J Vis (2022) 25:1267–1289
https://doi.org/10.1007/s12650-022-00843-w

REGULAR PAPER

Oliver Moseler · Lucas Kreber · Stephan Diehl

The ThreadRadar visualization for debugging concurrent Java programs

Received: 1 February 2022 / Revised: 23 March 2022 / Accepted: 30 March 2022 / Published online: 24 May 2022

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 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 windows that also allow the user to configure thread filters interactively. 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 usability test and consequently implemented tool improvements.
Keywords Debugging · Concurrency · Performance · Java · Thread · Visualization

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 (Wael et al. 2014; Pinto et al. 2015). Developing correct concurrent programs is difficult (Sutter and Larus 2005). Therefore, programming libraries have been developed to facilitate this task. Unfortunately, as by the example of concurrent collections, such libraries are often misused which leads to defect prone software (Lin and Dig 2015). 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. Reading and understanding the source code is an integral element of every debugging task. The way source code is visually presented in traditional source-code editors does not convey much information on whether the code is executed concurrently or in parallel in the first place. Comprehending the dynamic behavior of source code and which phenomena occur when multiple threads

O. Moseler (✉) · L. Kreber · S. Diehl
Software-Engineering Group, University of Trier, Trier, Germany
E-mail: moseler@uni-trier.de

L. Kreber
E-mail: kreberl@uni-trier.de

S. Diehl
E-mail: diehl@uni-trier.de

execute the code requires a high level of abstraction and thus, causes a high mental load. Developers still lack thread-aware programming tools that facilitate the understanding of concurrent programs (Cornelissen et al. 2009). Ideally, these should be part of their daily work environment typically including an Integrated Development Environment (IDE).

In this work, we present our approach for visual support for debugging of performance and concurrency issues. We implemented our approach in the CodeSparks-JPT debugging tool that augments the source code with interactive glyph-based visualizations representing thread information 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. These metrics measure the relative runtime consumption of an artifact with respect to the overall profiled run of the Java program. Based on these metrics, we developed a glyph-based visualization (Borgo et al. 2013) showing how runtime consumption is distributed over clusters and types of threads executing the source-code artifacts. This visualization, hereinafter called a *ThreadRadar*, is embedded in the source code in a sparkline (Tufte 2006) manner (Fig. 1), i.e., as a word-sized graphic with typographic resolution. In combination with former in situ visualizations (Beck et al. 2013) and other tool features it enhances program understanding of concurrent programs. More detailed thread information can be retrieved on demand in a detail window. An overview of all recorded source-code artifacts of a program run is presented in an overview window. Both windows allow to further explore and filter the profiling data on the basis of threads and source-code artifacts.

This article is an extended version of our previous conference paper (Moseler et al. 2021). Next to small changes throughout the article, we added the following major extensions:

- Formal model of the stack sampling and the performance metrics in Sect. 2.
- Description of the intended workflow with our approach in Sect. 2.
- Extended description of Bug 2 in Sect. 5.2. In addition, the fix is validated using the ThreadRadar on examples of different producer-consumer thread ratios.
- Bug 3 that shows applicability on another class of concurrency bugs (race condition) in Sect. 5.3.
- More detail on the design and methodology of the formative usability test and its results in Sect. 6.
- Description of improvements in response to the usability test, in particular the *Histogram View* in Sect. 6.1.
- Extended discussion of limitations, in particular the paragraphs *Scalability* and *Formative Usability Test* in Sect. 7.
- Extended discussion of related work in Sect. 8.

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 person debugging the

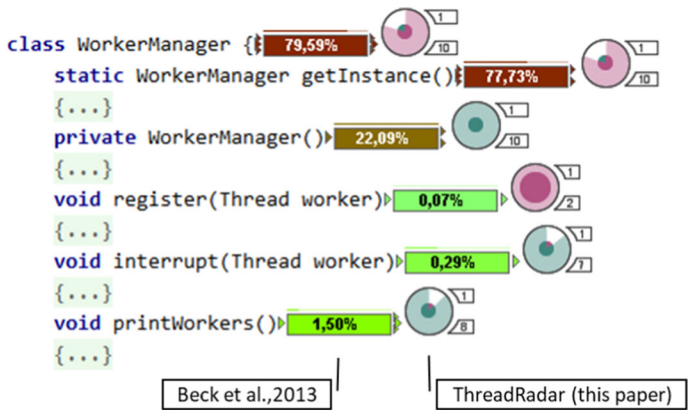


Fig. 1 Examples of the ThreadRadar based on a test-case run of Bug 1 (class `WorkerManager`, Sect. 5)

code already has a deep understanding of the source code to be debugged and expectations of 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. This consequently makes 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. Concurrent programming also employs techniques such as synchronization mechanisms to coordinate threads. Therefore, the execution of program code by different threads additionally includes the aspect of the state of the threads, 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 glyph-based visualizations presenting the thread information in the source-code editor:

2.1 Profiling

We periodically sample the stack traces of all threads of a running program. Formally, we model a stack frame as a tuple $f = (c, m, l)$ of source-code artifacts where l is the line of code of the method m in class c executed by the method call which is represented by this stack frame. A sample captures the stack of each thread at a certain point in time. Thus, a sample with time stamp τ is modeled as a set $\sigma = \{(\tau, th_1, fs_1), \ldots, (\tau, th_q, fs_q)\}$ where $th_j \in TH$ are pairwise disjoint thread identifiers and $fs_i = (f_1, \ldots, f_k)$ are stack traces, i.e., sequences of stack frames.

According to the Java specification, threads can be in various states (New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated). As we want to focus on the runtime that a thread is active, and ignore when it is blocked resp. waiting on synchronization mechanisms or sleeping, we only consider the stack traces of those threads with state *Runnable*. As a consequence, waiting time on operating system resources such as the processor and I/O is still captured by our approach according to its definition (Oracle 2020c). Thus, imbalance issues according to such resources can be investigated. Furthermore, attempts to

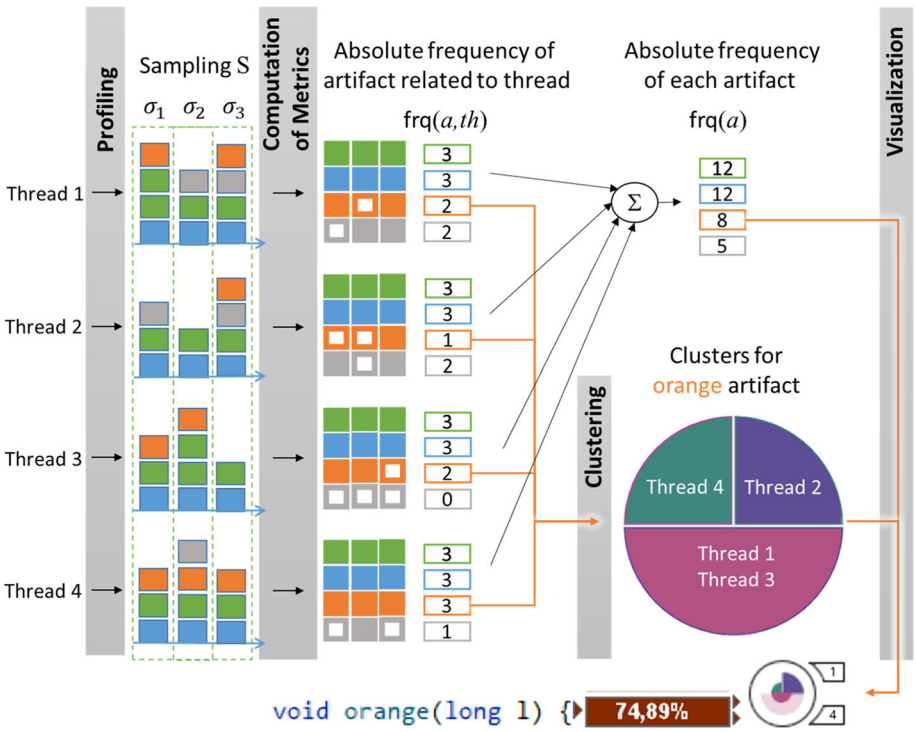


Fig. 2 Approach: Profiling, computation of metrics, clustering and visualization. The computational runtime of each artifact **a** is the absolute frequency of **a** divided by the total number of threads sampled, here FRQ is 12 = 3 samples * 4 threads (see Equations 2, 4, and 5). The clustering for the artifact is based on the artifact-relative computational runtime of the threads (see Equation 6)

(re)enter Java monitors are captured because for that to happen, the corresponding thread has to switch its state to *Runnable*.

