ThreadRadar: A Thread-Aware Visualization for Debugging Concurrent Java Programs

Oliver Moseler
University of Trier, Germany
moseler@uni-trier.de

Lucas Kreber
University of Trier, Germany
kreberl@uni-trier.de

Stephan Diehl
University of Trier, Germany
diehl@uni-trier.de

ABSTRACT

Due to non-deterministic behavior and thread interleaving of concurrent programs, the debugging of concurrency and performance issues is a rather difficult and often tedious task. In this paper, we present an approach that combines statistical profiling, clustering and visualization to facilitate this task. We implemented our approach in a tool which is integrated as a plugin into a widely used IDE. First, we introduce our approach with details on the profiling and clustering strategy that produce runtime metrics and clusters of threads for source-code artifacts at different levels of abstraction (class and method) and the entire program. Next, we explain the design of our sparkline visualization which represents the clusters in situ, i.e. embedded in the program text next to the related source-code artifact in the source-code editor. More detailed information is available in separate views that also allow the user to interactively configure thread filters. In a demonstration study we illustrate the usefulness of the tool for understanding and fixing performance and concurrency issues. Finally, we report on first formative results from a small-scale user study.

KEYWORDS

debugging, concurrency, performance, Java, thread, visualization

ACM Reference Format:
Oliver Moseler, Lucas Kreber, and Stephan Diehl. 2021. ThreadRadar: A Thread-Aware Visualization for Debugging Concurrent Java Programs. In *The 14th International Symposium on Visual Information Communication and Interaction (VINCI '21), September 6–8, 2021, Potsdam, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3481549.3481566

1 INTRODUCTION

Today, almost every computing device is equipped with multiple CPU cores. Developers can leverage this processor feature by using concurrent programming to improve software performance. Studies show that concurrent programming is becoming more popular across general purpose languages, such as Java [19, 25]. Developing correct concurrent programs is difficult [23]. The non-deterministic behavior and thread interleaving of concurrent programs renders debugging of concurrency and performance issues even more difficult and time consuming compared with single threaded programs.

VINCI '21, September 6–8, 2021, Potsdam, Germany

ACM ISBN 978-1-4503-8647-0/21/09…$15.00
https://doi.org/10.1145/3481549.3481566

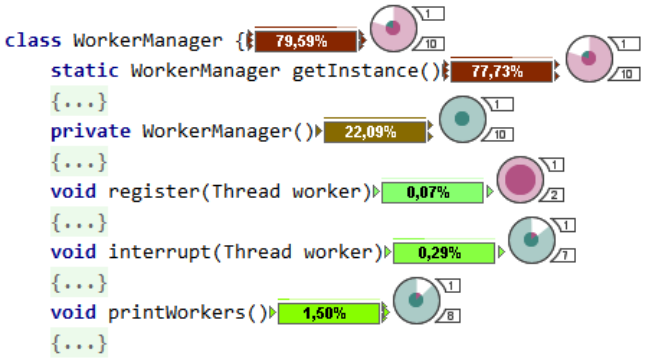


Figure 1: Examples of the thread visualization based on a test-case run of Bug 1 (class `WorkerManager`, Section 5).

Reading and understanding the source code is an integral element of every debugging task. Developers still lack on thread-aware programming tools that facilitate the understanding of concurrent programs [7]. Ideally, these should be part of their daily work environment typically including an IDE.

In this work, we present our approach for visual debugging of performance and concurrency issues. We implemented our approach in the CodeSparks-JPT debugging tool that augments the source code with a thread-aware visualization in the source-code editor of a widely used Java IDE, namely the Intellij IDEA. We use statistical profiling based on stack sampling to compute dynamic performance metrics for the entire program as well as for individual source-code artifacts, such as Java classes and methods. The metric shows the relative runtime consumption of an artifact with respect to the overall profiled run of the Java program. Based on this metric, we developed a visualization showing how runtime consumption is distributed over clusters and types of threads executing the source-code artifacts. The visualization is embedded in the source code in a sparkline [24] manner (Figure 1). In combination with former in-situ visualizations and other tool features it enhances program understanding of concurrent programs. In addition to the in-situ visualizations, more detailed thread information can be retrieved on demand in a detail view or the overview window providing features to further explore and filter the profiling data on the basis of threads and source-code artifacts.

2 APPROACH

The first task during debugging is to reproduce the failing program run. This can be done by creating a dedicated test case running the program under inspection with a fixed input. In many situations such a test case only executes a fraction of the code base. Furthermore, it can be assumed that the developer has a deep understanding of the source code to be debugged and expectations of

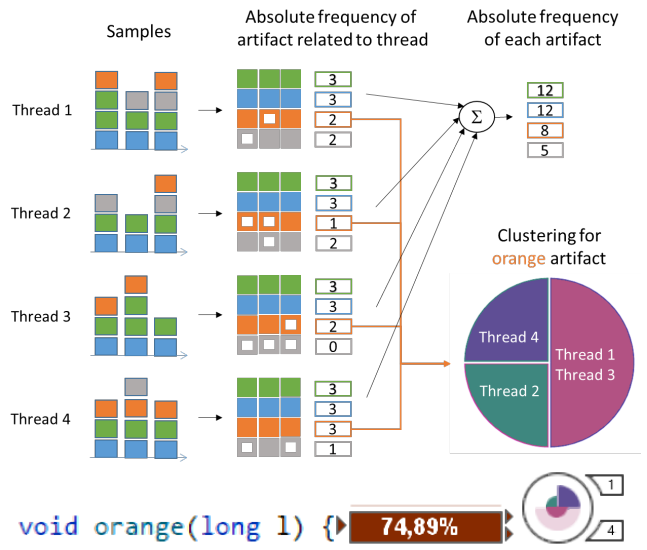


Figure 2: Approach: Profiling, computation of metrics and clustering. The runtime of each artifact *a* is the absolute frequency of *a* divided by the total number of threads sampled, here: 12 = 3 samples * 4 threads.

