Detection of Suspicious Time Windows in Memory Monitoring

Markus Weninger
Institute for System Software
CD Laboratory MEVSS
Johannes Kepler University
Linz, Austria
markus.weninger@jku.at

Elias Gander
CD Laboratory MEVSS
Johannes Kepler University
Linz, Austria
elias.gander@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract

Modern memory monitoring tools do not only offer analyses at a single point in time, but also offer features to analyze the memory evolution over time. These features provide more detailed insights into an application's behavior, yet they also make the tools more complex and harder to use.

Analyses over time are typically performed on certain time windows within which the application behaves abnormally. Such suspicious time windows first have to be detected by the users, which is a non-trivial task, especially for novice users that have no experience in memory monitoring.

In this paper, we present algorithms to automatically detect suspicious time windows that exhibit (1) continuous memory growth, (2) high GC utilization, or (3) high memory churn. For each of these problems we also discuss its root causes and implications.

To show the feasibility of our detection techniques, we integrated them into AntTracks, a memory monitoring tool developed by us. Throughout the paper, we present their usage on various problems and real-world applications.

CCS Concepts • **Software and its engineering → Software defect analysis**; *Software performance*; • **Mathematics of computing** → Time series analysis.

Keywords Memory Monitoring, Automatic Time Window Detection, Memory Leak Analysis, Memory Churn Analysis

ACM Reference Format:
Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Detection of Suspicious Time Windows in Memory Monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357390.3361025

MPLR '19, October 21–22, 2019, Athens, Greece
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19), October 21–22, 2019, Athens, Greece*, https://doi.org/10.1145/3357390.3361025.

1 Introduction

Modern programming languages such as Java use automatic garbage collection. Heap objects that are no longer reachable from so-called *GC roots* (e.g., from static fields or thread-local variables) are automatically reclaimed by a garbage collector (GC). Nevertheless, memory problems can still occur even in garbage-collected languages.

A *memory leak* [13] occurs if objects that are no longer needed remain reachable from GC roots due to programming errors. This leads to a *continuously growing memory consumption* which can cause the application to run out of memory, crashing it in the worst case [19].

Even though modern garbage collectors execute certain garbage-collection-related operations concurrently to the application [8, 12, 18], many garbage collection algorithms require *stop-the-world* pauses, i.e., the application is halted while the GC is running. Such GC phases can make up a significant portion of the application's run time.

A *high memory churn rate* stems from frequent unnecessary creation and collection of objects, also known as excessive dynamic allocations [40–42]. This leads to increased work for allocating these objects on the heap and an increased number of garbage collections, which can have a negative impact on an application's performance.

Such memory anomalies manifest themselves in various ways. They lead to different patterns in metrics such as memory consumption, GC frequency, or GC time. Inspecting and interpreting visualizations of these metrics, either in a tabular form or as time-series charts, can be hard for users, especially if they do not have a background in memory analysis.

The aim of this work is to ease the use of memory monitoring tools for novice users. To do so, we free users from the task of searching for different types of suspicious time windows by providing algorithms that automatically detect them. Thus, our contributions are:

1. different algorithms and heuristics to automatically detect suspicious time windows with
   a. continuous memory growth (see Section 3.2).
   b. high GC utilization (see Section 3.3).

   c. high memory churn (see Section 3.4).
   including discussions on the root causes and the implications of the different types of memory anomalies.
2. a working implementation of our approach in the offline memory monitoring tool AntTracks Analyzer.

2 Background

AntTracks consists of two parts: The AntTracks VM [23–25], a virtual machine based on the Java Hotspot VM [47], and the AntTracks Analyzer [2, 51–56], a trace-based memory analysis tool. Since the techniques presented in this paper have been integrated into AntTracks, this section discusses memory traces and how AntTracks uses them.

