Updating Gin's profiler for current Java

(removed for double-blind review)

***Abstract*—Genetic improvement is a young and growing field. With much research still to be done, a number of tools to support the research community have emerged, with Gin being one such tool targeted at GI for Java.**

One core component of Gin is the profiler, which is used to identify 'hot' methods in target applications: methods where the CPU spends most time and so may offer the most fertile sections of code for improvements to run time. Gin's profiler is HPROF, which was included with JDKs up to version 8. HPROF is no longer supported and so needs replaced if Gin is to support later versions of Java. Furthermore, little investigation has been made within the GI community comparing different profiling approaches.

With this paper and its associated & accepted pull request, we replace Gin's CPU profiler to allow Gin to be applied to current Java code.

I. Introduction

Genetic improvement (GI) is a field of software engineering that aims to automatically improve code. Though GI first emerged in 1995, research in the topic began rising from 2008 and had continue to grow in recent years [1]. Genetic improvement can target functional improvements of code, such as bug fixes (e.g., GenProg [2]), or non-functional improvements, such as run time (e.g., [3]).

To stimulate the growth of research in GI, the Gin tool [4], [5] was created, with the aim to be a compact and simple tool for Genetic Improvement 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 [6] for automated test creation and HPROF [7] for estimating the runtime of methods. Gin removes the need for reinventing the wheel. Rather than spend time writing the foundations for genetic improvement, researchers can immediately begin building upon the software.

GI works by applying transformations (mutations) to existing code, creating a search space of potential newzcode. Typically, this search space is then explored to discover mutant code that has the same or "close enough" functionality as the original code and makes improvement in a target property such as runtime or energy consumption. The search space is often sparse and very large, 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., [8], [9]).

In GI, the targeted applications are often profiled, as this reduces the search space by identifying worthwhile targets.[1] Interestingly, however, existing GI frameworks rarely offer support for profiling; for example, PyGGI and PyGGI 2.0 do not provide support in their current versions. Instead, profiling appears to be often done by the respective team on an ad-hoc basis and with a variety of tools due to the targeted applications and objectives. For example, [11] profiles code by counting the lines-of-code; [12] uses nVidia's CUDA performance profiler; [13] employs callgrind15 and gprof16 to profile; and [14] uses Corbertura 2.1.1. Hence, keeping Gin's profiler up-to-date has the potential to enable future research more easily than other GI tools that do not provide built-in support.

Currently, Gin is implemented in the Java language and solely used for the mutation of Java code. Although it is written in, and targets, Java 8, this is now dated compared to Java's latest version, 18 [15]. Due to changes in the Java Development Kit (JDK) some parts of Gin no longer work in newer versions. The two major changes impacting Gin are the removal of HPROF, a profiling tool used to analyse methods to identify those used often, and greatly tightened restrictions on the use of reflection [15].

The goal of this work is to tackle the first of these changes: upgrading Gin's profiler to support recent Java versions up to 18. We have two research questions in this work: (RQ1) which profiling tool is most suitable to replace HPROF in Gin? and (RQ2) is the replacement profiler likely to find different hot methods to those identified by HPROF?

To answer RQ1, multiple alternative CPU profiling tools will be compared rigorously using chosen criteria to select a candidate. This analysis leads us to choose Java Flight Recorder (JFR) to replace HPROF as Gin's profiler. While many GI approaches use profiling in some capacity there has been little experimental testing or validation of the profiling approaches used. Thus, to answer RQ2, we also perform an experimental comparison of HPROF and JFR on a toy program and an open source application to test the consistency by which both identify hot methods.

The rest of the paper is structured as follows. Section II describes the problem to be tackled in terms of the requirements for Gin's profiling tool and compares the various candidate profilers against these requirements. Section III describes our experimental study and Section IV discusses the results. In Section V we give our conclusions and suggestions for future work.

II. Requirements and Profiler Tools

There are many profiling tools available for Java. The following paragraphs identify useful properties to compare a number of tools and examine the adequacy of each tool in Gin's context.

[1] A rare exception is the optimisation of straight-line Assembly code in [10].