its dynamic behavior. In concurrent programming, and in particular the debugging of concurrency bugs, source-code artifacts are potentially executed throughout many different threads which might have non-deterministic interleaving. Thus making it very difficult to assess and distinguish to what extent a source-code artifact has been executed by which thread. Consequently, for concurrency debugging, it is hard to find the threads which are related to the bug. Since concurrent programming also employs techniques to coordinate threads, e.g. through synchronization mechanisms, another aspect to characterize how program code is executed by threads is added, for instance waiting on a lock in contrast to actively executing instructions. As illustrated in Fig. 2, our approach addresses these issues by combining a) statistical profiling, b) thread-aware runtime metrics, c) clustering of threads on the basis of these metrics and d) interactive in-situ visualizations presenting the thread information in the source-code editor:

Profiling. We utilize a statistical profiling, in particular a stack sampling to periodically record the stack traces of all threads of a run of an application.

Runtime metrics. From this, we compute a runtime metric for each source-code artifact *a*. To this end, we aggregate all occurrences of *a* in the stack traces of all threads of all samples and put this in relation to the complete sampling. We call this metric the *computational runtime* of *a*. It indicates the relative frequency of the source-code artifact being active in the stack traces of the threads with state *Runnable*. We further disassemble the artifact runtime of *a* into *a*-relative runtimes of of each thread *t*, i.e. the ratio of the runtime that *t* spends executing code in *a* and the total computational runtime of *a*.

Clustering. For each source-code artifact *a*, we cluster its executing threads on the basis of the *a*-relative computational runtime of the threads. Therefore, we utilize the constrained k-means clustering algorithm [26] with $k = 3$ to compute a maximum of three clusters. We further use $\epsilon = max(0.01, 1/(2n))$, where n is the number of threads executing the artifact as a must-link constraint on the metric value and, the absolute difference of two metric values as the similarity measure. Due to this clustering strategy, usually all threads in a cluster have similar metric values. In other words, the metric value of each thread is close to the average metric of the cluster. To enforce the computation of clusters with threads with low, intermediate and high metric value, we initialise the clustering algorithm with three specially selected centroids, namely the threads having the minimal, median and maximal metric value, respectively. Our clustering strategy is supposed to support the extraction of outlier threads to automatically reveal those threads having different runtime behavior than others. In the context of concurrency debugging, this might be beneficial to discover the threads directly related to the actual root cause of the bug.

In-Situ Thread Visualization. The thread clustering also serves the purpose to reduce the amount of data to design a scalable visualization of the thread information which we will introduce in Section 3. Furthermore, the restriction to a maximum of three thread clusters not only allows to classify the threads in three runtime categories but also fits in smoothly with design decisions for the thread visualization.

3 IN-SITU THREAD VISUALIZATION

The conceptual design of the thread visualization was developed on the basis of Java programs and performance data. In the following, we introduce the visual requirements derived from this, lead through the development of the visualization and discuss how it applies to different levels of abstraction.

3.1 Visual requirements

Prior to the conceptual design of the thread visualization, we identified two essential visual requirements. First, the visualization should be placed close to the source-code artifact it is related to, such that the source code remains readable and the visualization itself does not overlay any portion of the code. Second, the visualization should be compact and have a fixed size independent of the amount of profiling data.

Placement. The artifacts we address are Java classes and methods. The most intuitive placement for a visualization for Java classes or -methods is the header of their declaration. According to common Java code formatting guidelines, there is at least one empty line between class or method declarations, and thus, a height of two lines is available for visualizations, namely the line of the header of the declaration itself plus the empty line before the header.

Compactness & Scalability. We choose a radial design, namely a variant of a pie chart, for Java classes and -methods which needs constant space no matter how the underlying data grows.

3.2 The ThreadRadar visualization step by step

We illustrate the development of our radial thread visualization for Java classes and methods by step-wise refinement. We started with

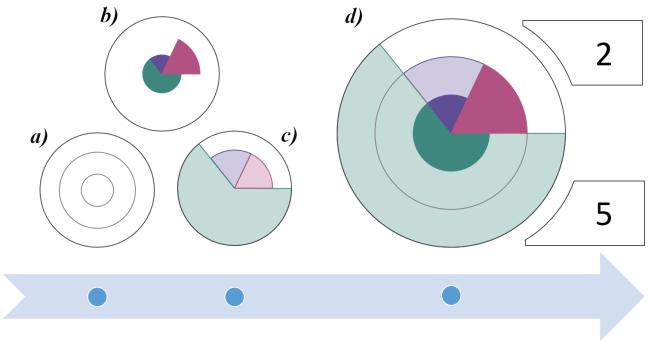


Figure 3: Combining the circular lines indicating the three-valued scale (a) with the pie chart with radii indicating average computational runtimes (b) and the pie chart with weaker colored sectors and radii indicating the sum of the computational runtimes (c) finally yields the ThreadRadar visualization.


a simple pie chart approach. By applying the clustering strategy outlined in Section 2, we map the profiled data to the three sectors. This has also the advantage, that it is feasible to select three clearly distinguishable colors to draw each cluster, especially considering the small size of the visualization. The angle of each sector indicates the percentage of threads in the corresponding cluster with respect to the total number of threads that executed the source-code artifact.