Putting all pieces together, a sampling $S = (\sigma_1, \ldots, \sigma_n)$ is a sequence of samples with disjoint time stamps.

2.2 Runtime metrics

Based on a sampling, we first compute how frequent a source-code artifact a occurs in all sampled stack traces of a thread th as well as in all sampled stack traces of all threads:

$$\mathsf{frq}(a, th) = |\{(\tau, th, fs) \in \sigma \mid \sigma \in S \wedge a \in f \in fs\}| \tag{1}$$

$$\mathsf{frq}(a) = \sum_{th \in TH} \mathsf{frq}(a, th) \tag{2}$$

On the level of the entire program, we compute how often a certain thread was active when a sample was taken as well as the total number of sampled stack traces:

$$\mathsf{frq}(th) = |\{(\tau, th, fs) \in \sigma \mid \sigma \in S\}| \tag{3}$$

$$\mathsf{FRQ} = \sum_{\sigma \in S} |\sigma| \tag{4}$$

We use the term *computational runtime* to indicate that we only consider threads of state *Runnable* as discussed above. To make the frequencies of different artifacts more comparable, we put them in relation to FRQ and define the computational runtime of a source-code artifact a as:

$$\mathsf{crt}(a) = \frac{\mathsf{frq}(a)}{\mathsf{FRQ}} \tag{5}$$

It indicates the relative frequency of the source-code artifact in the stack traces of all threads. We further disassemble the computational runtime of a into a-relative runtimes of each thread th, i.e., the ratio of the runtime that th spends executing code in a and the total artifact runtime of a:

$$\mathsf{crt}(a, th) = \frac{\mathsf{frq}(a, th)}{\mathsf{frq}(a)} \tag{6}$$

Consequently, for the entire program, the computational runtime of a thread th is

$$\mathsf{crt}(th) = \frac{\mathsf{frq}(th)}{\mathsf{FRQ}} \tag{7}$$

Based on the sampling S we also compute method, self, caller, and callee times for the in situ visualizations that do not discriminate between threads (Beck et al. 2013).

2.3 Clustering

For each source-code artifact a, we cluster its executing threads on the basis of the a-relative computational runtime of the threads. Initially, we used k-means clustering, but found in experiments that often threads were grouped into different clusters due to the inaccuracy of statistical profiling rather than other effects like scheduling, synchronization or garbage collection.

To make the clustering less sensible to the small inaccuracies and more stable over multiple runs of the same test case, we utilize the constrained k-means clustering algorithm (Wagstaff et al. 2001). We 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 use $k = 3$ to compute a maximum of three clusters and initialize 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.

2.4 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 Sect. 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. It reduces the required visual (in terms of shapes, colors and pixel) as well as the cognitive (in terms of human memory and thought) effort. From our experience so far, we found that the three clusters are often sufficient to draw conclusions about concurrency or performance issues, e.g., one of the clusters mostly contains threads with unexpected behavior or threads with the same expected behavior are spread across multiple clusters. The latter can be the case when threads are of the same type and thus expected to execute the same code. This turns out to be a helpful heuristic for debugging (Sect. 5). Limitations arising from the restriction to three clusters as well as the mentioned assumptions are discussed in Sect. 7.

The neat integration of our approach into the IDE enables a natural debugging workflow: Run the program under inspection with given inputs and profiling enabled, explore the profiling data with the help of the ThreadRadars and other features, make hypotheses on either performance-decreasing or defect-inducing circumstances, modify the source code, rerun the program, evaluate the hypotheses and start over. Due to the stack-sampling approach of the data collection, multiple runs are recommended since the profiling results might vary.

3 In Situ ThreadRadars

The conceptual design of our 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 ThreadRadar glyph and discuss how it applies to different levels of abstraction.

3.1 Visual requirements

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

3.1.1 Placement

The artifacts we address are Java classes and methods. The most intuitive placement for a ThreadRadar 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 ThreadRadar, namely the line of the header of the declaration itself plus the empty line before the header.

3.1.2 Compactness and 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 much the underlying data grows.

3.2 The ThreadRadar glyph step by step

We illustrate the development of ThreadRadar glyph, for Java classes and methods by step-wise refinement. We started with a simple pie chart approach. By applying the clustering strategy outlined in Sect. 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 glyph. 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 drawing area, we do not use a continuous scale, but instead a three-valued scale to depict the average metric value.

Note, we map low (<33%), medium ($\geq$ 33% and <66%) and high ($\geq$ 66%) metric values to the first, second and third level of the three-valued scale, respectively.

To facilitate reading of the sector radii, we add circular lines indicating the levels of the three-value scale (Fig. 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 glyph 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 (Fig. 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.

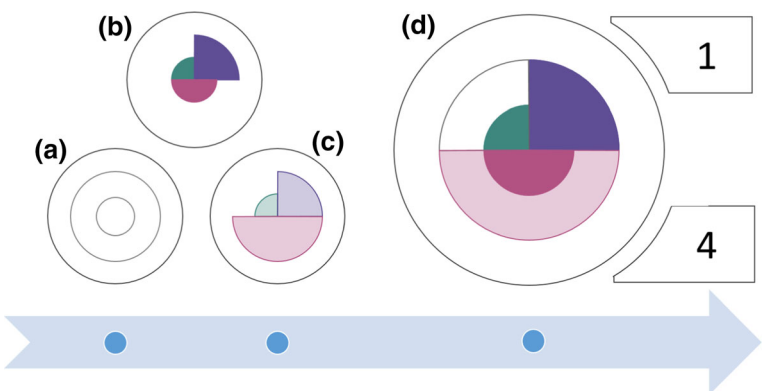


Fig. 3 Combining the circular lines indicating the three-valued scale (**a**) with the pie chart with radii indicating average computational runtimes (**b**) and the pie chart with weaker colored sectors and radii indicating the sum of the computational runtimes (**c**) finally yields the ThreadRadar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

Drawing the three components discussed above on top of each other into a single diagram yields the *ThreadRadar* (Fig. 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 (Fig. 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 glyph, the tool provides a detail view in an on-demand popup window which we describe in Sect. 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, it shows the runtime distribution of thread clusters across every method in that class. The ThreadRadars of all methods of a class are directly accessible and immediately visible, since they are also present in the source code of the class. If the class has too many or too long methods to fit on the screen at once, IDE features such as code folding can be used to work around this problem (like done in Fig. 1). To get more information about particular methods, the developer can investigate the ThreadRadars of those methods by comparing them visually. Since all ThreadRadars refer to the same program run, potential caller/callee relations of the methods may be detectable. Moreover, direct relationships of a ThreadRadar for a single method to the ThreadRadar of its class become apparent when, e.g., a large portion of the runtime consumption of a class is attributable to a single one of its methods.

