.662 (0.104)	–	0.849 (0.053)	–
	Gson	0.704 (0.081)	0.446 (0.133)	0.697 (0.078)	0.389 (0.173)
	Jcodec	0.707 (0.111)	0.812 (0.062)	0.820 (0.057)	0.737 (0.072)
	Junit4	0.892 (0.031)	0.760 (0.103)	0.899 (0.030)	0.914 (0.081)
	OpenNLP	0.696 (0.163)	0.912 (0.049)	0.887 (0.056)	0.859 (0.105)
	Perwendel Spark	0.743 (0.458)	0.830 (0.040)	0.824 (0.047)	–
INTERSECTION	Disruptor	1.000 (0.000)	–	0.923 (0.049)	–
	Gson	0.932 (0.058)	0.828 (0.100)	0.973 (0.045)	0.900 (0.100)
	Jcodec	0.719 (0.109)	0.813 (0.063)	0.823 (0.062)	0.767 (0.072)
	Junit4	0.896 (0.031)	0.774 (0.101)	0.907 (0.030)	0.919 (0.082)
	OpenNLP	0.699 (0.056)	0.912 (0.049)	0.887 (0.056)	0.859 (0.105)
	Perwendel Spark	0.928 (0.145)	0.896 (0.038)	0.918 (0.043)	–



(a) HPROF8

(b) JFR8

(c) JFR9

(d) JFR17

Fig. 5 Heatmaps showing the ranks allocated to methods in each repeat run for GSON with the experimental pipeline used througout the rest of the paper (i.e., without rebooting between runs). Darker means closer to rank 1 (most CPU-intensive); bright yellow means not ranked at all. There is one row per method identified as *hot*, sorted in order of first appearance in the profiling data. There are 20 columns, one for each repeat of the profiler, sorted in time order

4.2 RQ2. Is there a difference in the *hot* methods identified by each profiler?

The profiler and JDK combinations studied each produce generally consistent results across repeat runs but what happens when we exchange one profiler/JDK pair for another? As software is updated to support new Java versions, the results reported for older JDKs may become invalid. It might be tempting in a GI study to take a standard set of *hot* methods, and test a new GI technique for them, but if the JDK is different, these methods may no longer be *hot*. Why does this matter? For experimentation in GI we might expect to do, e.g., landscape analysis, on the same projects in different contexts. More broadly, though, it is useful to know how tied to a particular JDK and profiler any experimental results are going to be. We now consider the differences in rankings produced by different profiling tools, under different JDKs.

Table 5 WRBO between repeat runs of different profilers

	Project	Median WRBO (IQR)				
		HPROF8 vs JFR8	HPROF8 vs JFR9	JFR8 vs JFR9	JFR8 vs JFR17	JFR9 vs JFR17
UNION	Disruptor	–	0.492 (0.113)	–	–	–
	Gson	0.191 (0.079)	0.173 (0.065)	0.320 (0.086)	0.117 (0.074)	0.106 (0.071)
	Jcodec	0.148 (0.055)	0.140 (0.059)	0.796 (0.059)	0.578 (0.054)	0.601 (0.055)
	Junit4	0.177 (0.023)	0.210 (0.029)	0.701 (0.047)	0.682 (0.055)	0.687 (0.019)
	OpenNLP	0.218 (0.042)	0.346 (0.052)	0.549 (0.021)	0.211 (0.015)	0.532 (0.049)
	Perwendel Spark	0.202 (0.064)	0.493 (0.079)	0.560 (0.040)	–	–
INTERSECTION	Disruptor	-	0.404 (0.017)	–	–	–
	Gson	0.828 (0.100)	0.099 (0.031)	0.149 (0.000)	–	–
	Jcodec	0.151 (0.056)	0.144 (0.059)	0.796 (0.059)	0.574 (0.052)	0.598 (0.050)
	Junit4	0.175 (0.025)	0.210 (0.028)	0.705 (0.049)	0.687 (0.056)	0.690 (0.020)
	OpenNLP	0.219 (0.043)	0.346 (0.052)	0.549 (0.021)	0.211 (0.015)	0.532 (0.049)
	Perwendel Spark	0.148 (0.045)	0.348 (0.041)	0.552 (0.048)	–	–

Table 5 reports the WRBO figures comparing all pairs of ranked lists produced by the repeat runs of each profiler. WRBO between JFR variants is highest, around 0.6 in UNION and INTERSECTION sets. Similarity between lists generated by HPROF and JFR8 (and, we observed, also JFR9 and JFR17) is low. So the *hot* methods found by JFR under any of JDK8, 9, and 17 have more overlap with each other than with the set of methods returned by HPROF. Differences between UNION and INTERSECTION are not consistent enough to say whether one of the two sets has more similarity over different profilers.

The low WRBO figures are largely due to different methods appearing in the sets. For example, with JCodec, INTERSECTION contained 50 *hot* methods for HPROF8 and 72 *hot* methods for JFR8. Of these, 42 were found by JFR but not HPROF, 20 were found by HPROF but not JFR, and 30 were found by both. With Gson, only one of the *hot* methods identified by HPROF was identified by JFR9, and none of the JFR17 *hot* methods were identified by the other two profiler runs.

Figure 6 shows scatter plots comparing the mean ranks of methods common to each pair of profilers. The means were computed over the set of ranks for each method (so runs that did not count a method as *hot* were not included in the statistics for that method). Across UNION, there is a strong correlation between method ranks as seen in Figs. 6a,c. In contrast, in the INTERSECTION scatter plots there seems to be almost no visible correlation between HPROF8 and JFR8 ranks.

The strongest trend in the UNION plots is on the right side with those methods having lower ranks/higher numbers. For HPROF8 vs JFR8 there is significant noise on the left side of the graph where the 'hotter' methods sit. This explains the low WRBO values, which place more weight on the highest-ranked methods.