To emphasize threads which have high computational runtime compared to all others, we use the radius of each sector to represent the average metric value of the cluster. Due to the restricted available visualization area, we do not use a continuous scale, but instead a three-valued scale to depict the average metric value.

To facilitate reading of the sector radii, we add circular lines indicating the levels of the three-value scale (Figure 3a). If there are only a few threads with high metric value in a cluster, it would result in a thin (small angled) sector with large radius, which visually sticks out and would more likely attract the developer's attention.

In addition to the average metric values, the visualization also shows the sum of all metric values in a cluster. To this end, the same sectors are used, but with different fill color and different radii (Figure 3c). More precisely, each sector is drawn with the same, but weaker color (same hue, but lower saturation) as in the previous case and the sum of the metric values in the cluster is mapped to the radius of these weaker colored sectors. Since the average metric value will always be lower or equal to the sum of the metric values, the weaker colored sector will be at least as large as the stronger colored saturated one. The following examples show two cases where the clusters have the same sizes, but different distribution of metric values:


Two thread clusters. One (green) has many threads with low metric value each, while the other (red) has few threads with medium metric value each.


Two thread clusters. One (light green) has many threads with low metric value each but these values sum up to a value higher than 66%, while the other (red) has few threads having low metric value each.

Drawing the three components discussed above on top of each other into a single diagram yields the *ThreadRadar* visualization (Figure 3d). To provide some information about the absolute numbers of threads and types of threads contained in all clusters, two pedestals are placed on the right, one at the bottom and one upside down at the top (Figure 3d). The lower pedestal shows the absolute number of threads in the data set with respect to the source-code artifact, here, threads executing a given Java method or any method of a given Java class. In contrast, the upper pedestal displays the absolute number of types of threads that executed the source-code artifact. In Java, these types are `java.lang.Thread` and its subclasses. These two numbers help the developer to estimate the size of each cluster and also provide an additional hint on how the clusters might be distributed across different thread types. Thus, it becomes easier to create hypotheses on how the code is actually being executed amongst the threads. In order to inspect and get more detail on the data represented by the ThreadRadar, the tool provides a detail view which we describe in Section 4.2.

3.3 Considering different Levels of Abstraction

The ThreadRadar of a class in Java aggregates the profiling data of all methods in this class. Therefore, the visualization shows the runtime distribution of thread clusters across every method in that class. To get more information about particular methods, the developer can investigate the ThreadRadars of those methods. This opens the possibility to compare methods of a single class with each other, but also with methods of other classes.

Since the visualization presents its information on a thread-cluster basis, we provide a thread-filtering mechanism introduced in the next section. It allows to create the visualizations for arbitrary subsets of threads.

As mentioned in Section 2, we also cluster threads on the level of the entire program based on their computational runtime and apply the ThreadRadar visualization on that level. Unlike the other source-code artifacts, there is no natural location in the source code representing the entire program. Therefore, we put the ThreadRadar for the program in the *Overview Window*, which is also presented in the next section.

4 IMPLEMENTATION

We implemented our approach as a plugin for an integrated development environment IntelliJ IDEA [10].

4.1 At a Glance

The plugin allows to start the currently selected run configuration with profiling enabled. Such a run consists of four phases: The collection of profiling data, the post-mortem processing of the data, i.e. the calculation of the computational runtimes, the matching of the profiling results to source-code elements in the IDE and the creation as well as the display of corresponding visualizations.

The data collection phase in CodeSparks-JPT is realized through a JVMTI agent [17] which implements statistical profiling based on stack sampling. After the JVM initialisation, the agent periodically records the stack traces of all Java threads of any state (Runnable, Waiting etc.) together with additional meta data, such as the class of

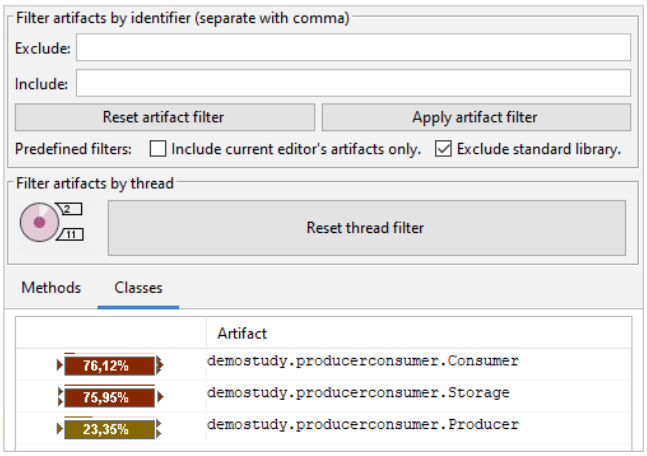


Figure 4: Overview window showing the list of classes.

the threads and the corresponding Java file. This profiling approach provides very low runtime overhead at the expense of accuracy. The sampling rate is preset to 3ms, but can be adjusted. Altogether, it provides heuristic information about the runtime behavior of the program under inspection. In the post-mortem processing phase, the thread-aware runtime metrics and other profiling results are calculated. The profiling results are matched to the source-code artifacts using the *Program Structure Interface* (PSI) [1]. The plugin uses information from the PSI to place the visualizations relative to the source-code artifacts.