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

As mentioned in Sect. 2, we also cluster threads on the level of the entire program based on their computational runtime and apply the ThreadRadar 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 *Program ThreadRadar* in the *Overview Window*, which is also presented in the next section.

4 Implementation

We implemented our approach in a tool named CodeSparks-JPT (short for **J**ava, **P**erformance and **T**hreads) as a plugin for an integrated development environment, namely IntelliJ IDEA (JetBrains s.r.o. 2021).

4.1 At a glance

The plugin allows the user 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 (Oracle 2020b) which implements statistical profiling based on stack sampling. After the JVM initialization, 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 currently executed Java byte-code instruction, the class of 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) (JetBrains s.r.o. 2020). It provides a decorated abstract syntax tree (PSI tree) for code files consisting of PSI elements, which can be matched both to source-code artifacts of the programming-language as well as to their dynamic instances. In particular, a PSI element stores its placement within the visible area of the source-code editor. The plugin uses information from the PSI to place the visualizations relative to the source-code artifacts.

We implemented method-level visualizations for profiling-data similar to those proposed by Beck et al. (2013) for the Eclipse IDE. In their visualization, the colored rectangles depict the total relative computational runtime of a given source-code artifact. We extended their visualization to class level and developed a novel visualization for thread-related data on different levels of abstraction (Sect. 3).

4.2 Other features

In addition to the ThreadRadars embedded in the source code, 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. In addition, the overview window supports the filtering of the lists on the basis of the artifact identifiers. To this end, two text fields are provided to define inclusion and exclusion rules. Furthermore, predefined filters can be applied by selecting the corresponding checkbox. One excludes all standard library artifacts from the lists and the other excludes all artifacts but those in the currently opened file in the source-code editor. The Program 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 window

The ThreadRadar glyphs 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 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 artifact-relative computational runtime 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. In case a thread is deselected, the corresponding text turns grey. With every change in the selection of the threads, the ThreadRadar in the preview area updates accordingly. Within the preview area, we repeat the ThreadRadar in an enlarged version to facilitate the inspection of the thread clusters. When hovering over a sector with the mouse, the border of the sector is colored in orange. 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 window 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 ThreadRadars. 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 demonstrate in the next section.

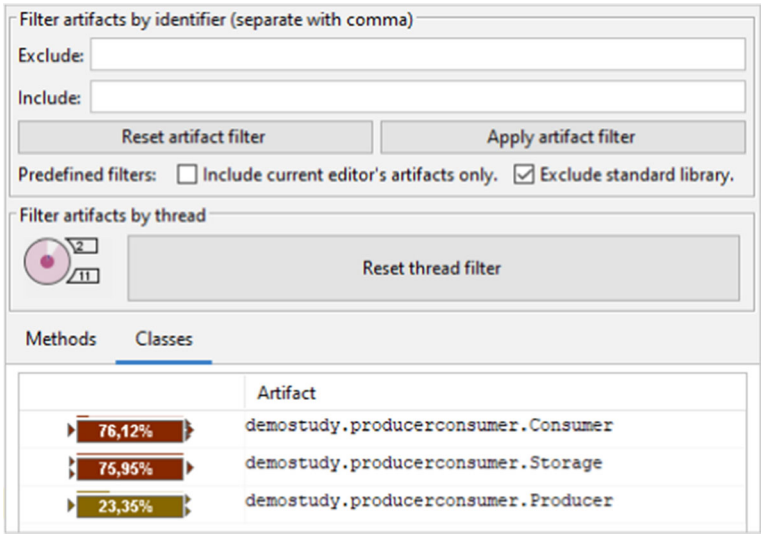


Fig. 4 Overview window showing the list of classes

5 Demonstration study

To demonstrate our approach and, in particular, to illustrate the usefulness of the ThreadRadar, in this section, we present three examples. For the first example, we present more detail and introduce the program code itself as well as the test case, and explain how we conduct inspections with the help of the ThreadRadars. Finally, we explain how the ThreadRadars point to the actual issue and propose a fix. In the supplementary material (Moseler et al. 2022), we provide the plugin, the source code of the three examples as well as a video briefly showcasing the features of our tool.

All demonstrations were conducted on a Windows 10 PC (16GB RAM, Intel Core i7 8650U CPU with 4 cores/8 threads) connected to a full HD 27" screen using IntelliJ IDEA version 2020.3.1 and AdoptOpenJDK version 11.0.9+11 with HotSpot JVM.

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 Program ThreadRadar (Fig. 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 using the thread filtering mechanism of the detail window (Sect. 4.2).

We observe that a few worker threads have substantially lower computational runtime. A look at the detail window of the Program ThreadRadar and especially at the type list confirms this, as two worker threads turn out to be outliers (Fig. 6b). Consequently, we inspect the class `Worker` and in particular its `run()`-method. Not surprisingly, the ThreadRadar of the `run()`-method and the Program ThreadRadar 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 the ThreadRadar of that method which again is similar to the ThreadRadar of method `run()` and the Program ThreadRadar (Fig. 6a and 1). The

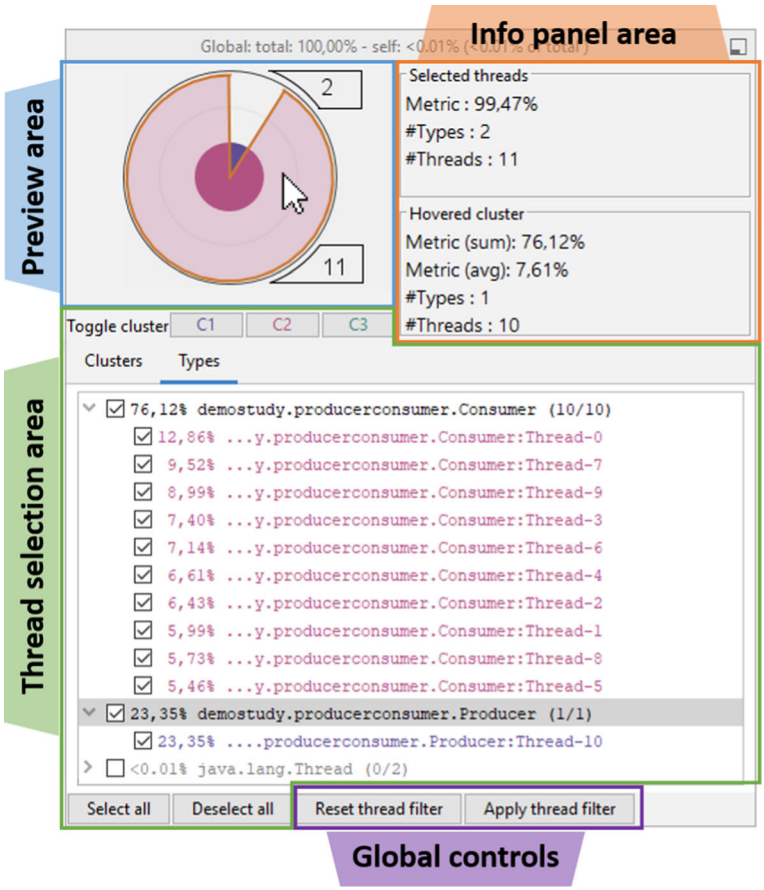


Fig. 5 Detail view with applied thread filter

only callee of the method `getInstance()` that could affect runtime is the constructor of the class `WorkerManager`.