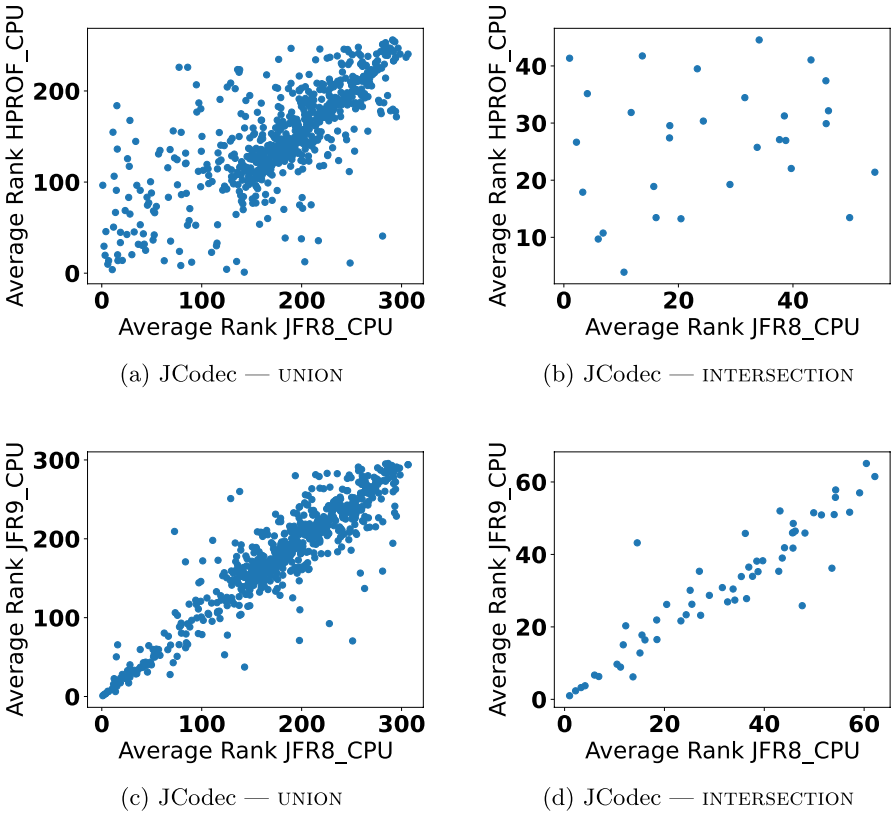
Table 6 Spearman correlation between mean ranks over repeat runs of different profilers. For Gson, the INTERSECTION sets had no overlap for JFR9 vs JFR17 and HPROF8 vs JFR17, and a single common *hot* method for HPROF8 vs JFR9, so it was impossible to compute a correlation (though it was possible to compute WRBO for Table 5 as WRBO does not require perfectly overlapping lists)

	Project	Spearman correlation				
		HPROF8 vs JFR8	HPROF8 vs JFR9	JFR8 vs JFR9	JFR8 vs JFR17	JFR9 vs JFR17
UNION	Disruptor	–	0.514	–	–	–
	Gson	0.650	0.661	0.822	0.530	0.506
	Jcodec	0.759	0.756	0.888	0.784	0.808
	Junit4	0.631	0.670	0.901	0.852	0.858
	OpenNLP	0.585	0.607	0.839	0.617	0.765
	Perwendel Spark	0.740	0.682	0.864	–	–
INTERSECTION	Disruptor	–	1.000	–	–	–
	Gson	–	–	–	–	–
	Jcodec	0.157	0.101	0.915	0.664	0.850
	Junit4	0.006	0.329	0.808	0.827	0.712
	OpenNLP	0.657	0.537	0.884	0.783	0.803
	Perwendel Spark	1.000	0.700	0.786	–	–

Table 6 reports the Spearman correlation for the mean ranks. Across UNION methods, the Spearman correlation coefficient is high for all pairs of profilers. Though it is slightly lower for HPROF8 vs JFR8, it still always exceeds 0.6. For INTERSECTION, Spearman correlation is high between all JFR runs, but drops dramatically for HPROF8 vs JFR8. The one exception (Perwendel Spark, with a Spearman correlation of 1.0 for HPROF8 vs JFR8 on INTERSECTION) is because in that case INTERSECTION only contained two *hot* methods.

When moving from UNION to MEDIAN and then to INTERSECTION, the number of *hot* methods decreases greatly, as we move towards methods that only appear consistently across repeat runs. When looking at JCodec's HPROF8 vs JFR8 results, the UNION methods span ranks 0 to 300, MEDIAN methods span 0 to 200 and intersection methods only fall between ranks 0 to 40. Aligning this with the plots in Fig. 6b there seems to be almost no correlation between HPROF and JFR8 INTERSECTION methods, whereas, the UNION set exhibits a clearer trend. Though, this trend is more apparent on the right side of the plots where lower ranked methods are present.

Methods found in higher mean ranks have less consistent ranks between runs than those found in lower ranks. It would seem that the hottest methods vary considerably from one profiler to the other but less so when changing JDK. This is consistent with the findings of Mytkowicz et al. (2010), in that moving from one profiler to another produces quite different results. This is a little suprising, as we might expect the optimisations introduced by a new JDK to also have a large effect on the hottest methods where the CPU spends most time. However, it is the case that the larger jump in JDK (8 to 17) does lead to a lower WRBO and correlation, albeit not as low as the change from JFR to HPROF.



(a) JCodec — UNION (b) JCodec — INTERSECTION

(c) JCodec — UNION (d) JCodec — INTERSECTION

Fig. 6 Mean ranks of *hot* methods for JCodec: HPROF8 vs JFR8 and JFR8 vs JFR9. Each of the points represents one method, and it denotes the mean ranks returned by either profiler, taken over the 20 repeat runs. Only methods found by both profilers are included. 'Union' means methods identified by at least one repeat run of the profiler, 'Intersection' means methods identified by all repeat runs. The UNION plots generally show a strong linear relationship between ranks from one profiler and another; where this is driven by the long tail of low-ranked seldon-appearing methods the WRBO will be low but Spearman correlation high (Tables 5 and 6). If the hottest methods are ranked similarly then WRBO and Spearman will both be high. Some INTERSECTION plots show very little relationship as is seen in Fig. 6b; the methods consistently identified by one profiler are different to those from another. Others (mostly when comparing JFR on different JDKs) have a much stronger relationship (e.g., Fig. 6d), implying that the methods are ranked similarly

This higher consistency for low-ranked methods may come not only from profilers but from Gin's internal ranking method. Gin resolves ties in method counts by ranking methods in the order they are seen.

If both profilers in a pairing returned identical results, the scatter plot would resemble the function: $x = y$. How each point on the graph differs from this function may explain where differences in each profiler arises.

First, the plots in Fig. 6c can be broken into 3 sections using x, mean JFR8 rank, and y, mean HPROF8 rank. The middle, where $x = y$, the top, where $x < y$ and the bottom, where $x > y$. The middle is the ideal where there are consistent results

between the two profilers. Points in the top section occur when JFR8 finds methods in a lower rank (i.e., higher number) than HPROF8 and *vice versa* for those in the upper ranks. Over all projects, more points exist in the top half than the bottom. That is, HPROF8 generally allocates lower ranks to methods also identified as *hot* by JFR8. After accumulating and taking the mean difference between x and y coordinates of points on the plot: mean rank of union methods between HPROF and JFR8 for JCodec, the function found is $1.6x = y$. Even though JFR finds more *hot* methods and therefore has a larger range of potential *hot* method ranks HPROF8 coordinates are 1.6 times larger.