Beck et al. [5] proposed a visualization for profiling-data on a method level for the Eclipse IDE. We implemented similar visualizations. Furthermore, we extended their visualization to class level and developed a novel visualization for thread-related data on different levels of abstraction (Section 3).

4.2 Other Features

In addition to the in-situ visualizations, the plugin displays more information in two popup windows which contain several other features mutually enhancing each other.

4.2.1 Overview Window. The overview window (Fig. 4) provides two lists of source-code artifacts (methods and classes) sorted in descending order according to their computational runtimes. Through the entries in the list, it is possible to navigate to the respective source-code artifact in the editor. The ThreadRadar placed in the overview window corresponds to the thread clustering for the entire program. It shows the clustering for all recorded threads based on their computational runtime over all source-code artifacts. It can serve as a starting point for debugging sessions and provides an overview of all threads which are currently selected or deselected by the filter.

4.2.2 Detail View. The in-situ thread visualizations only give a rough impression of the data. A click on a ThreadRadar opens a popup window presenting detailed information (Fig. 5). The detail view consists of four areas, the info panel area (orange), the preview area (blue), the thread selection area (green) and the global controls (purple). All threads that executed the source-code artifact are listed in an indented-tree view in the thread-selection area. There are two

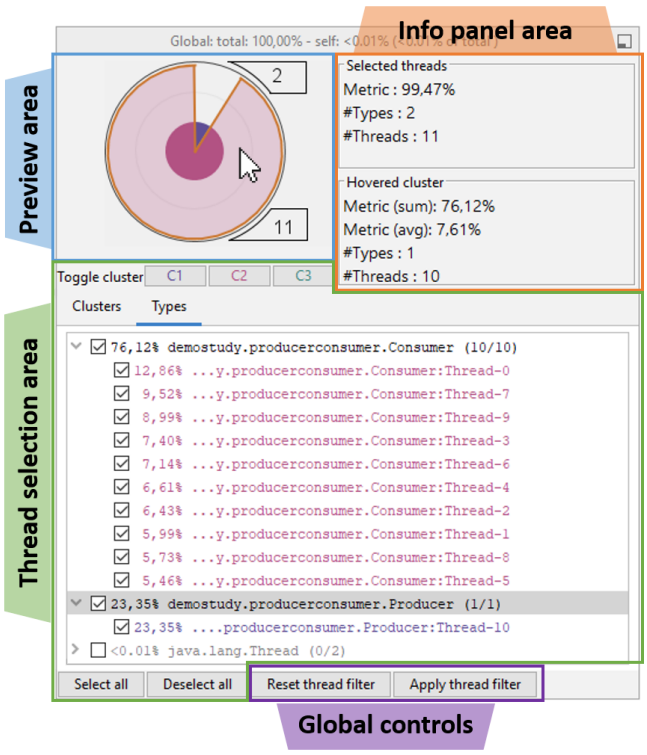


Figure 5: Detail view with applied thread filter.

tabs which differ in the sorting of the threads – either by cluster or by type. Each entry is composed of the name of the thread and its computational runtime with respect to the source-code artifact and is colored in the same color as the sector representing the cluster which contains the thread. The entries are grouped by categories (either the name of the cluster or the thread type). With the checkboxes, the developer can select single threads or a complete category at once. The info panel shows detailed information on both, the entirety of the current thread selection and the currently hovered cluster in the preview area in two independent sub-panels. This information includes the sum of the computational runtimes, the number of different thread types and the total number of threads in the selection.

Above all, the detail view allows to configure thread filters which are defined through the thread selection. The application of these filters results in the exclusion of all deselected threads globally from the visualizations. A filter defines the threads considered in the clustering algorithm in order to exclude less interesting threads, which enables an incremental inspection of the data on the basis of selected threads. Together with existing IDE features, the plugin, especially the ThreadRadars, form a powerful source of information for programming tasks which include program comprehension. In particular, this applies to debugging of concurrent Java programs as we showcase in a demonstration study in the next section.

5 DEMONSTRATION STUDY

To demonstrate our approach and, in particular, to illustrate the usefulness of the ThreadRadar, in this section, we present two examples. For the first example, we present more detail and introduce

```java
public class Worker extends Thread {
 public void run() {
  WorkerManager.getInstance().register(this);
  while(!Thread.currentThread().isInterrupted()) {
   /* virtual load */
   WorkerManager.getInstance().printWorkers();
   WorkerManager.getInstance().interrupt(this);
  }
  workerThreadDone(); }
 private void workerThreadDone() {
  printf("%s done\n", Thread.currentThread().getName());
 } }
```

```java
public class WorkerManagerMain {
 public static void main(String[] args) {
  final int nrOfThreads = 10;
  for (int i = 0; i < nrOfThreads; i++) {
   new Worker().start();
  }
  timeout();
 }
 private static void timeout() {/* termination */} }
```

```java
public class WorkerManager {
 private static WorkerManager instance = null;
 final Set<Thread> registeredWorkers = new HashSet<>();
 static WorkerManager getInstance(){
  /* virtual load */
  if (instance == null) instance = new WorkerManager();
  return instance; }
 private WorkerManager(){/* virtual load */}
 public void register(Thread worker){
  synchronized (registeredWorkers){
   registeredWorkers.add(worker);
  } }
 public void interrupt(Thread worker){
  synchronized (registeredWorkers){
   if (registeredWorkers.contains(worker)){
    registeredWorkers.remove(worker);
    worker.interrupt();
   } } }
 public void printWorkers(){
  synchronized (registeredWorkers){
   for (Thread thread : registeredWorkers)
    println(thread.getName());
   } println("------------");  } } }
```