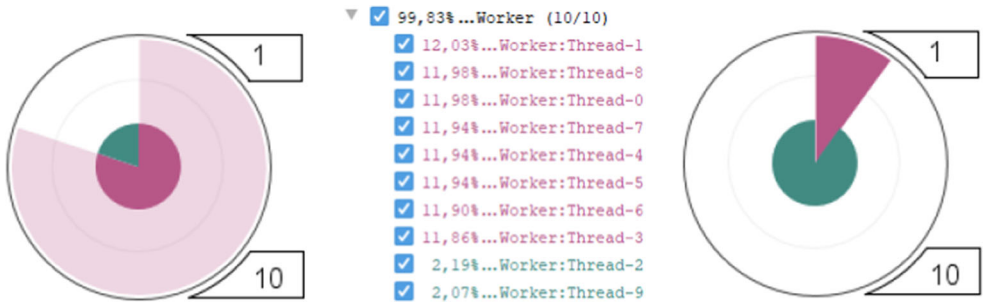
The lower pedestal of the ThreadRadar of the constructor method shows that it is executed by ten threads, namely all worker threads (Fig. 7 and 1).

This is suspicious because in the singleton pattern, only one thread 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 Program ThreadRadar showing that a single thread consumes considerably more computational runtime than all others, namely the thread that calls the constructor of class `WorkerManager` (Fig. 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 Fig. 7 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. The producer threads produce entities and put them in a storage, while the consumer threads take entities from the storage and consume them. Since all threads access a shared storage, it requires synchronization. To achieve maximal throughput in terms of entities produced and consumed per second, the synchronization has to be implemented as fine



(a) A run of the test case with the bug.

(b) The corresponding list of thread types to (a).

(c) A run of the test case with the fix.

Fig. 6 Program ThreadRadars of test-case runs of Bug 1 and corresponding thread type list in the detail view

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 built-in Java monitor together with corresponding calls to the methods `wait()` and `notifyAll()` is used to synchronize the access to the storage. Furthermore, one producer and ten consumer threads are created. Since the test-case would run infinitely, it is constructed such that a run will be terminated after 10 seconds by a dedicated daemon thread. Also, code to calculate and print the throughput of the producer and consumer threads is added. Note, since the tool uses statistical profiling, in particular stack sampling, we added some code to cause some virtual workload.

In Fig. 8a, the Program ThreadRadar 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 window (Fig. 8b) 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. We also see that the single producer thread consumes 18.05% computational runtime, while the consumer threads have an aggregated computational of 80.63% which is not what we expected.

Furthermore, the throughput results in 147 units/s for the producer thread and 146.9 units/s for the consumer threads for this test-case run. The fact that the numbers are almost identical indicates a correct synchronization of the threads. The virtual load is set so that the production of one unit takes at least one millisecond. Thus, by ignoring all other computational effort in the test-case, in particular the thread synchronization, a maximal throughput of 1000 units/s is possible. Hence, we conclude that the current implementation of the producer/consumer pattern is not optimal and thus, according to Nistor et al. (2013) and Jin et al. (2012), contains a performance bug.

The producer/consumer pattern is symmetric, i.e., while the consumer threads wait on the storage being non-empty, the producer threads wait on the storage being not full. Furthermore the consumers take entities from the storage whereas the producers add entities to the storage. So when a producer adds an entity to the storage, the storage becomes not empty which means that a consumer is then able to take an entity from the storage and vice versa. Hence, the producer threads should only notify consumer threads and vice versa. In the implementation of the producer/consumer pattern used for this test-case (Moseler et al. 2022), threads of any type notify all threads waiting on the shared monitor independent of their type (`notifyAll()`). This results in both unnecessary competition to gain the monitor lock as well as unnecessary checks of the condition.

Altogether, 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 built-in Java monitor (`synchronized`) with a monitor with explicit condition variables (`ArrayBlockingQueue` (Oracle 2020a)).

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 Program ThreadRadar (Fig. 8c).

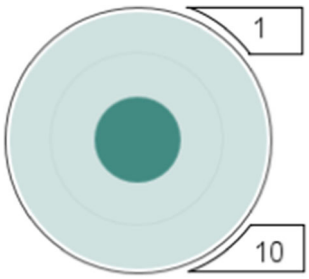


Fig. 7 ThreadRadar of the constructor of class `WorkerManager` of a test-case run of Bug 1

In particular, changing the number of consumer threads does not influence the expected distribution of the computational runtime across the two thread types. If there is exactly one producer and one consumer thread (ratio 1:1), the Program ThreadRadar displays one cluster with an average computational runtime of approx. 50%, because the clustering algorithm naturally places the two threads in the same cluster (Fig. 9a). For other producer-consumer ratios, the Program ThreadRadar always displays two thread clusters, each with approx. 50% aggregated computational runtime, but with different sector angles reflecting the ratio of consumer to producer threads (Fig. 9b and 9c). In particular, there is one cluster containing the single producer thread and one cluster containing all the consumer threads. For the former cluster, the radius of the cluster sector always indicates 50% total as well as average computational runtime independent of the number of threads in the latter cluster.

This indicates that we reached the optimal implementation with respect to throughput, which is further confirmed by the computed values (992.1 units/s for the consumer and 992.3 units/s for the producer threads) which are close to the maximal throughput.

5.3 Race condition (Bug 3)

In this example, we look at a non-deadlock concurrency bug which consists of a race condition between multiple threads where the concrete execution order of the statements executed by different threads is crucial.

The bug consists in reader and writer threads accessing a shared variable where the termination of the reader threads depends on a concrete value of that shared variable (tested by ==), while the writer threads increment the value.

Although the shared variable is properly protected by a Java monitor to guarantee mutual access, this is not sufficient to control the order in which the threads access the shared variable and thus to prevent that some reader threads miss the concrete value to terminate. In Fig. 10a, we present the Program ThreadRadar of a run of the corresponding test case. It shows that twelve threads of two types, namely ten reader plus two writer threads, have been partitioned into two clusters. The angle of the green cluster sector takes up a third

(a) A run of the test case with the bug.

(b) The corresponding list of thread types to (a).

(c) A run of the test case with the fix.

Fig. 8 Program ThreadRadars of test-case runs of Bug 2 and corresponding list of thread types in the detail view

of the complete circle and thus contains four threads. It follows, that at least one cluster contains threads of different thread types. Looking at the list of threads of the green cluster, we find that it contains two writer and two reader threads (Fig. 10b).

Therefore, the green cluster is the most interesting. It has both low aggregated as well as average computational runtime. In contrast, the red cluster only consists of reader threads and has high aggregated and low average computational runtime. In this run, two reader threads were able to terminate. All other reader threads kept on running resulting in a comparably higher computational runtime. The Program ThreadRadar reflects this behavior. Consequently, the ThreadRadar for the reader-thread class and, in particular its `run()`-method shows that the reader threads are divided into two clusters. Namely the threads that terminated and the threads that kept running. Therefore, investigating the cause of the different runtime behavior of the reader threads is an important aspect of uncovering the bug. Note that the concrete number of terminating threads may vary due to non-deterministic thread interleaving. Theoretically, it is possible for any number of reader threads to terminate.

A valid fix to this concurrency bug is to change the condition of the reader threads to test the shared variable using $>=$ instead of $==$. Since the code uses a form of a *spin-wait* which is considered an anti-pattern (Hallal et al. 2004), another solution is recommended, such as using a monitor and appropriate conditions that can be signaled and waited for. In Fig. 10c, we present a Program ThreadRadar from a test case run with the fix. It can be seen that twelve threads of two types are grouped together in one single cluster, namely the ten reader and two writer threads. Note, any other thread type such as `java.lang.Thread` is already excluded with the help of the thread filtering mechanism. Furthermore, the threads of the cluster have a small average but a high aggregated computational runtime. The expected correct behavior is that all reader threads terminate right after the writer threads terminate. Thus, all involved threads have similar computational runtime with no outliers. Again, the Program ThreadRadar reflects this behavior.