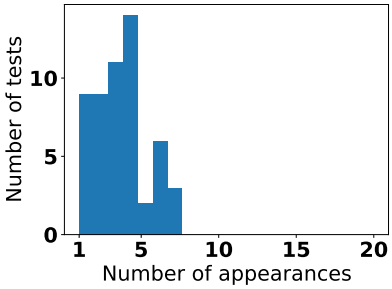
These asymmetric errors suggest that the differences in profiling results between HPROF8 and JFR8 are not due to uniform random noise. There are two reasons methods may exist in the top half. Either HPROF8 does not profile long running methods enough or JFR8 profiles methods too much. As both profilers have the same snapshot interval it is unlikely that JFR8 profiles methods more times than it should. Likely, the cause is noise when profiling. This noise can not be uniformly random over both profilers as the errors are assymetric. Therefore, it can be ascertained HPROF8's profiling is more affected by the noise of seldom-appearing methods than JFR8.

Overall then, changing the profiler and JDK combination rarely produces a similar set of results, though changes from one JDK to another have less impact if the profiler is kept the same. It is crucial to repeat the profiling to re-identify the *hot* methods for a given application should either profiler or JDK change.

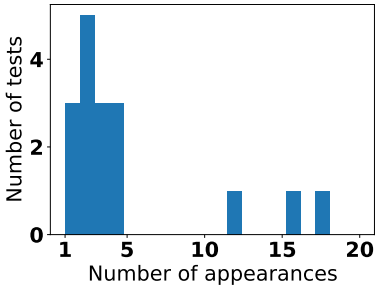
4.3 RQ3. How consistently does each profiler identify unit tests associated with *hot* methods?

One of the features of Gin is the ability to identify tests that execute each *hot* method; those tests responsible for the method being called each time it is seen on the stack. This is a form of Regression Test Selection (RTS) (Yoo and Harman 2012). The idea is to determine the parts of the test suite relevant to each *hot* method, so that only those rather than the whole suite need to be run during the GI search. A similar approach was used by Harrand et al. (2019) when selecting a location to make edits. They suggested edits should only be made in areas covered by a test case otherwise dead code will be edited. We now consider the tests identified by the profiler for each *hot* method in our experiments. To simplify the analysis, here we only consider the INTERSECTION *hot* methods for each profiler, that is, those methods identified by every repeat run of the profiler. For each *hot* method, we examined the 20 sets of tests that were identified by the profiler for it (each repeat run of the profiler identifying one set of tests for that method). We counted the number of times each test appears for each *hot* method.

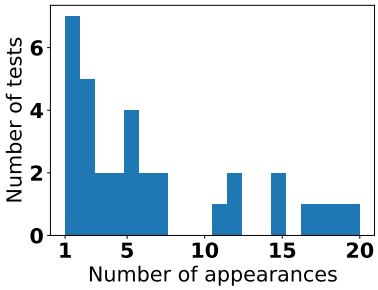
The full set of histograms reporting the distribution of the number of appearances of tests for each *hot* method for each project and profiler is included in our supporting data artefact (Brownlee and Watkinson 2024). These are hugely varied but four examples are given in Figure 7 capturing the most common types of distribution



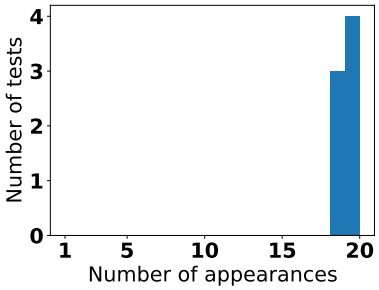
(a) Hot Method Type A: no single unit test identified as calling this method in more than 10 repeats of the profiler.



(b) Hot Method Type B: many tests identified as calling this method in one or two repeats of the profiler, none identified in all 20 repeats.



(c) Hot Method Type C: many tests identified as calling this method under 5 repeats of the profiler, with some identified in all 20 repeats.



(d) Hot Method Type D: all tests 20 identified as calling this method in most (19–20) repeats of the profiler.

Fig. 7 Typical distributions of test appearances. OpenNLP, JFR9. Each plot is for a different *hot* method and shows the rate at which tests were detected as calling that method method

seen. For OpenNLP, there were 57 *hot* methods identified by the profiling within UNION. Of these:

- 6 of the *hot* methods had the distribution shown in Fig. 7a, where many unit tests were only identified by one of the repeat runs of the profiler, and none appeared in 10 or more repeats.
- 20 of the 57 followed Fig. 7b: many tests having one appearance, some tests appearing in more than 10 of the repeats, but none in all 20 repeats.
- 22 of the 57 followed Fig. 7c: as above, but with some tests appearing in all 20 repeats.
- 9 of the 57 followed Fig. 7d, where all tests identified appeared in nearly all repeats.

The last of these four categories is the most desirable: the tests being identified by following the traces produced by the profiler consistently enough to lend some confidence that they are actually of use. In most examples tests only appear occasionally

Table 7 WRBO within repeat runs of each profiler when investigating memory footprint. Each figure is the median with interquartile range (IQR) in brackets. Values closer to 1 suggest that the set of methods identified as *hot* and their rank ordering are highly consistent across all repeat runs of the profiler

	Project	Median WRBO (IQR)			
		HPROF8	JFR8	JFR9	JFR17
UNION	Disruptor	1.000 (0.000)	–	0.945 (0.024)	–
	Gson	0.818 (0.130)	0.895 (0.105)	0.816 (0.053)	0.692 (0.102)
	Jcodec	0.751 (0.110)	0.812 (0.062)	0.911 (0.031)	0.871 (0.042)
	Junit4	0.908 (0.021)	0.945 (0.138)	0.840 (0.133)	0.893 (0.025)
	OpenNLP	0.936 (0.027)	–	0.952 (0.034)	0.932 (0.042)
	Perwendel Spark	0.662 (0.104)	0.950 (0.026)	0.981 (0.020)	–
INTERSECTION	Disruptor	1.000 (0.000)	–	0.977 (0.027)	–
	Gson	1.000 (0.000)	0.956 (0.093)	0.838 (0.055)	0.716 (0.100)
	Jcodec	0.800 (0.108)	0.813 (0.063)	0.915 (0.031)	0.871 (0.042)
	Junit4	0.962 (0.022)	0.945 (0.138)	0.891 (0.057)	0.895 (0.025)
	OpenNLP	0.937 (0.026)	–	0.952 (0.034)	0.932 (0.041)
	Perwendel Spark	0.983 (0.018)	0.969 (0.027)	0.984 (0.021)	–

regardless of profiler. This clearly suggests that we should not use such an approach to select subsets of the test suite: at a minimum, in the context of GI, one should still follow the good practice of running the entire test suite on any patched code or employ established RTS techniques, e.g., (Yoo and Harman 2012; Guizzo et al. 2021).