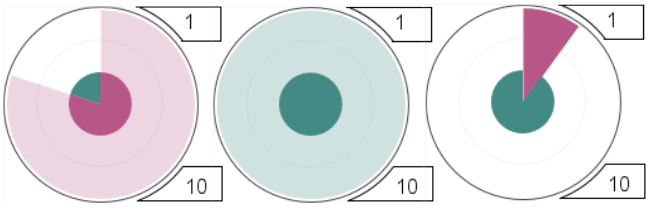
Listing 1: Truncated Java code of Bug 1. The full Java code is available in the supplementary material [15].

the program code itself as well as the test case, and explain how we conduct inspections with the help of the ThreadRadar. Finally, we explain how the visualization points to the actual issue and propose a fix. **In the supplementary material [15], we provide the plugin, the source code of the two examples as well as a video briefly showcasing the features of our tool.**

5.1 Data Race (Bug 1)

A prominent class of non-deadlock concurrency bugs are data races due to insufficient synchronization. In the first example, we present such a bug. The test case, i.e. class `WorkerManagerMain` (Listing 1) creates ten worker threads and terminates the run after five seconds. Otherwise, the program would run infinitely which is the observable manifestation of the concurrency bug. The intended behavior of the program is that worker threads register on a worker manager when they start. Until the worker threads get the signal to stop, they keep running in a loop. After each iteration, the workers inform the worker manager that they are ready to be interrupted (class `Worker` in Listing 1). The worker manager is implemented as a singleton. When a worker thread calls the method `interrupt(this)` on the worker manager instance, it is removed from the set of registered workers and the signal to interrupt is sent (class `WorkerManager` in Listing 1).

When we run the test case, the ThreadRadar of the program (Figure 6a) shows two thread clusters of in total ten threads (lower pedestal) of one thread type (upper pedestal). Note, at this point the threads of type `java.lang.Thread` were already excluded.



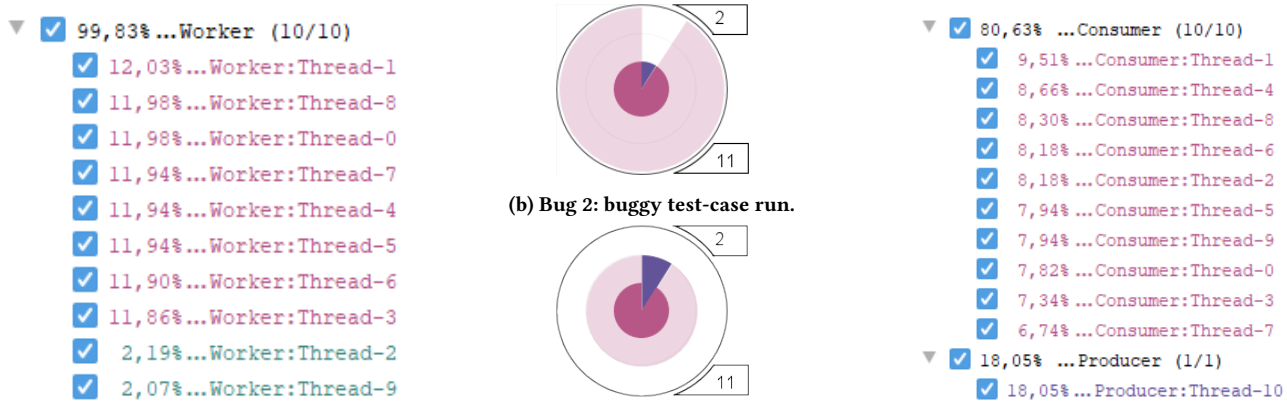
(a) Entire program. **(b) Constructor of `WorkerManager`.** **(c) Entire program with fix.**

Figure 6: ThreadRadars from test-case runs of Bug 1.

We observe that a few worker threads have substantially lower computational runtime. A look at the detail view of the ThreadRadar of the program and especially at the type list confirms this, as two worker threads turn out to be outliers (Figure 7a). Consequently, we inspect the class `Worker` and in particular its `run()`-method. Not surprisingly, the ThreadRadar of the `run()`-method and the entire program are identical. The callee list of the `run()`-method reveals that the callee consuming most computational runtime is the method `getInstance()`. Therefore, we further inspect that method which again, has a similar ThreadRadar as method `run()` and the program (Figure 6a and 1). The only callee of the method `getInstance()` that could affect runtime is the constructor of the class `WorkerManager`. The lower pedestal of the ThreadRadar of the constructor method shows that it is executed by ten threads, namely all worker threads (Figure 6b and 1). This is suspicious because in the singleton pattern, only one thread at all should execute the constructor. Now we have found the defect in the code: The method `getInstance()` is not properly synchronized. As a consequence, multiple instances of the `WorkerManager` class exist and it is non-deterministic with which instance a worker thread registers. In general, how many threads are registered in how many instances is also non-deterministic. In a thread-safe implementation of the singleton pattern only one thread is ever allowed to call the constructor of the singleton class. A common fix to this bug is to use double-checked locking within the method `getInstance()`. A run of the test case with this fix results in a ThreadRadar of the program showing that a single thread consumes considerably more computational runtime than all others, namely the thread that calls the constructor of class `WorkerManager` (Figure 6c). Furthermore, in the fixed version the lower pedestal of the ThreadRadar of the constructor of class `WorkerManager` always shows one thread executing this method (in contrast to Figure 6b and 1).