In the current version of Gin, HPROF is used for CPU profiling of the Java Virtual Machine (JVM). At chosen intervals (default 10ms) it takes a sample of the call stack. The stack is then descended until a method in the target application is found (rather than a test or Java/external library API method). Gin writes this data to a CSV file identifying "hot methods" or those that should be focused on in mutation [5].

Gin is a toolbox, it is simple and compact. Any external programs connected to it should not require advanced knowledge to use. Moreover, these tools should not need any input from users of Gin making automation a necessity. Automation also means being run from the command line by Gin or from an API inside Gin. Interactivity is not needed and should be avoided to reduce overhead. Alongside automation, integration is important. New features should plug directly into code that already exists without changes having to be made. Depending on the quantity of tests, initial runs may take up to hours on large programs meaning a profiler cannot add too much time to this process. Lastly, the profiling tool chosen needs to be free due to Gin's nature as a research implement.

Java profiling and diagnostic tools can be grouped into two realms, third party, and those incorporated in the Java package. Third party programs, either free or subscription based, are Jprofiler and NetBeans. Built-in diagnostic tools are JConsole, Java flight recorder (JFR), Java Mission Control (JMC) and VisualVM. We now briefly summarise each.

JProfiler:

A heap and CPU profiler for Java that shows method calls, run time and memory usage. Users can choose to record or visualise data on demand. Its focus is on low overhead, and it has the option to display how selected JProfiler settings will affect program performance.

NetBeans:

This third party Java profiler by Apache can be used to analyse CPU usage and memory allocation. It is interactive nature and it cannot be operated from the command line.

JConsole:

This graphical tool is used to visualise CPU and memory usage in a Java program. When operated from the command line, JConsole monitors a Java application and outputs this data in a graph format. To save this data as a CSV file, user input is needed. JConsole has significant overhead as it measures all activity in the JVM and is recommended to be used remotely to save resources.[2]

VisualVM:

Similar to NetBeans and JProfiler, VisualVM provides a graphical interface to visualise profiling information. It is also similarly interactive and cannot be operated from the command line.

Java Flight Recorder (JFR):

JFR records information on events that occur in the JVM. The event Gin is most concerned with is function calls. Due to only taking information on events, along with other design specifications, JFR has little overhead. Contents of a flight recording are output to a .jfr file after a recording ends.

[2]For example, see the discussion at https://stackoverflow.com/questions/6577758

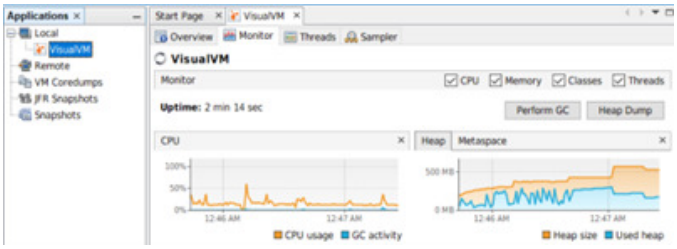

Fig. 1. VisualVM interface; VisualVM is focused towards developers who need to visualise data while working on programs, creating graphs adds to the overhead caused by using such an interactive tool.

Java Mission Control (JMC):

JMC was initially created to plug into JFR to visualise data it collects; it now exists as a standalone tool. However, it still acts to visualise data for developers, making it interactive in nature and inoperable from the command line or batch file.

Firstly, JProfiler can be removed from the candidates. JProfiler can only be used with a subscription making it unsuitable for Gin. Next, for ease of integration, other third-party programs VisualVM and NetBeans can be removed from consideration. Any software external to Java adds more dependencies and is less preferable than a Java-internal solution. Now that built-in Java tools are left, comparison between the two can be made in terms of: performance, automation and integration. Performance of a diagnostic tool is determined by the number of input and output (I/O) operations. Similarly to HPROF, JFR only takes information on events that occur in the JVM. This means it avoids heap profiling which may be costly. JFR also has a specialised data flow to reduce disk I/O operations. JFR collects data and keeps it in a buffer and when this buffer is full, flushes data to the disk. This makes JFR extremely fast even though it collects large amounts of data on all events that run in a Java program. JConsole, VisualVM and JMC all use a graphical interface to visualise data. The I/O operations needed to transfer data then create graphs make these tools slow in comparison to JFR. Though visual depictions of data make understanding heap usage easier for developers, this feature is outside the scope of Gin, which only requires data on CPU usage. Keeping this data in the simplest form possible will reduce with overhead. In terms of integration and automation all the programs considered can be activated from the command line and all come packaged in the newest version of the JDK. This removes the need for any added dependencies and keeps Gin as compact as possible. Following the above points, JFR appears to be the best choice to replace HPROF: it can be used from the command line, has negligible overhead and is still supported in current Java versions.