4.4 RQ4. Memory

We now briefly explore the consistency of profiling for memory. The same profilers are able to sample object allocation events in the JVM to match memory allocation to specific functions. Functions allocating the most memory throughout the recording process will be returned as described in Sect. 2.2.

The distributions of method appearances in memory profiling results, in Fig. 8, are similar to those seen for CPU use in Fig. 4. Several methods are identified as *hot* in terms of memory consistently across all repeat runs for each project, with a long tail of methods appearing less frequently.

Each profiler still shows some consistency in ranking the hottest methods. Table 7 shows that WRBO within repeat runs of a single profiler/JDK combination is over 0.69 in all cases. Comparisons of different profiler/JDK combinations are given in Tables 8 and 9, where we report WRBO and Spearman correlation for each pair of profilers, for UNION. WRBO values for HPROF8 vs JFR8 are low, but higher than was seen for CPU use (Table 5), suggesting more overlap in the *hot* methods each profiler found. In contrast, WRBO was lower than observed for CPU between JFR on different JDKs, suggesting that changes to the JDK made more of an impact on profiles for memory than for CPU. Spearman values vary considerably per project, though generally following the trend of WRBO. Exceptions (low WRBO, high

Table 8 Median WRBO between all pairs of repeat runs of different profilers when investigating memory footprint. Profiler results for all methods in UNION

Project	Median WRBO (IQR)				
	HPROF8 vs JFR8	HPROF8 vs JFR9	JFR8 vs JFR9	JFR8 vs JFR17	JFR9 vs JFR17
Disruptor	–	0.181 (0.001)	–	–	–
Gson	0.316 (0.033)	0.092 (0.044)	0.188 (0.034)	0.055 (0.029)	0.182 (0.037)
Jcodec	0.191 (0.089)	0.675 (0.117)	0.156 (0.015)	0.171 (0.040)	0.844 (0.044)
Junit4	0.341 (0.083)	0.529 (0.107)	0.408 (0.107)	0.329 (0.081)	0.568 (0.103)
OpenNLP	–	0.731 (0.019)	–	–	0.737 (0.046)
Perwendel Spark	0.543 (0.014)	0.750 (0.010)	0.464 (0.007)	–	–

Table 9 Spearman correlation between mean ranks over repeat runs of different profilers when investigating memory footprint. Profiler results for all methods in UNION

Project	Spearman				
	HPROF8 vs JFR8	HPROF8 vs JFR9	JFR8 vs JFR9	JFR8 vs JFR17	JFR9 vs JFR17
Disruptor	–	-1.000	–	–	–
Gson	0.500	-0.607	0.189	0.212	0.576
Jcodec	0.137	0.605	0.452	0.398	0.792
Junit4	0.235	0.549	0.118	0.111	0.728
OpenNLP	–	0.734	–	–	0.841
Perwendel Spark	0.619	0.797	0.518	–	–

Spearman, for Gson on JFR9 vs JFR17, and JCodec on JFR8 vs JFR9 and JFR9 vs JFR17, were driven by very low overlaps between the *hot* methods identified by the two profilers, where those which were identified by both were ranked similarly.) Given this variability, the best practice would still be as noted earlier for CPU: rerun the profiler for each new profiler/JDK, and keep the most frequently appearing methods across the repeat runs.

In order to confirm that the two profilers do identify different sets of *hot* methods, we also tried comparing the results for CPU and memory profiling when using the same profiler and JDK. WRBO and Spearman correlation comparing the ranks of methods identified for CPU and memory use, by each profiler, are given in Tables 10 and 11. While in most cases there is a weak positive correlation between *hot* methods for CPU and memory, the results vary considerably. For example, for Gson, the Spearman correlation between mean ranks of *hot* methods in UNION for CPU and memory were: 0.596 for JFR17, 0.751 for JFR9, and −0.786 (i.e., negative) for HPROF8. For JCodec, those figures were 0.294, 0.427, and 0.422 respectively. The corresponding scatter plots showing how method ranks compare for

Table 10 Median WRBO between all pairs of repeat runs of each profiler when investigating CPU use and memory use. Results for all methods in UNION

Project	Median WRBO (IQR)			
	HPROF8	JFR8	JFR9	JFR17
Disruptor	0.220 (0.050)	–	0.432 (0.023)	–
Gson	0.044 (0.025)	0.331 (0.072)	0.630 (0.064)	0.316 (0.160)
Jcodec	0.178 (0.055)	0.014 (0.037)	0.054 (0.028)	0.015 (0.037)
Junit4	0.217 (0.017)	0.248 (0.077)	0.463 (0.100)	0.524 (0.034)
OpenNLP	0.289 (0.054)	–	0.276 (0.023)	0.340 (0.050)
Perwendel Spark	0.174 (0.043)	0.415 (0.044)	0.605 (0.057)	–

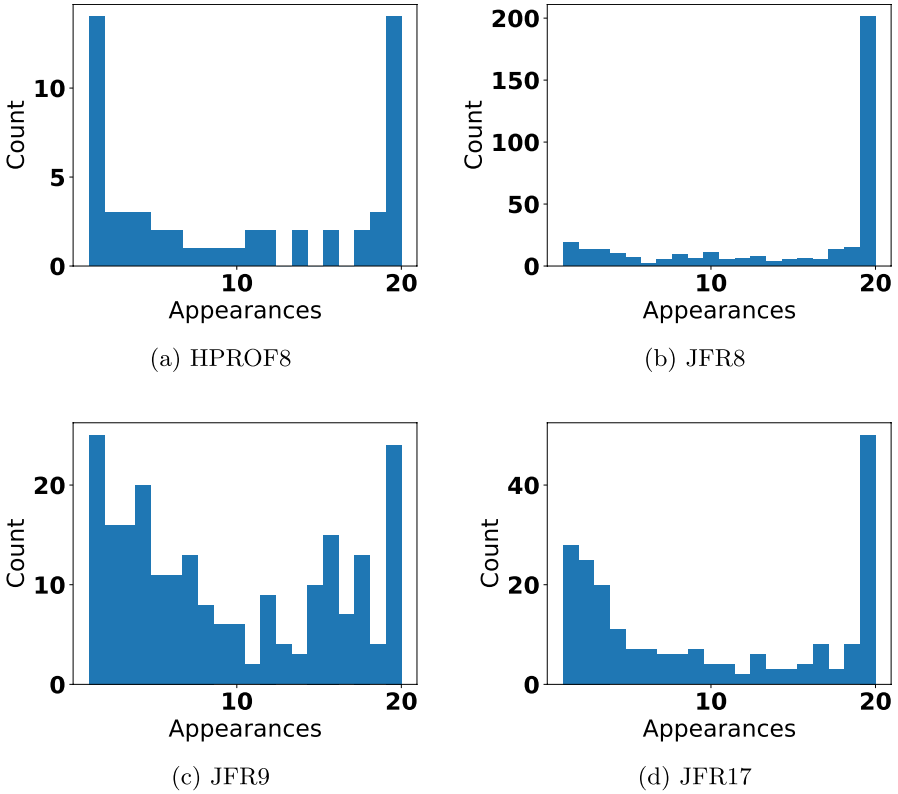
Table 11 Spearman correlation between mean ranks over repeat runs of each profiler when investigating CPU use and memory use. Results for all methods in UNION