2.1 Memory Snapshots versus Memory Traces

Many state-of-the-art tools use memory snapshots, i.e., heap dumps, for memory analysis, whereas AntTracks uses memory traces. While heap dumps may be sufficient for heap state analysis at a single point in time, it has been shown that they are not well suited for memory analysis over time [51]. This has various reasons. One of them is that heap dumps do not preserve object identities, i.e., one cannot distinguish whether two objects in two different heap dumps are the same or not.

Trace-based approaches try to circumvent the shortcomings of snapshots by continuously recording information while an application runs. Beside typical *memory traces* [5, 15, 16, 34, 35, 58, 61] that encode memory- and GC-related events such object allocations, object deaths, or object field accesses, there also also other trace types such as *execution traces* that rather focus on call hierarchy information [6, 17, 46].

2.2 Trace Recording by the AntTracks VM

The AntTracks VM records memory events such as object allocations and object movements during GC by writing them into trace files [24]. Trace recording introduces a low run-time overhead of about 5%. Information about GC roots and the references between objects can also be added to the trace [23, 53]. To reduce the trace size, the AntTracks VM does not record any redundant data and applies compression [25].

2.3 AntTracks Analyzer

2.3.1 Reconstruction

The AntTracks Analyzer is able to parse a trace file by incrementally processing its events, which enables it to reconstruct the heap state for every garbage collection point [2]. A heap state is the set of heap objects that were live in the monitored application at a certain point in time. For every heap object, a number of properties can be reconstructed, including its address, its type, its allocation site, the heap objects it references, and the heap objects it is referenced by.

2.3.2 Analysis

The AntTracks Analyzer's core mechanism is object classification and multi-level grouping [54, 56] in which heap objects can be grouped according to certain criteria such as type, allocation site, allocating thread, and so on.

Various techniques to analyze the memory evolution over time have been presented in the past. For example, the approach described in [51, 52] detects the common problem of data structure growth. Handled incorrectly, data structures are often the root cause of memory leaks. In [55], we present an analysis technique that visualizes how the heap composition (i.e., the heap classified by a given object classifier combination) develops over time. All these approaches rely on a previously selected time window. By detecting suitable time windows automatically, such analysis techniques may be easier to apply for novice users.

3 Approach

This section explains how we support users to detect memory anomalies in an application's memory behavior. We present three different time window types, discuss their root causes and implications, and show heuristics and algorithms to automatically detect them. For each time window type, we also show an example on how AntTracks detects a suspicious time window in a real-world application and discuss how this window covers the problem's root cause.

3.1 Desired Window Characteristics

An ideal time window would outline just that portion of the program that should be investigated to find and remove the root cause of the underlying problem. Thus, we define characteristics that a detected time windows should exhibit.

Size Constraints First, detected time windows are desired to be *short* since one or more analysis techniques will be applied on the selected time window. The run time of most of these techniques depends on the number of garbage collections covered by the window. Despite this, windows should also cover a minimum number of garbage collections to prevent them from being only short, less important outliers.

Relevance The detected time windows should cover allocations and objects related to the underlying problem and as little noise, i.e., allocation and objects not related to the problem, as possible. If a window contained noise, e.g., allocations that are not relevant to a memory leak, the noise will also distort the results of analyses applied on the window. This makes it harder to reveal the root cause of the problem.

For example, a memory leak might manifest itself only after a certain point in time. Before that point, fluctuations in the memory can happen due to various reasons such as initialization procedures. These fluctuations are irrelevant for memory leak analysis, i.e., noise, and should be excluded from the detected time window.

Maximum Intensity Problems such as high garbage collection overhead generally do not persist throughout the whole application, but rather occur as *hotspots*. A detection algorithm should find the window that covers the most intense hotspot, e.g., the window with the highest overall garbage collection overhead.

Severity Since the aim of this work is to support novice users by automatically detecting *suspicious* time windows, every detection algorithm has to define thresholds to decide whether a detected time window is indeed suspicious. Windows that are not considered to be suspicious should not be presented to the users. For example, a detected garbage collection overhead hotspot may only be considered suspicious if the garbage collection overhead over the window exceeds a certain threshold, e.g., 10%.