Integrating JFR with Gin is relatively simple. To run JFR simply requires changing the flags to the Java command that runs the profiler. The code to use JFR is similar to the pre-existing HPROF code in Gin's "gin.util.Profiler" class. These

TABLE I
BASELINE PROFILING FUNCTIONS; THE NAME OF EACH FUNCTION ALONG WITH THE NUMBER OF PRIMES IT CALCULATES AND THE TIME IT TAKES TO RUN.

Function name	Primes found	Time taken (ms)
P5()	5000	24
P10()	10 000	75
P15()	15 000	153
P20()	20 000	266
P25()	25 000	412
P30()	30 000	588

similarities mean that the profiler can use JFR or HPROF simply by changing the run command arguments, from:

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

to

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

JFR also reports the data differently to HPROF; so the parsing step within Gin's Profiler class needed amended to use the built-in Java RecordingFile class.

III. EXPERIMENT AND RESULTS

Next, we compare both HPROF and JFR in experiments. They both profile the JVM to produce statistics on what occurs inside it, although, they operate differently.

Note that the ground truth is unknown, and that our primary goal is to replace HPROF with something that provides qualitatively "similar" results.

Experimental Setup Our experimentation compares the HPROF and JFR tools. HPROF is freely available with Java Development Kit (JDK) versions 8 and below, whereas JFR is available with versions 9 and up; consequently a key difference is that runs were carried out using JDK 8 for HPROF and JDK 9 for JFR. All profiling is done inside of Gin using its profiling utility. The projects being profiled are a synthetic baseline program, the Gin "Simple example" that comes packaged in Gin, and Spark, a Java web framework found on GitHub[3]. Results were taken in the form of CSV files that Gin outputs directly.

Baseline Method The basic outline of the experiment was to profile a project with both JFR and HPROF to determine if they both found the same methods as hot methods and found these methods running the same number of times. First, a baseline test was run. This initial test profiled a program calculating prime numbers. Ten functions were included labeled p5, p10, p15, p20, p25 and p30. Each px function calculated the first x * 1000 primes. The use of a prime calculation function allowed for clear view of how each tool profiled the JVM, with a simple repeat of the processing to make more CPU-intensive methods.

[3]Apache Spark: https://github.com/apache/spark

TABLE II
BASELINE PROFILING FUNCTIONS; RESULTS.

Function	HPROF samples	JFR samples
P5()	1.25	1.75
P10()	5.75	5
P15()	12.5	8.5
P20()	23	14.75
P25()	34	22.75
P30()	54.5	32.25

JFR and HPROF both have profiling intervals of 10ms: they both take a snapshot of the call stack every 10ms. Therefore, it can be predetermined what the expected number of function calls is for each method by dividing the time taken by 10. The functions were implanted into Gin's "maven-simple" example program which is used by five unit tests. These tests run by the program multiple times meaning each function was run 10 times.

Baseline Results Table II shows the average number of profiles of each function. As expected, the functions calculating more primes appear in the profiling more often.

In the longer-running functions, it can be seen that HPROF consistently finds more samples of each function, meaning that HPROF finds each function at the top of the call stack more often. Further, some important data not in the table is the methods found most by JFR and HPROF. JFR finds 'Java.util.Arrays.copyOf' almost three times more than any other function and, similarly, HPROF finds 'Java.io.fileInputStream.readBytes' three times more than any other function. These two functions are present when HPROF and JFR sample threads but are not found by both profilers. This is due to the fact that JFR samples events that occur in the JVM, whereas, HPROF samples all active threads. Events refer to function calls, including those by a running program and by internal Java classes. Therefore, JFR finds a huge number of Java.util and Java.lang functions that are almost always running when a call is made to operate on a Java language data structure. HPROF can sample I/O operations occurring in the JVM, along with sleeping, resting or stopped threads, that JFR cannot see. These are not positives or negatives, simply differences in the way both profilers work. For Gin, only consistent identification of hot methods is needed for both tools to be deemed candidates for a test profiler.