Project	Spearman			
	HPROF8	JFR8	JFR9	JFR17
Disruptor	−1.000	–	0.609	–
Gson	−0.786	0.586	0.751	0.596
Jcodec	0.422	0.206	0.427	0.294
Junit4	0.317	0.213	0.620	0.832
OpenNLP	0.539	–	0.679	0.757
Perwendel Spark	0.223	0.600	0.357	–

CPU and memory are in Fig. 9 for Gson and Fig. 10 for JCodec. In some cases, the method ranks for these two properties are similar, with a strong positive correlation, i.e. the same methods that consume most CPU time and consume most memory. In others cases the ranks for CPU and memory are inconsistent; from zero correlation, to strong negative correlation, i.e, the methods consuming most CPU time consumed the least memory, and vice versa. It is clear that there is no simple relationship between the two and profiling for both properties separately is valuable. Further investigation of the relationship between these and other properties would be an interesting direction for future research.

5 Discussion

Differences between each ranking of *hot* methods produced by running the profilers is partially explained by random variations or 'noise' during profiling. This is the only source of variation for one profiler/JDK combination; changing these also naturally leads to a different set of *hot* methods. Thus the same method may not be found the same number of times in every profiling run. The impact of this is that different *hot* methods may be identified; whether these are used by a human developer or for targeting within GI the result will potentially be the missing of improvable parts of the code. Fortunately, for one profiler and JDK combination, the results are



(a) HPROF8 (b) JFR8

(c) JFR9 (d) JFR17

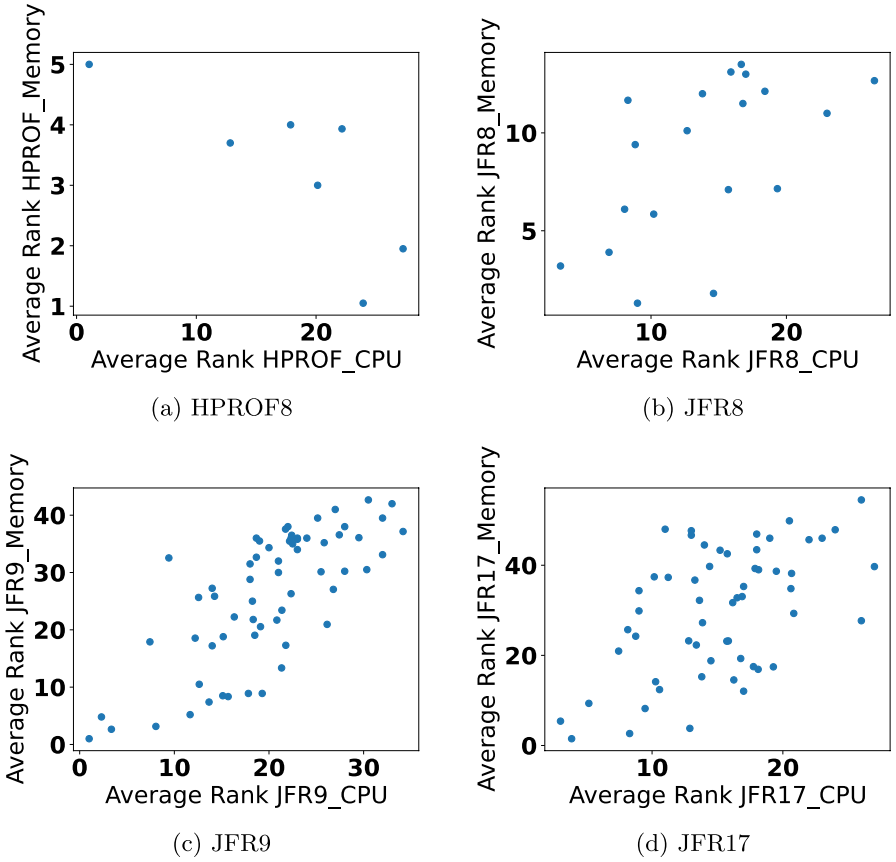
Fig. 8 Distribution of *hot* method appearances across repeat runs of the memory profilers for JUnit4, i.e., how often we see methods appearing only once, twice, up to 20 times

reasonably consistent, and the hottest methods do indeed appear across all repeat runs. Where the profiler and JDK change, then the results are different enough to always require rerunning the profiling.

One main trend seen in results was consistency within HPROF and JFR *hot* method lists. HPROF had greater variation in *hot* method lists compared to JFR. This was first seen in the WRBO within repeat runs of the same project with each profiler. This effect propagated to the WRBO of *hot* method lists between HPROF and JFR.

These clear differences between the two profilers can only partially be attributed to random noise. A large part of the difference in results comes from the two different methodologies the profilers use (Sect. 2).

The method HPROF uses is to check the top of the call stack and increment the count for whatever method is there. If HPROF takes 10 samples and a Java language function is at the top of the call stack each time HPROF will return no *hot* methods. This may be useful if HPROF is used as a general performance profiler. A software engineer could remove a particularly slow function from their program.



(a) HPROF8 (b) JFR8

(c) JFR9 (d) JFR17

Fig. 9 Mean ranks of *hot* methods for GSON for CPU and memory with each profiler / JDK combination, taken for UNION over 20 repeat runs

In the context of GI, no edits can be made to Java API functions so these results are not useful unless the calling function can be traced in the stack, which is not always true for HPROF. This links back to how profiler accuracy is judged: in the practical context of how results will be used.

With JFR the call stack is traversed until a function relating to the main program is found. This eliminates all Java API or other unrelated functions from results. If JFR takes 10 samples, the count of 10 *hot* methods will be incremented. This approach removes a lot of noise from function run times.