5.2 Synchronization Granularity Issue (Bug 2)

In the second example, we investigate a Java implementation of the producer/consumer pattern. In this pattern, two types of threads, producer and consumer threads, are involved. Since all threads access a shared storage, it requires synchronization. To achieve



(a) Bug 1: buggy test-case run.

(b) Bug 2: buggy test-case run.

(c) Bug 2: run with the fix.

(d) Bug 2: buggy test-case run.

Figure 7: ThreadRadars of the entire program for Bug 2 (a, b) and respective lists of thread types in the detail view (c, d).

maximal throughput in terms of entities produced and consumed per second, the synchronization has to be implemented as fine grained as possible. Since the complexity of the computations actually performed by producer and consumer threads are very similar, we further expect that both, the aggregated computational runtime of producer and consumer threads will each make up approximately 50% of the total computational runtime independent of the actual number of threads used. In the example, a builtin Java monitor together with corresponding calls to the methods `wait()` and `notify()` is used to synchronize the access to the storage. Furthermore, one producer and ten consumer threads are created. In Figure 7b, the ThreadRadar of the entire program shows that the source-code artifacts are executed by two clusters of threads. The red cluster contains many threads with high total and low average computational runtime, while the blue cluster contains few threads with both, low total and low average computational runtime. The detail view (Figure 7d) reveals that the two clusters match the thread types, i.e. the red cluster solely consists of consumer and the blue cluster of producer threads, respectively. The performance bug consists in unnecessary notifications of threads on the monitor although their condition to wait on the monitor still holds. This results in unnecessary condition checks and postpones the activity of the thread which can interact with the storage. The defect is the use of a Java monitor which only provides one single implicit monitor condition. To fix this, we replace the builtin Java monitor (`synchronized`) with a monitor with explicit condition variables (`ArrayBlockingQueue` [16]). Applying and running this fix results in the expected distribution of computational runtime across the consumer and producer threads which is fully reflected by the ThreadRadar (Figure 7c).

5.3 Discussion

With the two examples, we demonstrated the usefulness of the ThreadRadar for investigating different kinds of bugs. With the first example, we investigated threads of a single type which distribute over two clusters. This example constitutes a non-deadlock concurrency bug due to a data race. With the second example, we investigated two types of threads which distribute over two thread clusters with no intersection. This example constitutes a performance bug due to synchronization granularity issues. Overall, with our two examples, we present use cases of the designed visualization across both, a major concurrency bug pattern [12] as well as synchronization performance issues [2]. Although the examples of our demonstration study are not directly real-world bugs, they where crafted on the basis of such bugs. Data race bugs similar to Bug 1 can be found in bug tracking systems of popular Java open-source projects such as *Apache Batik*. For instance, the concrete bugs *Batik-802*, *Batik-322* and *Batik-939* are concerned with non-blocking concurrency bugs due to a data race and their fixes also present a double-checked locking solution. For Bug 2 we found the producer/consumer pattern a fitting example. Not only because it is a well known problem to most software developers but also that it is a part of popular bug collections such as the *Software-Artifact Infrastructure Repository* [8]. Altogether, the ThreadRadar makes visible how the computational runtime is distributed amongst threads of the entire program as well as on the level of classes and methods. It enables the exploration of the underlying data through features such as thread and source-code artifact filtering. Furthermore, the visualization facilitates code comprehension and, as a consequence, debugging by providing valuable information and the ability to test hypotheses. Through the visualization, we can distinguish between buggy and intended program behavior which makes it possible to manually assess code changes and, in particular, bug fixes.

6 FIRST EMPIRICAL RESULTS

In a small-scale user study, we asked two PhD students from a different research group at our department to debug two concurrency bugs with and without the thread-aware visualizations. We used Bug 1 from the demonstration study and another one which we added to the supplementary material [15]. The participants had not only to find the defect, but also to familiarize with the code.

During the debugging sessions, the participants were encouraged to follow the *Think Aloud* approach to verbalize their thoughts. Afterwards, we conducted a group interview with both participants to recap their experiences with our tool. It turned out, that participant A actively used the visualizations whereas participant B mostly read the program code. Both participants did not only find

the bug when using the tool with the thread-aware visualizations, but also fixed it. In a subsequent focus-group interview, they both agreed that the thread-aware visualizations were helpful, in particular to verify their proposed bug fixes. One participant stated that the coloring (weak and strong saturation) of the sectors were comprehensible and allowed to obtain the required information at first sight. While the small-scale user study only provided first indications, that our tool, and with that our approach can support concurrency debugging, it certainly helped us to test our study design for future investigations.

Furthermore, we got formative feedback. One participant was missing an absolute runtime value in conjunction with the relative runtime metrics. The participant stated, that it was impossible to asses if the program was running "a second or a week". Although, this information was not necessary to solve the problem, the participant was curious to have that information. Another suggestion of one of the participants was to show the distribution of the runtimes of the threads within a cluster, because it could help to decide which cluster to inspect. To achieve this, a histogram of the artifact-related computational runtimes of the threads of a cluster could be added to the detail view.

7 LIMITATIONS

As shown in the demonstration study, our approach can lead to valuable insights for a variety of thread-related code issues. It requires a test case that consists of the potentially relevant classes and the program input that reproduces the suspicious behavior. Beyond that, there are several limitations to our approach which we discuss in the following.