5.4 Discussion

With the three 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. The third example represents a non-deadlock concurrency bug due to a race condition, where the order of read and write accesses to a properly protected shared variable is crucial. This example showcases our approach on two thread types which distribute over two clusters but with intersection.

Overall, with our three examples, we present use cases of the designed visualization across both, two major concurrency bug patterns (Lu et al. 2008) as well as synchronization performance issues (Alam et al. 2017). Although the examples of our demonstration study are not directly real-world bugs, they were 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* (Do et al. 2005). Besides this, Bug 3 is derived from the bug *dbcp4* (*DBCP-271*) of the JaConTeBe (Lin et al. 2015) collection which actually

(a) Ratio 1:1 **(b)** Ratio 1:2 **(c)** Ratio 1:30

Fig. 9 Program ThreadRadars of test-case runs of Bug 2 with fix showing different producer-consumer ratios

(a) A run of the test case with the bug.

(b) The corresponding list of thread types to (a).

(c) A run of the test case with the fix.

Fig. 10 Program ThreadRadars of test-case runs of Bug 3 and corresponding list of thread types in the detail window

denotes a bug due to inconsistent synchronization. We rewrote this bug so that a correct synchronization is performed but still reader threads were not able to terminate due to a race condition that effects the intended execution order of statements. Altogether, the ThreadRadars make 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 ThreadRadars facilitate 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 Formative usability test

To get formative feedback on our tool implementing the ThreadRadar and to determine whether computer scientists other than the developers would be able to use and benefit from the tool, we conducted a usability test where we asked two PhD students from a different research group at our department to debug two concurrency bugs from the demonstration study with and without the ThreadRadar.

In Table 1, we provide an overview of the sequence of phases and sessions of the usability test. In particular, each participant runs through two phases of subsequent introduction and debugging sessions. At first, we introduce profiling, stack sampling and the adopted performance visualization to the participants. Each introduction includes a demonstration video as well as a small hands-on example. Afterward, a debugging session on a bug of the demonstration study is conducted. For the first debugging session, the participants are equipped with the tool without the ThreadRadar, i.e., solely the adopted performance visualizations are featured. Next, we introduce the thread-aware performance metrics and the ThreadRadar, again with a demonstration video and a hands-on example. Subsequently, a second debugging session with the full tool, i.e., including the ThreadRadar, is conducted on a different bug from the demonstration study. We swap the bugs from the two debugging sessions for the two participants. In this way, we prevent the participant's perception of the usefulness of the ThreadRadar from being potentially attributed to the difficulty of the bug. Also, we test the ThreadRadar on two different bugs which offers a broader potential on feedback as they may require different information and support for debugging. Furthermore, we divided the sessions into two consecutive phases to enable the participants to compare the tool with and without the ThreadRadar as well as to avoid introducing too many concepts and features at once.

In the debugging sessions, the task of the participants was to familiarize with the code and to locate the defect in the code. Furthermore, they were asked to propose and implement a fix to the bug. During the debugging sessions, the participants were encouraged to follow the *Think Aloud* approach to verbalize their thoughts. Additionally, we integrated a logging mechanism into the tool that logs the usage of the tool features such as mouse click events on the ThreadRadar glyphs to open the detail window. Furthermore, the session host was present the whole time to help with technical problems, observe the participants and to take

Table 1 Sequence of sessions of the formative usability test for the two participants

Phase	Session	Participant	
		1	2
I	Introduction	Profiling, stack sampling, adopted performance visualizations	
	Debugging without ThreadRadar	Bug 1	Bug 3
II	Introduction	Thread-aware metrics, ThreadRadar, tool features	
	Debugging with ThreadRadar	Bug 3	Bug 1
III	Structured Group Interview		

further notes. Afterward, we conducted and recorded a structured group interview with both participants to recap their experiences with our tool.

Based on our notes and log files of Phases I and II, we made the following observations:

O1: Participant A actively used the ThreadRadars together with both the overview and the detail window whereas participant B mostly read the program code.

O2: While participant A used the classes and methods tab of the overview window as well as its navigation feature only once, the participant opened the detail window 17 times and therein investigated the different clusters for six classes and methods that are related to the bug.

O3: Both participants did not only find the bug when using the tool with the ThreadRadar, but also fixed it.

O4: In the detail window, the participants tried to navigate to the class of a particular thread in a cluster by clicking at a sector or an item in the list of thread-types, although in this context, clicking was already assigned to select threads for the thread filters.

Next, we summarize the participants' comments from the subsequent group interview:

C1: They both agreed that the ThreadRadars were helpful, in particular to verify their proposed bug fixes.

C2: One participant stated that the coloring (weak and strong saturation) of the sectors was comprehensible and allowed to obtain the required information at first sight.

C3: One participant would have liked to have an absolute runtime value in conjunction with the relative runtime metrics. The participant stated, that it was impossible to assess 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.

C4: 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.

While the usability test provided first indications (O3, C1) that our tool, and with that our approach can support concurrency debugging, it certainly helped us to test its design for future investigations. Most importantly, we got formative feedback.

6.1 Tool improvements

Based on the results of the usability test, we made improvements to our tool. As a consequence of Observation O4, we added an extra tab to the overview window that lists all threads included in the thread filter and allows to navigate to the source code executed by the thread. More importantly, the participants requested more insight in the distribution of the computational runtime of the threads for a given artifact (Comment C4). This information could not only serve the purpose to better decide which cluster to inspect but also to get a feel for different aspects of the applied clustering of the threads. Our overall approach clusters the threads based on statistical-profiling data, which may vary from one run to another, and we may be dealing with non-deterministic behavior. Thus, an important aspect is to evaluate whether the boundaries of each computed thread cluster are appropriate. For instance, the distribution of the data points could indicate a clear division into naturally emerging clusters or it could be diffuse, making the computed clustering appear less reasonable.

To this end, we extended the detail window with a histogram showing the distribution of the artifact-relative computational runtime of the threads. In Fig. 11, we present a histogram for the entire program of a test-case run of Bug 3 with the bug (see Sect. 5.3).

In the histogram, each square represents a different thread. The coloring of the squares matches the color of the corresponding thread cluster. The x-axis of the histogram is linearly scaled and limited by the

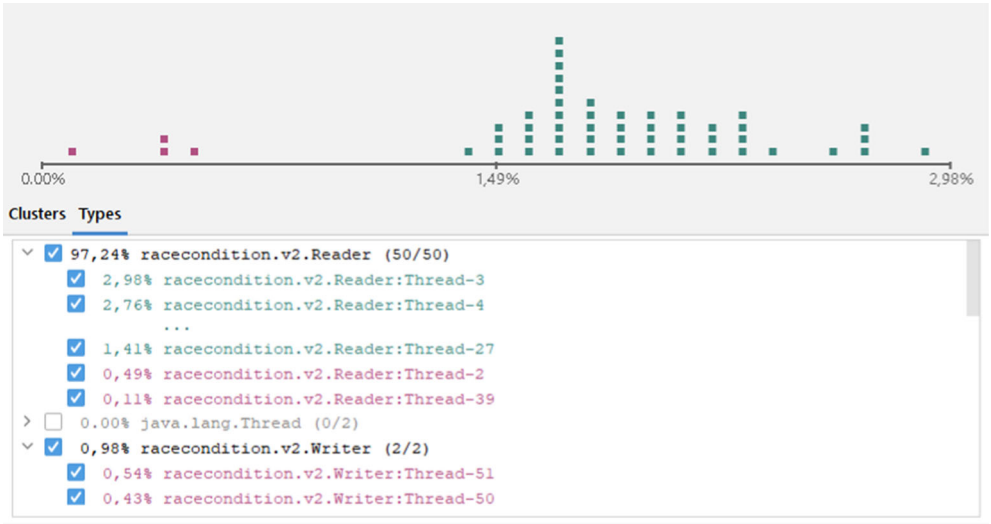


Fig. 11 Histogram for the entire program of the artifact-relative computational runtime of a test case run of Bug 3 with the bug. Here, 50 reader and two writer threads are involved

maximum artifact-relative computational runtime of the threads. The horizontal placement of the squares corresponds to the respective artifact-relative computational runtime of the represented thread. Identical values are vertically stacked. The actual placement of the squares on the y-axis depends on the maximum number of threads to be stacked in the overall histogram. The histogram is robust to resizing the window as the concrete placement of the squares is dependent on the available window width and height. Also, the histogram is updated when changes are made to the thread selection. The clusters and thread-types view is located below the histogram so that the effect of changes in the selection of threads can be observed immediately in the histogram.