As an example looking at how JFR reduces noise, a function 'main' could have a loop that calls a Java internal function 'vector'. HPROF will only see the 'main' function when 'vector' is not running. Alternatively, JFR will see the function regardless of whether 'vector' is running or 'main' is running. From this, it can be seen that JFR is likely to return a higher sample count for the 'main' function. Further, the longer 'vector' runs, the less samples HPROF takes and the shorter 'vector' runs the more samples

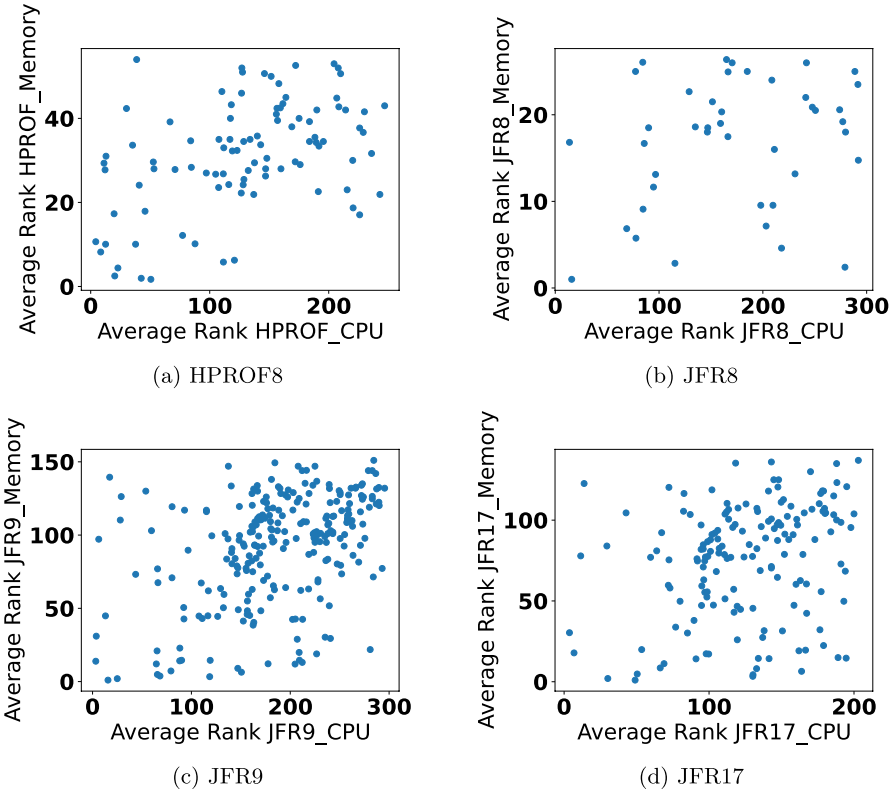


Fig. 10 Mean ranks of *hot* methods for JCodec for CPU and memory with each profiler / JDK combination, taken for UNION over 20 repeat runs

HPROF takes of 'main'. JFR will always see 'main' the number of times it samples the call stack when 'main' is running.

The above example explains how HPROF does not identify functions by how long they run for but instead how long sections of code in which no other functions are called run for. It could be considered that this is a better result than JFR, that records a function on top of the call stack even when an internal function is running on top of it. Although, manipulating how a function calls internal classes may still improve it's run time. Therefore, JFR may be more appropriate for Gin (and other GI tools) than HPROF. The more consistent results of JFR further motivate its use for both CPU and memory profiling.

6 Threats to validity

Our results are heavily dependent on how representative the projects selected for the study are. Five of the six formed part of a previous systematic study in GI search spaces by Petke et al. (2023), where they were chosen following a systematic process using criteria including popularity, scale, and test suite size. The sixth, Perwendel Spark, was chosen arbitrarily by Watkinson and Brownlee (2023) from the set of non-trivial Java projects available on GitHub as part of the original study migrating Gin's profiler from HPROF to JFR. The projects studied cover a variety of application areas and we have no reason to suspect they are outliers.

The results are limited to the specific CPU and OS combination used for our experimental platform. We focused on two specific profilers and three JDK implementations. In part this limitation was imposed by the need to use profilers that we could interrogate programatically. It would certainly be interesting to extend the study to include other profilers but nevertheless the results as they stand are a useful place to start in the exploration of this topic.

Results in this paper may be impacted by the sample rate of each profiler. Both HPROF and JFR use an added 'randomness' to profiling frequency. A 10 ms profiling interval means 10 ms + t where t is any number between −10 and 10. This removes errors from functions that may do the same thing every 10 ms. This would be a disadvantage for other uses of the profiler in which a user may see the same call stack every sample. Further, HPROF uses yield points (Mytkowicz et al. 2010). Yield points are time when it is safe to run a garbage collector, they are placed by the compiler. Although, the compiler may omit yield points if no memory is dynamically allocated as an optimisation which can skew profiling results, by causing some samples to be missed. Furthermore, the filtering of methods to those in the target projects noted in Sect. 2, while necessary in the context of GI, omits system and library calls that will be of interest from the point of view of profiling outside the context of GI.

7 Related work

7.1 Profilers in a GI framework

GI works by applying transformations (mutations) to existing code. The space of transformations is explored to discover code variants with the same or "close enough" functionality as the original code that improve a target property such as runtime or energy consumption. The search space is often sparse and very large i.e., very few functioning variants of the code, very far apart, making it difficult to find code that both retains functionality and performs better than the original. Thus one focus of research has been to determine ways to make this space more amenable to search. This can include new, smarter, mutation operators (e.g., (Harrand et al. 2019; Brownlee et al. 2020)).

Alternatively, reducing the search space by "searching in the right place" (Ahmad et al. 2022) involves profiling the target application in order to identify *hot* methods. As such, in GI the targeted applications are often profiled.[3] For this purpose it is crucial that a profiler identifies functions that impact heavily on the target property. If, say, we are aiming to reduce run time, and waiting or sleeping threads are profiled, the GI search may target a function that waits a long time for other tasks to be done but does not actually complete much work itself. Furthermore, if the selection of *hot* methods varies dependent on the profiler or, worse, from one run of the profiler to the next, the GI search space is likely to change. In this context, further investigation of profiling is an important topic for the GI research community. Furthermore, given their original design purpose to identify bottlenecks in code for human developers, deeper exploration of profiler performance is of more general interest to the software development community.

However, existing GI frameworks rarely offer support for profiling; for example, PyGGI and Magpie (Blot and Petke 2022) do not provide support in their current versions. Instead, profiling appears to be often done by the development team on an ad-hoc basis and with a variety of tools due to the targeted applications and objectives. For example, Haraldsson et al. (2017) profiled code by counting the lines-of-code; Langdon et al. (2017) used nVidia's CUDA performance profiler; Petke et al. (2018) employed callgrind15 and gprof16 to profile; Petke (2022); Langdon and Alexander (2023); Schulte et al. (2014) used the 'perf' tool to count CPU instructions, and Bokhari et al. (2018) used Corbertura 2.1.1.