3.2 Memory Leak Window

If a Java application contains a memory leak, certain objects are unintentionally kept alive, causing them to accumulate over time even though they are no longer needed. Consequently, the occupied heap space grows until the application runs out of memory, causing it to crash.

Unfortunately, such a growth trend may be difficult to recognize in a long running application's memory evolution, especially for novice users. Section 3.2.1 discusses the reasons for this in more detail. Thus, we present two algorithms to automatically detect time windows with suspicious growth trends, freeing the user from this task.

3.2.1 Trace Preprocessing

Detecting a memory-leak-induced growth trend based on the occupied heap memory can be difficult. First, the growth might be slow and only significant after the application has run for a long time. Additionally, the occupied memory fluctuates due to garbage collections, which makes it harder to see a clear trend. Finally, growth trends can be masked by *floating garbage*, that is, objects that are no longer reachable but have not been garbage collected yet. Thus, our approach detects growth trends based on the *reachable memory*, that is, the part of the heap memory that is reachable from *GC roots*. The reachable memory is unaffected by garbage collections and free from floating garbage which makes it the ideal basis to detect memory-leak-induced growth trends.

To calculate the reachable memory of a certain heap state, we start at the GC roots. By following all references recursively, we find all live objects on the heap. Summing up their sizes results in the amount of reachable memory, i.e., the memory in the heap that is alive.

This reachable memory calculation happens during trace file parsing, i.e., when the trace file is read for the first time. Calculating the reachable memory for every reconstructed heap state can slow down this parsing process. If performance is of concern to the user, AntTracks allows them to enable sampled reachable memory calculation, i.e., to calculate the reachable memory only for certain heap states. When sampling is enabled, by default the reachable memory is calculated only for every second reconstructed heap state, i.e., at every other garbage collection, roughly cutting the time spent on reachable memory calculation in half. In our experience, this sampling frequency works well with the algorithms presented in Section 3.2.2. Nevertheless, users can adjust the sampling frequency to either reduce the parsing time or to increase the precision of the resulting reachable memory trend.

3.2.2 Automatic Time Window Detection

In the following, we present two algorithms to detect a time window with a growth trend in reachable memory. Both algorithms operate on a time series of reachable memory that was collected according to the preprocessing steps defined in Section 3.2.1.

Linear-regression-based Algorithm

This algorithm starts with an initial window that (1) includes the end of the application and (2) covers the last 10% of all garbage collection points. It performs a linear regression [30] over all reachable memory data points covered by the window and stores the slope of the linear regression line together with the current time window.

Next, the window is expanded to cover one more data point. Again, the algorithm performs a linear regression and stores the slope together with the time window. It continues in this way until the last time window, ranging from the application's start to the application's end, has been handled. Among all the stored windows, the one with the greatest regression line slope is chosen as the resulting time window. Finally, we also require that its regression slope is positive.

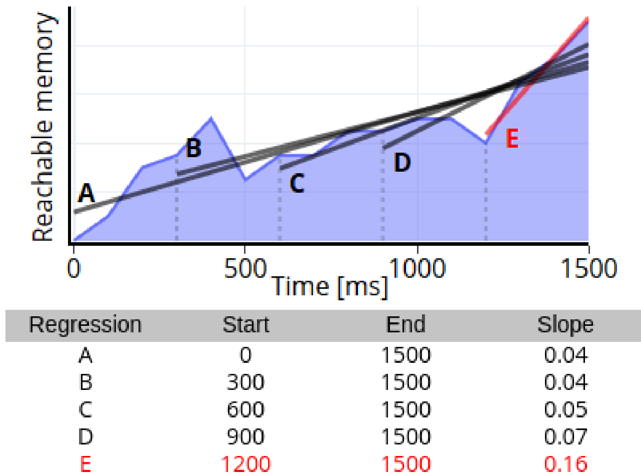
Figure 1 illustrates this algorithm. The plot shows 5 of the regression lines that would be calculated in the course of the algorithm. The regression line E has the greatest slope. Thus, the algorithm would return the window ranging from 1200 ms to 1500 ms. Note that due to limited space, we did not plot all regression lines.