Profiling. The use of a statistical profiling, here stack sampling, has well known drawbacks [14]. In our investigations of concurrency bugs, the heuristical data was sufficient to catch all thread behavior of interest. It was reasonable to utilize relative runtime consumptions to point out exceptional behavior in terms of work balance of threads and clusters of threads to each other. As mentioned in the small-scale user study, assessing if a program runs seconds, minutes etc. is hard with the use of relative runtimes. To tackle this, a combination of the statistical profiling with another approach is imaginable or to approximate the absolute runtime values on the basis of the relative runtimes. The latter would be highly error-prone, as assumptions have to be made about the degree of parallelism.

Many concurrency defects such as race conditions are based on event-ordering issues. With our approach, we focus on the visualization of symptoms of concurrency bugs rather than the extraction of the root cause in the first place.

Scalability. Long running programs that utilize many classes potentially exhibit a large number of threads and types of threads. Our clustering is currently limited to compute a maximum of three clusters. That might be too restrictive to describe the work balancing behavior of such program runs. A completely different metric to cluster threads might be appropriate. See for instance [18] where a set of transactions, i.e. essentially stack traces, is partitioned into groups of isomorphic tree structures to extract patterns. The use of the thread-filter feature can mitigate the problem of a vast number of threads and makes our approach partially applicable. In particular, the magic number of three clusters not only serves the purpose of a smooth visual integration of the data into the source-code editor but also turned out to be an adequate number of clusters for debugging the concurrency defects that we examined so far.

Concurrency bug types. In this work, we concentrated on concurrency bug types such as data races and synchronization granularity issues. Both of them have in common, that they are non-blocking. Another famous type of concurrency bugs are resource and communication deadlocks. In classical deadlocks, for instance dining philosophers, all involved threads will be blocked at a certain program point. Those are not directly visible in our approach because we omit data of blocked threads in favor of actively running threads. Even if we included all waiting and blocked threads in the clustering metric, we did not detect any visual anomalies in any of the examples we have examined so far. Therefore, at this stage, deadlocks are out of scope of our approach.

8 RELATED WORK

In their systematic mapping study on the visual augmentation of source-code editors [22] Sulír et al. found, that only few work exists which augments source code with data from dynamic analyses, such as profiling. For instance, Beck et al. [5] introduced in-situ visualizations for runtime-consumption data on the level of methods. While their methodology is applicable for single threaded programs and integrated into the Eclipse IDE, we implemented a similar approach for IntelliJ IDEA providing more levels of abstraction and in particular, we aimed at concurrent programs. Cito et al. integrated runtime-performance traces into the source-code editor by highlighting source-code elements, such as method calls and loops [6]. While they also provide tooltips to retrieve details on demand, their approach neither utilizes further visualizations nor delivers insights on concurrency. Senseo extends the IDE with interactive views and in-situ visualizations presenting various dynamic performance metrics [20]. Albeit their tool facilitates performance-optimization tasks, no concurrency data is considered. Alcocer et al. [3] introduce a radial sparkline visualization called *Spark Circle* integrated in commit graphs to visualize memory and execution time changes across different versions of the source code. While their focus lies on performance changes related to the evolution of the software system, we focus on concrete debugging of concurrency bugs.

There are many approaches to visualize concurrency based on program execution traces. Karran et al. introduce *SyncTrace*, a visualization technique combining multiple variants of icicle plots, edge bundling and various interaction possibilities for the analysis of dependencies between threads [11]. UML-based diagrams, in particular UML-sequence diagrams have been employed for visualization of concurrency [4, 13]. Both, the tool *ThreadCity* [9] as well as *SynchroViz* [27] use a specialised city metaphor to visualize the static structure of a software system and combine it with a traffic metaphor to outline thread activities. While *SynchroViz* focuses on synchronization concepts, *ThreadCity* works with function calls and makes use of pie charts to aggregate thread data. They group threads into three static categories (incoming, internal and outgoing calls). Furthermore, Röthlisberger et al. [21] utilize clustering to map continuous distributions of dynamic metric values to a discrete

six-valued scale. We similarly compute three thread clusters based on computational runtime consumption. Most approaches focus on data of inter-thread correspondence, such as caller/callee relation, blocking and synchronization. In contrast to that, we take advantage of the relative runtime consumption of threads aggregated for source-code artifacts.

9 CONCLUSION

We presented our concurrency debugging approach which especially consists of augmenting the source code with a thread-aware visualization in the source-code editor of Intellij IDEA. While similar approaches focus on the visualization of pure runtime consumption, we foster the awareness of source code being executed concurrently by different threads. Although we developed the visualization for the Java language in the first place, our approach is certainly applicable to other programming languages which make use of similar syntactical elements and programming-language concepts. Similarly, while the ThreadRadar shows distributions of runtime data, the visualization approach including the clustering could be applied to other profiling data like memory usage or energy consumption as well as other profiling approaches and metrics.

In a demonstration study, we applied our tool to analyze and fix two bugs, in particular a non-deadlock concurrency bug and a performance bug. With that, we showcased and discussed the usefulness of the ThreadRadar for program-comprehension tasks in the context of debugging. Subsequently, we outlined limitations of our approach on various aspects. Thus, future work includes scalability, other types of bugs, programming languages, and performance metrics in particular, leveraging also waiting and blocking time of threads. Through the conducted small-scale user study we gained first formative results for our tool, as well as first evidence that our approach can support the concurrency debugging process.