The Java GI toolbox Gin (White 2017; Brownlee et al. 2019), which forms the focus of the present study, was developed with the aim to facilitate GI research. Gin is open-source, meaning researchers can build upon the existing code, and as far as possible makes use of multiple existing tools such as EvoSuite (Fraser and Arcuri 2011) for automated test creation. Gin also includes a profiling tool to identify *hot* methods for both CPU and memory use in a target software project. Recently, we (Watkinson and Brownlee 2023) updated the profiler from HPROF (Oracle 2011) to Java Flight Recorder (JFR) (GitHub 2020) to allow Gin to migrate from Java 8 to the latest Long Term Support release of Java, version 17. Comparison was given of the outputs of both profilers for CPU use in a toy program and an open source software project. The present work extends and consolidates that experimentation.

7.2 Comparison and analysis of profilers

To our knowledge there is no work comparing profilers specifically in the context of GI. However, there have been some experimental studies comparing profilers more generally.

Analysing Gprof (a profiler for GCC), Varley (1993) published research critiquing the profiler. Their research gives insight into the ways a profiler might be judged. The paper dives into the Gprof methodology to explain its downfall and

[3] A rare exception is the optimisation of straight-line Assembly code by Kuepper et al. (2022).

presents the execution of Gprof on an example program. Finally, the paper presents the positives and negatives of the profiling tool in relation to the expected profiling results. This almost acts as a unit test style analysis. The expected profiling result is manually made and compared to the true profiling result to identify inconsistencies which are stated as profiling inaccuracies.

In their proposal of a sample-based call stack style profiler, Froyd et al. (2004) sets out a number of experiments to compare profilers. They run a program and profile it with multiple tools then compare the time taken. They focus solely on the overhead added by certain profilers.

Similarly to our previous paper, Patel and Rajwat (2013) offer a number of profiler alternatives and compare each of them to address their limitations and advantages. Further, they go on to classify many types of embedded profiling methods. Unlike other work they do not run any experiments, and simply review the architecture and methodology of each profiler to assess accuracy and overhead.

Mytkowicz et al. (2010) comes closest to the work in this paper. They aimed to evaluate the accuracy of Java profilers, comparing four Java profilers available at the time (including HPROF) and found considerable differences in the *hot* methods identified by each. They acknowledged that there may not be a necessarily 'correct' profile, but that an 'actionable' profile could be deemed as accurate. Actionable means if *hot* methods identified by the profile are optimised, the whole program could see a large improvement. This more practical approach to accuracy aligns with the work in this paper; a profiler is needed for a specific purpose and so should be judged in terms of this purpose.

7.3 Paper novelty

The present paper extends our original assessment of profiler suitability (Watkinson and Brownlee 2023), where we used both qualitative and quantitative comparisons. The qualitative analysis, whereby JFR was chosen as a replacement for HPROF, followed closely work by Patel and Rajwat (2013); each profiler's architecture, design and methodology was analysed and used to select a fitting profiler for the context. As opposed to previous work, the present paper focuses on a quantitative comparison of profilers that fall within the same class, all the candidates being sample-based call-stack profilers. We also investigate memory profiling, for which we could not find any previous examples, and test case subset selection for a GI context.

8 Conclusions

Profiling to identify target areas for improvement is an important part of the typical GI pipeline and software development in general. This study has contributed an experimental comparison of four profiler/JDK combinations.

We are able to draw several conclusions from our results. For Research Question 1 (RQ1), we have shown that, while there is certainly noise in the results for a single profiler run, each profiler in the study consistently identifies the top-ranked *hot*

methods over the course of 20 repeats. In some cases fewer repeats may be possible, and we suggest 5 repeats. For RQ2, we have shown that moving from one CPU profiler and JDK to another produces very different results and certainly makes it necessary to repeat the profiling stage for the new context. This is also important for those maintaining software: migrating a program to a new Java version will require a rerun of profiling. For RQ3, we show that, while it is possible to determine a subset of the test suite during the profiling process, the identified tests vary considerably, and a far more promising direction is to employ Regression Test Selection techniques such as (Yoo and Harman 2012; Guizzo et al. 2021). For RQ4, we have shown that memory profiling does indeed follow a similar general pattern to CPU profiling. There is strong consistency within the top-ranked *hot* methods across repeat runs for one profiler/JDK combination, but some large differences when exchanging one profiler/JDK for another.

There remains considerable work to be done in this area. Beyond the obvious directions of further experiments with additional target applications, different profilers, different JDKs, and hardware architectures to confirm our results, it would certainly be interesting to compare profilers that target other properties, as well as measuring the impact on the GI search space of any variations that are observed. There is an interesting broader question about what happens at the long tail of seldom used methods, and deeper understanding of the causes of the noise we observed in our results. A standard test suite for the study of profilers would also be an interesting and useful addition for Java and other platforms.

Acknowledgements The authors would like to thank the anonymous reviewers for their constructive suggestions, and Jay Brownlee for guidance on statistical analysis.

Author contributions M.W and A.B wrote the main manuscript text. M.W. completed the experimental study and implementation; A.B. completed the statistical analysis and generated the plots. All authors reviewed the manuscript.

Data availability The full set of results and visualisations from our experiments can be found at https://github.com/MylesWatkinson/replication_package and in a permanent artefact at (Brownlee and Watkinson 2024).

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

Ahmad, H., Cashin, P., Forrest, S., Weimer, W.: Digging into semantics: Where do search-based software repair methods search? In: Rudolph, G., Kononova, A.V., Aguirre, H., Kerschke, P., Ochoa, G., Tušar, T. (eds.) Parallel Problem Solving from Nature - PPSN XVII, pp. 3–18. Springer, Dortmund (2022)

Apache: Introduction to Profiling Java Applications in NetBeans IDE (2020). https://netbeans.apache.org/tutorial/main/kb/docs/java/profiler-intro/

Blot, A., Petke, J.: MAGPIE: Machine Automated General Performance Improvement via Evolution of Software (2022)

Bokhari, M.A., Alexander, B., Wagner, M.: In-vivo and offline optimisation of energy use in the presence of small energy signals: A case study on a popular Android library. In: 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services. MobiQuitous '18, pp. 207–215. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3286978.3287014

Brownlee, A.E.I., Petke, J., Alexander, B., Barr, E.T., Wagner, M., White, D.R.: Gin: genetic improvement research made easy. In: Genetic and Evolutionary Computation Conference, GECCO 2019, pp. 985–993. ACM, Prague, Czechia (2019). https://doi.org/10.1145/3321707.3321841

Brownlee, A.E.I., Petke, J., Rasburn, A.F.: Injecting shortcuts for faster running Java code. In: 2020 IEEE Congress on Evolutionary Computation (CEC). IEEE Press, Glasgow, Scotland (2020). https://doi.org/10.1109/CEC48606.2020.9185708