In the example of Fig. 11, we can visually assess that the computed clusters are plausible as there is a clear gap between the red colored data points on the left and the green colored data points on the right. Consequently, this clustering appears meaningful. The two reader threads that are grouped together with the two writer threads in the red cluster are the threads that were able to terminate while all other reader threads kept running. Inspections on why some of the reader threads behave differently would therefore be the next logical step in a debugging session.

In contrast to that, the histogram in Fig. 12 shows the distribution of the artifact-relative computational runtime of the threads for the entire program from another test-case run of Bug 3.

We see that there is one thread, namely a reader thread (`Thread-42`), that was assigned to the red cluster which seems inappropriate. Besides the fact that this thread is not one of the threads that was able to terminate, the histogram reveals that the distance to the nearest thread of the red cluster is much bigger than to the nearest thread of the green cluster. The k-means clustering strategy works with centroids, i.e., the smallest distance to the average of a group is the decisive parameter for the assignment of a data point to a group. Thus, the clustering works satisfactorily, and there is only one thread whose classification is questionable in this concrete case. Most importantly, the histogram facilitates the comprehension not only of the clustering but also the program behavior as it particularly helps to prevent from misinterpreting results from single test-case runs.

Altogether, the histogram view renders a valuable extension to the detail window of the ThreadRadar. While the ThreadRadar attracts attention by visualizing aggregated clustering results, the histogram allows a very detailed examination of the clusters. In particular, this fits well into the intended workflow with our overall approach for debugging concurrent programs.

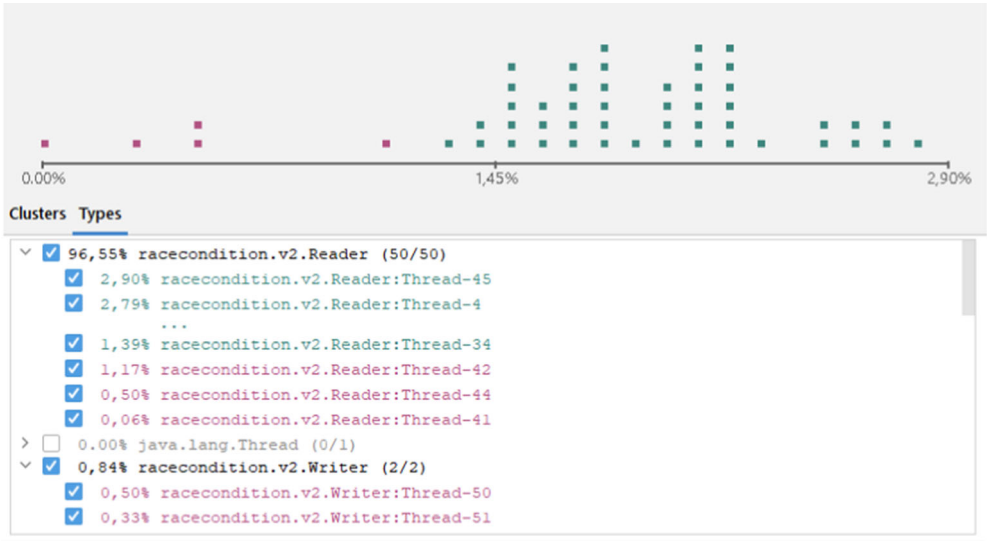


Fig. 12 Histogram for the entire program of the artifact-relative computational runtime of another test case run of Bug 3 with the bug. Here, the assignment of `Thread-42` to the red cluster seems inappropriate

7 Limitations

7.1 Profiling

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.

The use of a statistical profiling, here stack sampling, has well-known drawbacks (Nisbet et al. 2019; Mytkowicz et al. 2010). In our investigations of concurrency bugs, the heuristic data were 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 group interview (Comment C3), assessing if a program runs seconds, minutes, etc., is hard with the use of relative runtimes. To address this, it would be possible to combine statistical profiling with another approach or to approximate the absolute runtime values based on the relative values. The latter would be highly error-prone, as assumptions have to be made about the degree of parallelism. However, the opportunity to additionally provide absolute runtimes in conjunction to the relative runtimes appears promising in cases where exact timing and duration of events is a necessary debug information.

Furthermore, we solely consider threads with state *Runnable* (See Sect. 2). Waiting time on operating system resources such as the processor and I/O is captured by our approach. Thus imbalance issues according to such resources can be investigated. But we ignore the time threads spend on waiting or being blocked on Java monitors and sleeping. However, attempts to (re)enter Java monitors will still be accounted since the corresponding thread will switch state in order to do so.

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.

7.2 Scalability

Long running programs that utilize many classes potentially exhibit a large number of threads and types of threads. We use clustering methods to reduce the amount of data such that it can be visualized and investigated more easily. The current limit of a maximum of three clusters might be too restrictive to describe the work balancing behavior of such program runs and a completely different method to cluster threads might be appropriate. See, for instance, Pauw et al. (2006) where a set of transactions, i.e., essentially stack traces, is partitioned into groups of isomorphic tree structures to extract patterns. In our tool

the thread filter can mitigate the problem of a vast number of threads to some extent. 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.

Another scalability limitation arises from the use of constrained k-means clustering to counter the inaccuracy of statistical profiling. As a result, if the difference of the artifact-relative computational runtimes of two threads is smaller than ϵ (see Sect. 2.3), they are grouped into the same cluster, where ϵ is inversely proportional to the number of threads. As a consequence, if the clustering yields less than three clusters, a more fine-grained clustering computed with a smaller value of ϵ or no constraint at all might be more revealing.

Moreover, there are minor visual scalability concerns. The angles of the sectors that represent clusters of threads are computed relative to the total number of threads sampled for the given artifact. Thus, for example, if one cluster contains a hundred times the number of threads of another cluster, the angle of a sector might become too small to be visible or selectable in the detail window (see Fig. 13).

This issue can be countered with a minimum angle of the sectors. The only disadvantage in this case is that the ratio of the sector sizes does no longer exactly reflect the ratio of the cluster sizes.

Altogether, it demands a sophisticated coordination and tuning of each part of our approach, i.e., profiling, metrics, clustering and visual design to reach a good trade-off between scalability and applicability for performance debugging. In this work, we presented a working balance that is open for modification to other use cases.

7.3 Concurrency bug types

In this work, we concentrated on concurrency bug types such as data races, race conditions and synchronization granularity issues. All of them have in common that they are non-blocking. Another infamous 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. In experiments where 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.

7.4 Formative usability test

The usability test helped to identify usability problems and to find opportunities to improve our tool. While two participants are likely to uncover some of the most frequent issues, they will not necessarily find the most severe issues. We did not measure participant performance in terms of time and accuracy, nor did we compare the debugging process with our tool to a baseline that would allow any conclusion about the efficiency of our approach. Nevertheless, the participants used both the tool with and without the ThreadRadars.