REFERENCES

[1] JetBrains s.r.o. 2020. Program Structure Interface (PSI) / IntelliJ Platform SDK DevGuide. https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html

[2] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 298–313. https://doi.org/10.1145/3064176.3064186

[3] Juan Pablo Sandoval Alcocer, Harold Camacho Jaimes, Diego Costa, Alexandre Bergel, and Fabian Beck. 2019. Enhancing Commit Graphs with Visual Runtime Clues. In *2019 Working Conference on Software Visualization, VISSOFT 2019, Cleveland, OH, USA, September 30 - October 1, 2019*. IEEE, 28–32. https://doi.org/10.1109/VISSOFT.2019.00012

[4] Cyrille Artho, Klaus Havelund, and Shinichi Honiden. 2007. Visualization of Concurrent Program Executions. In *31st Annual International Computer Software and Applications Conference, COMPSAC 2007, Beijing, China, July 24-27, 2007. Volume 2*. 541–546. https://doi.org/10.1109/COMPSAC.2007.236

[5] Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter Daniel Rey. 2013. In situ understanding of performance bottlenecks through visually augmented code. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. 63–72. https://doi.org/10.1109/ICPC.2013.6613834

[6] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C. Gall. 2018. PerformanceHat: augmenting source code with runtime performance traces in the IDE. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 41–44. https://doi.org/10.1145/3183440.3183481

[7] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Trans. Software Eng.* 35, 5 (2009), 684–702. https://doi.org/10.1109/TSE.2009.28

[8] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[9] Sebastian Hahn, Matthias Trapp, Nikolai Wuttke, and Jürgen Döllner. 2015. Thread City: Combined Visualization of Structure and Activity for the Exploration of Multi-threaded Software Systems. In *19th International Conference on Information Visualisation, IV 2015, Barcelona, Spain, July 22-24, 2015*. 101–106. https://doi.org/10.1109/iV.2015.28

[10] JetBrains s.r.o. 2021. All Developer Tools and Products by JetBrains. https://www.jetbrains.com/products.html#type=ide

[11] Benjamin Karran, Jonas Trümper, and Jürgen Döllner. 2013. SYNCTRACE: Visual thread-interplay analysis. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*. 1–10. https://doi.org/10.1109/VISSOFT.2013.6650534

[12] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*. 329–339. https://doi.org/10.1145/1346281.1346323

[13] Katharina Mehner. 2005. *Trace based debugging and visualisation of concurrent Java programs with UML*. Ph.D. Dissertation. University of Paderborn, Germany. http://ubdata.uni-paderborn.de/ediss/17/2005/mehner/disserta.pdf

[14] Andy Nisbet, Nuno Miguel Nobre, Graham D. Riley, and Mikel Luján. 2019. Profiling and Tracing Support for Java Applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*. ACM, 119–126. https://doi.org/10.1145/3297663.3309677

[15] Oliver Moseler, Lucas Kreber, and Stephan Diehl. 2021. Supplementary Material to ThreadRadar: A Thread-Aware Visualization for Debugging Concurrent Java Programs. https://doi.org/10.5281/zenodo.4753849

[16] Oracle. 2020. ArrayBlockingQueue (Java SE 11 & JDK 11). https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ArrayBlockingQueue.html

[17] Oracle. 2020. JVM(TM) Tool Interface 11.0.0. https://docs.oracle.com/en/java/javase/11/docs/specs/jvmti.html

[18] Wim De Pauw, Sophia Krasikov, and John F. Morar. 2006. Execution patterns for visualizing web services. In *Proceedings of the ACM 2006 Symposium on Software Visualization, Brighton, UK, September 4-5, 2006*. ACM, 37–45. https://doi.org/10.1145/1148493.1148499

[19] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor Filho, and Roberto Souto Maior de Barros. 2015. A large-scale study on the usage of Java's concurrent programming constructs. *Journal of Systems and Software* 106 (2015), 59–81. https://doi.org/10.1016/j.jss.2015.04.064

[20] David Röthlisberger, Marcel Harry, Walter Binder, Philippe Moret, Danilo Ansaloni, Alex Villazón, and Oscar Nierstrasz. 2012. Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks. *IEEE Trans. Software Eng.* 38, 3 (2012), 579–591. https://doi.org/10.1109/TSE.2011.42

[21] David Röthlisberger, Marcel Harry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. 2009. Augmenting static source views in IDEs with dynamic metrics. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*. IEEE Computer Society, 253–262. https://doi.org/10.1109/ICSM.2009.5306302

[22] Matús Sulír, Michaela Bacíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual augmentation of source code editors: A systematic mapping study. *J. Vis. Lang. Comput.* 49 (2018), 46–59. https://doi.org/10.1016/j.jvlc.2018.10.001

[23] Herb Sutter and James R. Larus. 2005. Software and the concurrency revolution. *ACM Queue* 3, 7 (2005), 54–62. https://doi.org/10.1145/1095408.1095421

[24] Edward R. Tufte. 2006. *Beautiful Evidence*. Graphics Press, Cheshire, CT. 46–63 pages.

[25] Mattias De Wael, Stefan Marr, and Tom Van Cutsem. 2014. Fork/join parallelism in the wild: documenting patterns and anti-patterns in Java programs using the fork/join framework. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*. 39–50. https://doi.org/10.1145/2647508.2647511

[26] Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schrödl. 2001. Constrained K-Means Clustering with Background Knowledge. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 577–584.

[27] Jan Waller, Christian Wulf, Florian Fittkau, Philipp Dohring, and Wilhelm Hasselbring. 2013. Synchrovis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013*. 1–4. https://doi.org/10.1109/VISSOFT.2013.6650520