Brownlee, A.E.I., Watkinson, M.B.: Data and Processing Scripts for the Paper "Comparing Apples and Oranges? Investigating the Consistency of CPU and Memory Profiler Results Across Multiple Java Versions". note = [Online; accessed 7-February-2024]. http://hdl.handle.net/11667/226

Callan, J., Petke, J.: Multi-objective genetic improvement: A case study with evosuite. In: International Symposium on Search Based Software Engineering, pp. 111–117 (2022). Springer

Eclipse Foundation: Eclipse Downloads | The Eclipse Foundation (2019). https://www.eclipse.org/downloads/

EJ-Technologies: Java Profiler - JProfiler (2020). https://www.ej-technologies.com/products/jprofiler/overview.html

Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 416–419 (2011). ACM

Froyd, N., Mellor-Crummey, J., Fowler, R.: A Sample-Driven Call Stack Profiler, (2004). https://scholarship.rice.edu/bitstream/handle/1911/96328/TR04-437.pdf?sequence=1 &isAllowed=y

GitHub: Java Flight Recorder Events. https://bestsolution-at.github.io/jfr-doc/openjdk-17.html. [Online; accessed 12-July-2023] (2020)

Guizzo, G., Petke, J., Sarro, F., Harman, M.: Enhancing genetic improvement of software with regression test selection. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1323–1333 (2021). IEEE

Haraldsson, S.O., Woodward, J.R., Brownlee, A.E.I., Smith, A.V., Gudnason, V.: Genetic improvement of runtime and its fitness landscape in a bioinformatics application. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. GECCO '17, pp. 1521–1528. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3067695.3082526

Harrand, N., Allier, S., Rodriguez-Cancio, M., Monperrus, M., Baudry, B.: A journey among Java neutral program variants. Genet. Program Evolvable Mach. **20**(4), 531–580 (2019)

Kuepper, J., Erbsen, A., Gross, J., Conoly, O., Sun, C., Tian, S., Wu, D., Chlipala, A., Chuengsatiansup, C., Genkin, D., Wagner, M., Yarom, Y.: CryptOpt: Verified Compilation with Random Program Search for Cryptographic Primitives (2022). [Online; accessed 9-January-2023]

Langdon, W.B., Alexander, B.J.: Genetic Improvement of OLC and H3 with Magpie. 2023 IEEE/ACM International Workshop on Genetic Improvement (GI) (2023) https://doi.org/10.1109/gi59320.2023.00011

Langdon, W.B., Harman, M.: Optimizing existing software with genetic programming. IEEE Trans. Evol. Comput. **19**(1), 118–135 (2015). https://doi.org/10.1109/TEVC.2013.2281544

Langdon, W.B., Lam, B.Y.H., Modat, M., Petke, J., Harman, M.: Genetic improvement of GPU software. Genet. Program Evolvable Mach. **18**(1), 5–44 (2017). https://doi.org/10.1007/s10710-016-9273-9

Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Trans. Software Eng. **38**, 54–72 (2012). https://doi.org/10.1109/TSE.2011.104

Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Evaluating the accuracy of Java profilers. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '10, pp. 187–197. Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1806596.1806618

Oracle Corporation: Oracle JDK Migration Guide. https://docs.oracle.com/en/java/javase/18/migrate/index.html. [Online; accessed 9-January-2023] (2022)

Oracle: HPROF: A Heap/CPU Profiling Tool. https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html. [Online; accessed 12-July-2023] (2011)

Oracle: JDK Mission Control (2018). https://www.oracle.com/java/technologies/jdk-mission-control.html

Oracle: JFR Runtime Guide. https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170. [Online; accessed 12-July-2023] (2014)

Oracle: Using JConsole - Java SE Monitoring and Management Guide (2023). https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html

Patel, R., Rajwat, A.: A survey of embedded software profiling methodologies. International Journal of Embedded Systems and Applications **1** (2013) https://doi.org/10.5121/ijesa.2011.1203

Petke, J., Brownlee, A.E.I.: Software improvement with Gin: A case study. In: Search-Based Software Engineering: 11th International Symposium, SSBSE 2019, Tallinn, Estonia, August 31 - September 1, 2019, Proceedings, pp. 183–189. Springer, Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-030-27455-9_14

Petke, J.: Using genetic improvement to optimise optimisation algorithm implementations. (2022). https://api.semanticscholar.org/CorpusID:251435594

Petke, J., Haraldsson, S.O., Harman, M., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Trans. Evol. Comput. **22**(3), 415–432 (2017). https://doi.org/10.1109/TEVC.2017.2693219

Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Specialising software for different downstream applications using genetic improvement and code transplantation. IEEE Trans. Software Eng. **44**(6), 574–594 (2018). https://doi.org/10.1109/TSE.2017.2702606

Petke, J., Alexander, B., Barr, E.T., Brownlee, A.E., Wagner, M., White, D.R.: Program transformation landscapes for automated program modification using Gin. Empir. Softw. Eng. **28**(4), 1–41 (2023)

Raikar, K.: How to objectively compare two ranked lists in Python. Towareds Data Science (2023). https://towardsdatascience.com/how-to-objectively-compare-two-ranked-lists-in-python-b3d74e236f6a

Sarica, A., Quattrone, A., Quattrone, A.: Introducing the rank-biased overlap as similarity measure for feature importance in explainable machine learning: A case study on Parkinson's Disease. In: Mahmud, M., He, J., Vassanelli, S., Zundert, A., Zhong, N. (eds.) Brain Inform., pp. 129–139. Springer, Cham (2022)

Schulte, E., Dorn, J., Harding, S., Forrest, S., Weimer, W.: Post-compiler software optimization for reducing energy. SIGARCH Comput. Archit. News **42**(1), 639–652 (2014). https://doi.org/10.1145/2654822.2541980

Sedlacek, J., Hurka, T.: VisualVM: Home (2022). https://visualvm.github.io/

Varley, D.A.: Practical experience of the limitations of gprof. Software: Practice and Experience 23(4), 461–463 (1993) https://doi.org/10.1002/spe.4380230407

Watkinson, M., Brownlee, A.: Updating Gin's profiler for current Java. In: The 12th International Workshop on Genetic Improvement, at the International Conference on Software Engineering, Melbourne, Australia, May 20 2023 (2023). ACM

Webber, W., Moffat, A., Zobel, J.: A similarity measure for indefinite rankings. ACM Trans. Inform. Syst. (TOIS) **28**(4), 1–38 (2010)

White, D.R.: GI in no time. In: Genetic and Evolutionary Computation Conference, July 15-19, 2017, Companion Material Proceedings, pp. 1549–1550. ACM, Berlin, Germany (2017). https://doi.org/10.1145/3067695.3082515

Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Testing Verif. Reliability **22**(2), 67–120 (2012)