Moreover, we conducted the experiments on source code that is unknown to the participants which stands in contrast to our assumption that the person debugging the code already has a deep understanding of it (see Sect. 2). Thus, the participants had to read code in order to understand its static structure as well as its dynamic behavior. In the group interview, participant B stated that his or her primary task was to read and understand the source code before focusing on the bug. Participant B planned to use the ThreadRadars and

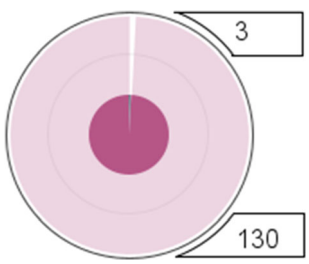


Fig. 13 ThreadRadar with a cluster sector angle that has become too small

tool features afterwards, but while familiarizing with the code he or she found the bug and a possible fix and thus did not investigate the ThreadRadars (Observation O1). Therefore, letting participants use our tool to inspect their own software in a usability test would be more realistic and time efficient because it would exclude the code reading and understanding process from the investigations and prevent from accidentally solving the task without the ThreadRadars. However, it places an additional constraint on the selection of candidates for a study: Participants must have their own concurrent Java programs that, in the best case, also contain a concurrency bug that can be investigated using our approach.

For the ThreadRadar approach, it is important to investigate how experts in the field, i.e., concurrency debugging, perform with and assess our tool compared to a state of the art tool. On the other hand, it could be promising to involve novices to investigate whether our approach lowers the barriers to enter, explore and understand the complex and unfamiliar task of concurrency debugging. We conducted the usability test with two computer science PhD students both of whom indicated that they were not experts in the field of concurrency debugging but were not completely inexperienced with it.

Altogether, the investigations we conducted are only formative in nature and test whether computer scientists are able to use the tool implementing the ThreadRadar approach. However, parts of the design of our usability test can serve as a basis for summative evaluations.

8 Related work

To discuss related work, we grouped it with respect to four different aspects: visualization in source-code editors, visualization of concurrency, tools for concurrency debugging.

8.1 Visualization in source-code editors

In their systematic mapping study on the visual augmentation of source-code editors Sulír et al. (2018) found, that only few work exists which augments source code with data from dynamic analyses, such as profiling. For instance, Beck et al. (2013) 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. (2018) integrated runtime-performance traces into the source-code editor by highlighting source-code elements, such as method calls and loops. 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 (Röthlisberger et al. 2012). Albeit their tool facilitates performance-optimization tasks, no concurrency data is considered. Alcocer et al. (2019) 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.

8.2 Visualization of concurrency

There are many approaches to visualize concurrency based on program execution traces. Karran et al. (2013) introduce *SyncTrace*, a visualization technique combining multiple variants of icicle plots, edge bundling and various interaction possibilities for the analysis of dependencies between threads. UML-based diagrams, in particular UML-sequence diagrams have been employed for visualization of concurrency (Artho et al. 2007; Mehner 2005). Both, the tool *ThreadCity* (Hahn et al. 2015) as well as *SynchroViz* (Waller et al. 2013) use a specialized 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. (2009) utilize clustering to map continuous distributions of dynamic metric values to a discrete six-valued scale. We similarly compute three thread clusters based on relative runtime consumption. Most approaches focus on data of inter-thread correspondence, such as caller/callee relation, blocking and synchronization. Furthermore, Isaacs et al. (2014) cluster processes according to logical time

and time-based metrics such as lateness to represent a large number of processes. In contrast to that, we take advantage of the relative runtime consumption of threads aggregated for source-code artifacts.

8.3 Concurrency debugging

Many visualization tools were proposed to support the debugging of concurrent software such as TIE (Maheswara et al. 2010), CHESS (Musuvathi and Qadeer 2006), JIVE and JOVE (Reiss and Renieris 2005) and DyView (Reiss and Karumuri 2010). All of them make use of separate views, while we directly embed our visualizations into the source code. We only utilize additional views for overview and details on demand.

Many other tools in the context of concurrency debugging focus on automated concurrency-bug detection instead of code comprehension (Savage et al. 1997; Flanagan and Freund 2010; Marino et al. 2009; Erickson et al. 2010). For instance, the tool FALCON (Park et al. 2010) applies a dynamic technique to identify faulty data access patterns in multi-threaded programs. While these tools localize the defects in the source code, our approach focuses on emphasizing the symptoms they yield at runtime.

9 Conclusion

We presented our concurrency debugging approach which especially consists of augmenting the source code with a glyph-based visualizations representing thread information 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 three bugs, in particular two different non-deadlock concurrency bugs and a performance bug. With that, we showcased and discussed the usefulness of the ThreadRadar for program-comprehension tasks in the context of debugging. Through a usability test we gained first formative results for our tool, as well as first evidence that our approach can support the concurrency debugging process. Subsequently, we outlined limitations of various aspects of our approach as well as of the usability test.

Thus, future work includes scalability, other types of bugs, programming languages, and performance metrics in particular, leveraging also waiting and blocking time of threads.