Test Methodology Both profilers are adequate for identifying hot methods in a controlled test, a larger project has been selected for profiling. The profiling target is Spark, an open source Java project found on GitHub. Spark comes packaged with a number of test cases that run for 30 minutes in Java 8 and 22 minutes in Java 9. Gin's profiling identified hot methods in Spark, running once with HPROF and once with JFR. This process was repeated 30 times to output 30 sets of hot methods files for each profiler. To run the profiler 30 times a bash function was used. This allows for a separate JVM to be made for each run. This avoids potential issues with the

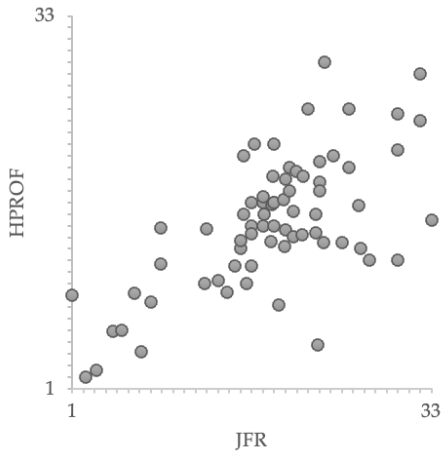

Fig. 2. Average ranks of hot methods: HPROF and JFR. Each of the 70 points represents one method, and it denotes the average ranks returned by either profiler. The averages are the results of 30 independent runs of each profiler.

JVM not closing properly after the profiler was run. It also forced Gin tests to be run before every instance to assure that the profiler would run as intended.

JFR and HPROF each output respective files that are used to extract results. HPROF outputs result to a .csv file that can be easily read and translated. Dissimilarly, JFR outputs to a .jfr file with a binary format that requires specific tools to view. Programmatically, this file is broken into RecordedEvents, a data structure from the JFR library, by first reading the .jfr file to a JFR RecordingFile. 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, Java Mission Control can easily visualise JFR files to debug.

Test Results Over 30 tests JFR identified 107 hot methods, these methods were used as true hot methods. HPROF found 70 of these methods in total and all of the top 10 most profiled methods.

As seen in Figure 2, the average ranks of the hot methods that both profilers find appear to be positively correlated, indicating that both profilers find comparable sets of hot methods. The Spearman correlation coefficient for the entire set of 70 functions is just 0.57; however, this result appears to be affected significantly by rarely sampled hot methods, as the Spearman correlation coefficient for the top 30 JFR methods is 0.80.

IV. Discussion

The difference between each data set is partially explained by random variations or 'noise' during profiling. Due to inconsistency of runtime over several samples, the same method may not be found the same number of times in every test. The change of the Java version between the tests is also likely to have impacted the sampling. Presumably, as Java versions

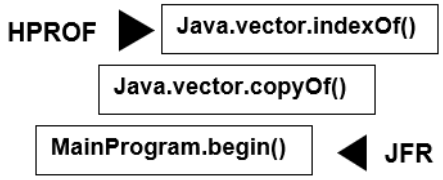

Fig. 3. Example of a call stack: three methods in a call stack showing the bottom of the stack, found by JFR and the top, found by HPROF.

increase, the efficiency of both the java compiler and JVM improve allowing for code to run faster, therefore, possibly changing profiling results.

As number of samples decreases, HPROF identified methods reduce in rank at a slower rate than JFR methods. This is due to the difference in number of methods sampled by each profiler. On average, JFR found 26.2 hot methods, whereas HPROF found only 21.6. This means, the ranking of hot methods was distributed differently depending on the profiler. The method of analysing profiling data accounts for this difference.

When analysing HPROF data, the top of the call stack is analysed and recorded. Methods found at the top of the call stack most often are shortlisted and those that belong to the main program are returned as hot methods; see Figure 3 for a sketch of a situation. Alternatively, JFR traverses the call stack until it discovers a method that is part of the main program, then, this method's number of samples is incremented. This means, a method that is part of the main program is found in every JFR 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 JFR consistently finds more hot methods and finds them more times in samples.