Appendix A Truncated Java code of Bug 1

```
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());
 } }
```

```
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 */} }
```

```
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 thesupplemental material (Moseler et al., 2022).

Author Contributions Oliver Moseler, Lucas Kreber, Stephan Diehl contributed to conceptualization; Oliver Moseler, Stephan Diehl contributed to methodology; Oliver Moseler contributed to investigation; Oliver Moseler contributed to data curation; Oliver Moseler contributed to writing—original draft preparation; Oliver Moseler, Stephan Diehl contributed to writing—review and editing; Oliver Moseler, Stephan Diehl contributed to visualization; Stephan Diehl helped in supervision; Oliver Moseler, Lucas Kreber helped in software.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

Alam MMU, Liu T, Zeng G, et al. (2017) Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In: Proceedings of the 12th European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017, pp 298–313, https://doi.org/10.1145/3064176.3064186

Alcocer JPS, Jaimes HC, Costa D, et al. (2019) Enhancing commit graphs with visual runtime clues. In: proceedings of working conference on software visualization, VISSOFT 2019, Cleveland, OH, USA, September 30 - October 1, 2019. IEEE, pp 28–32, https://doi.org/10.1109/VISSOFT.2019.00012

Artho C, Havelund K, Honiden S (2007) Visualization of concurrent program executions. In: proceedings of 31st annual international computer software and applications Conference, COMPSAC 2007, Beijing, China, July 24-27, 2007. Volume 2, pp 541–546, https://doi.org/10.1109/COMPSAC.2007.236

Beck F, Moseler O, Diehl S, et al. (2013) In situ understanding of performance bottlenecks through visually augmented code. In: proceedings of IEEE 21st international conference on program comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013, pp 63–72, https://doi.org/10.1109/ICPC.2013.6613834

Borgo R, Kehrer J, Chung DHS, et al. (2013) Glyph-based visualization: Foundations, design guidelines, techniques and applications. In: Sbert M, Szirmay-Kalos L (eds) 34th Annual Conference of the European Association for Computer Graphics, Eurographics 2013 - State of the Art Reports. Eurographics Association, pp 39–63

Cito J, Leitner P, Bosshard C, et al. (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, pp 41–44, https://doi.org/10.1145/3183440.3183481

Cornelissen B, Zaidman A, van Deursen A et al (2009) A systematic survey of program comprehension through dynamic analysis. IEEE Trans Software Eng 35(5):684–702. https://doi.org/10.1109/TSE.2009.28

Do H, Elbaum SG, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir Softw Eng 10(4):405–435. https://doi.org/10.1007/s10664-005-3861-2

Erickson J, Musuvathi M, Burckhardt S, et al. (2010) Effective data-race detection for the kernel. In: Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings, pp 151–162, http://www.usenix.org/events/osdi10/tech/full_papers/Erickson.pdf

Flanagan C, Freund SN (2010) Fasttrack: efficient and precise dynamic race detection. Commun ACM 53(11):93–101. https://doi.org/10.1145/1839676.1839699

Hahn S, Trapp M, Wuttke N, et al. (2015) Thread City: Combined visualization of structure and activity for the exploration of multi-threaded software systems. In: Proceedings of 19th International Conference on Information Visualisation, IV 2015, Barcelona, Spain, July 22-24, 2015, pp 101–106, https://doi.org/10.1109/iV.2015.28

Hallal H, Alikacem EH, Tunney WP, et al. (2004) Antipattern-based detection of deficiencies in Java multithreaded software. In: Proceedings of 4th International Conference on Quality Software (QSIC 2004), 8-10 September 2004, Braunschweig, Germany, pp 258–267, https://doi.org/10.1109/QSIC.2004.1357968

Isaacs KE, Bremer PT, Jusufi I et al (2014) Combing the communication hairball: Visualizing parallel execution traces using logical time. IEEE Trans Visual Comput Graphics 20(12):2349–2358. https://doi.org/10.1109/TVCG.2014.2346456

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. Accessed January 2020

JetBrains s.r.o. (2021) All developer tools and products by JetBrains. https://www.jetbrains.com/products.html#type=ide. Accessed May 2021

Jin G, Song L, Shi X, et al. (2012) Understanding and detecting real-world performance bugs. In: Proceedings of ACM SIGPLAN conference on programming language design and implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, pp 77–88, https://doi.org/10.1145/2254064.2254075

Karran B, Trümper J, Döllner J (2013) SYNCTRACE: visual thread-interplay analysis. In: Proceedings of 2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013, pp 1–10, https://doi.org/10.1109/VISSOFT.2013.6650534

Lin Y, Dig D (2015) A study and toolkit of CHECK-THEN-ACT idioms of Java concurrent collections. J Soft: Testing, Verification Reliability 25(4):397–425. https://doi.org/10.1002/stvr.1567

Lin Z, Marinov D, Zhong H, et al. (2015) JaConTeBe: A benchmark suite of real-world Java concurrency bugs (T). In: Cohen MB, Grunske L, Whalen M (eds) Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. IEEE Computer Society, pp 178–189, https://doi.org/10.1109/ASE.2015.87

Lu S, Park S, Seo E, et al. (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, pp 329–339, https://doi.org/10.1145/1346281.1346323

Maheswara G, Bradbury JS, Collins C (2010) TIE: an interactive visualization of thread interleavings. In: Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010, pp 215–216, https://doi.org/10.1145/1879211.1879247

Marino D, Musuvathi M, Narayanasamy S (2009) Literace: effective sampling for lightweight data-race detection. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009, pp 134–143, https://doi.org/10.1145/1542476.1542491

Mehner K (2005) Trace based debugging and visualisation of concurrent Java programs with UML. PhD thesis, University of Paderborn, Germany, http://ubdata.uni-paderborn.de/ediss/17/2005/mehner/disserta.pdf

Moseler O, Kreber L, Diehl S (2021) Threadradar: A thread-aware visualization for debugging concurrent java programs. In: Proceedings of the 14th International Symposium on Visual Information Communication and Interaction (VINCI). Association for Computing Machinery, New York, NY, USA, https://doi.org/10.1145/3481549.3481566

Moseler O, Kreber L, Diehl S (2022) Supplemental Material to: The ThreadRadar Visualization for Debugging Concurrent Java Programs. Zenodo. https://doi.org/10.5281/zenodo.5940561
Musuvathi M, Qadeer S (2006) CHESS: systematic stress testing of concurrent software. In: Proceedings of 16th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers, pp 15–16, https://doi.org/10.1007/978-3-540-71410-1_2
Mytkowicz T, Diwan A, Hauswirth M et al (2010) Evaluating the accuracy of java profilers. SIGPLAN Notices 45(6):187–197. https://doi.org/10.1145/1809028.1806618
Nisbet A, Nobre NM, Riley GD, et al. (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, pp 119–126, https://doi.org/10.1145/3297663.3309677
Nistor A, Song L, Marinov D, et al. (2013) Toddler: detecting performance problems via similar memory-access patterns. In: Proceedings of 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pp 562–571, https://doi.org/10.1109/ICSE.2013.6606602
Oracle (2020a) ArrayBlockingQueue (Java SE 11 & JDK 11). https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ArrayBlockingQueue.html. Accessed January 2020
Oracle (2020b) JVM(TM) Tool Interface 11.0.0. https://docs.oracle.com/en/java/javase/11/docs/specs/jvmti.html. Accessed January 2020
Oracle (2020c) Thread.State (Java SE 11 & JDK 11). https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.State.html#RUNNABLE. Accessed October 2020
Park S, Vuduc RW, Harrold MJ (2010) Falcon: fault localization in concurrent programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, pp 245–254, https://doi.org/10.1145/1806799.1806838
Pauw WD, Krasikov S, Morar JF (2006) Execution patterns for visualizing web services. In: Proceedings of the ACM 2006 Symposium on Software Visualization, Brighton, UK, September 4-5, 2006. ACM, pp 37–45, https://doi.org/10.1145/1148493.1148499
Pinto G, Torres W, Fernandes B et al (2015) A large-scale study on the usage of Java's concurrent programming constructs. J Syst Softw 106:59–81. https://doi.org/10.1016/j.jss.2015.04.064
Reiss SP, Karumuri S (2010) Visualizing threads, transactions and tasks. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010, pp 9–16, https://doi.org/10.1145/1806672.1806675
Reiss SP, Renieris M (2005) Demonstration of JIVE and JOVE: Java as it happens. In: Proceedings of 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, pp 662–663, https://doi.org/10.1145/1062455.1062597
Röthlisberger D, Harry M, Villazón A, et al. (2009) Augmenting static source views in ides with dynamic metrics. In: Proceedings of 25th IEEE International Conference on Software Maintenance (ICSM) 2009), September 20-26, 2009, Edmonton, Alberta, Canada. IEEE Computer Society, pp 253–262, https://doi.org/10.1109/ICSM.2009.5306302
Röthlisberger D, Harry M, Binder W et al (2012) Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. IEEE Trans Software Eng 38(3):579–591. https://doi.org/10.1109/TSE.2011.42
Savage S, Burrows M, Nelson G et al (1997) Eraser: a dynamic data race detector for multithreaded programs. ACM Trans Comput Syst 15(4):391–411. https://doi.org/10.1145/265924.265927
Sulír M, Bacíková M, Chodarev S et al (2018) Visual augmentation of source code editors: a systematic mapping study. J Vis Lang Comput 49:46–59. https://doi.org/10.1016/j.jvlc.2018.10.001
Sutter H, Larus JR (2005) Software and the concurrency revolution. ACM Queue 3(7):54–62. https://doi.org/10.1145/1095408.1095421
Tufte ER (2006) Beautiful Evidence. Graphics Press, Cheshire, CT
Wael MD, Marr S, Cutsem TV (2014) Fork/join parallelism in the wild: documenting patterns and anti-patterns in Java programs using the fork/join framework. In: Proceedings of 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, pp 39–50, https://doi.org/10.1145/2647508.2647511
Wagstaff K, Cardie C, Rogers S, et al. (2001) Constrained k-means clustering with background knowledge. In: Proceedings of the 18th International Conference on Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ICML '01, p 577–584
Waller J, Wulf C, Fittkau F, et al. (2013) Synchrovis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In: Proceedings of First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands, September 27-28, 2013, pp 1–4, https://doi.org/10.1109/VISSOFT.2013.6650520