The implications of the two different methods do not impact the final profiling result. If a method is called multiple times and this method calls many different Java language functions, it will never be profiled by JFR. Although, this method should still be deemed "hot" as manipulating the way it calls internal classes may improve its runtime.

Considerations for Integration of JFR Outside of the differences in how each profiler operates, they also interact differently with the system that they are executed on.

When using HPROF, results for an individual profiling run are ouput to a textfile. These textfiles may be multiple kilobytes in size, yet are often below 30KB. In contrast, JFR output files are in a binary format, this means, the user is unable to view them without writing code to analyse a file and the file sizes may be in the hundreds of KBs. This poses an issue as Gin will store many of these files on a user's system. For larger programs that have minutes or hours of tests, profiling result files could take up Gigabytes of space. One solution is to delete all files after profiling. As these files are processed by Gin itself, it is unlikely that a user will have need for them after profiling. Although, to offer functionality for any use case, a command line option is added to save files after profiling. The default value of this option is to delete

files.

Another command line option is added to alternate between HPROF and JFR. Gin is a research tool and therefore should be backwards compatible with Java versions. This compatibility allows for the use of Gin with other tools and utilities across Java versions. The default option is now set to use JFR.

V. Conclusions

After selecting JFR and testing it against HPROF, it can be seen that the new profiler produces similar but not identical results to the previous tool. For the 'hottest' 30 methods in our target project, there is a positive Spearman correlation of 0.80 between the rankings found by HPROF and JFR. So, although the method for use and interacting with Gin is slightly different, JFR provides results largely consistent with HPROF. Moreover, JFR's workflow can be seamlessly integrated into Gin. One remaining issue remains to upgrade Gin to more recent Java versions: reliance on Java reflection as part of the JUnit testing harnesses which has been more heavily restricted in recent versions. However, this should only break Gin for Java 16 onwards (and then, when using the internal testing code rather than an external JVM), so the improvement to the profiler has brought Gin considerably closer to modern Java.

Our accepted pull request can be found at https://github.com/gintool/gin/pull/RemovedForDoubleBlindReview.

References

[1] J. Petke, S. O. Haraldsson, M. Harman, D. R. White, Woodward, and J. R. Woodward, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, 2017.

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, pp. 54–72, 2012.

[3] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, 2015.

[4] D. R. White, "GI in no time," in *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*. ACM, 2017, pp. 1549–1550.

[5] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: genetic improvement research made easy," in *Genetic and Evolutionary Computation Conference, GECCO 2019*, A. Auger and T. Stützle, Eds. ACM, 2019, pp. 985–993.

[6] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011, pp. 416–419.

[7] Oracle Systems, "Hprof: A heap/cpu profiling tool," https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html, 2020, [Online; accessed 6-February-2019].

[8] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry, "A journey among java neutral program variants," *Genetic Programming and Evolvable Machines*, vol. 20, no. 4, pp. 531–580, 2019.

[9] A. E. Brownlee, J. Petke, and A. F. Rasburn, "Injecting shortcuts for faster running java code," in *2020 IEEE Congress on Evolutionary Computation (CEC)*. IEEE Press, 2020, p. 1–8.

[10] J. Kuepper *et al.*, "CryptOpt: Verified Compilation with Random Program Search for Cryptographic Primitives," 2022.

[11] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, A. V. Smith, and V. Gudnason, "Genetic improvement of runtime and its fitness landscape in a bioinformatics application," in *Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. ACM, 2017, p. 1521–1528.

[12] W. B. Langdon, B. Y. H. Lam, M. Modat, J. Petke, and M. Harman, "Genetic improvement of gpu software," *Genetic Programming and Evolvable Machines*, vol. 18, no. 1, pp. 5–44, 2017.

[13] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Specialising software for different downstream applications using genetic improvement and code transplantation," *IEEE Transactions on Software Engineering*, vol. 44, no. 6, pp. 574–594, 2018.

[14] M. A. Bokhari, B. Alexander, and M. Wagner, "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*, ser. MobiQuitous '18. ACM, 2018, p. 207–215.

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