Heuristic-based Algorithm

While the linear-regression-based algorithm is straightforward and easy to implement, the windows it detects do not always fulfill the criteria we defined in Section 3.1. This problem will be discussed in more detail in Section 3.2.3.

Considering the shortcomings of the linear-regression-based algorithm, we introduced a second time window detection algorithm. This algorithm mimics the way a human would search for a continuous memory growth: Find the longest time window over which the memory grows more-or-less continuously, allowing minor drops.

From the time series of reachable memory data points, the algorithm extracts the longest time window in which (1) the end of the application is included and (2) every data



Regression	Start	End	Slope
A	0	1500	0.04
B	300	1500	0.04
C	600	1500	0.05
D	900	1500	0.07
E	1200	1500	0.16

Figure 1. The linear-regression-based algorithm chooses the time windows with the steepest regression slope (*E*).

point within the window fulfills the growth condition. A data point fulfills the growth condition if its reachable memory is greater than that of the previous one. If this is not the case, a data point may still fulfill the growth condition if its reachable memory (1) is greater than the reachable memory of the window's first data point and (2) is at least 75% of the maximum reachable memory of all previous data points in the window. This heuristic requires a detected window to have an overall positive growth in reachable memory, but tolerates smaller drops.

The detection algorithm consists of the following steps. It starts with a window spanning only the application's first reachable memory data point. Next, it takes the second data point and tests whether it fulfills the growth condition with respect to the first point. If this is the case, the window is expanded up to this point. Otherwise, the current window is discarded and the algorithm starts again with a window spanning only the second data point. The algorithm continues in this way until the application's last data point has been handled. Finally, to make sure that the detected time window is long enough to be meaningful, the algorithm checks if the time window covers at least 10% of all garbage collections.

Figure 2 illustrates this algorithm. The initial time window starts at 0 ms and is expanded up to the first drop. Here the reachable memory drops by 50%. Thus, the current window is discarded and a new time window starts after the drop. From there on, the window can be expanded up to the end of the application because the second encountered drop is not strong enough.

Narrowing the Time Window In many applications, the reachable memory growth is not equally strong over the whole detected window. In such cases it is often possible to find a shorter subwindow with a higher growth rate. As stated in Section 3.1, a short time window is desired since subsequent analyses take less time to complete. Additionally,

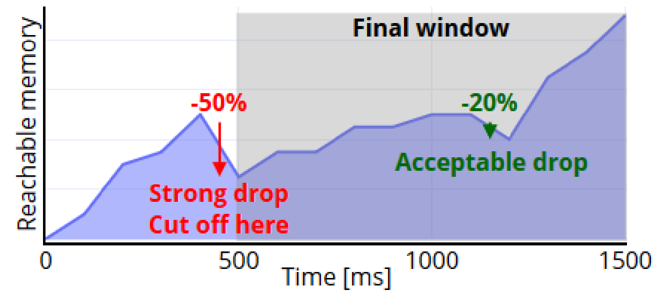


Figure 2. The heuristic-based algorithm detects the longest window without strong drops in the reachable memory.

if the growth over the shorter subwindow is caused by the memory leak, the problematic objects will stand out even more during the analysis. Consequently, the causes of the memory leak will be easier to recognize. Nevertheless, there is a chance that the strong growth covered by the shorter subwindow is actually unrelated to the memory leak, which instead manifests itself only in the slow and steady growth over the full time window. Thus, in AntTracks we decided to present both of these windows to the users. We leave it to their choice whether they want to perform a quick inspection of the strongest growth subwindow first.

When calculating a subwindow, we determine its minimum and maximum size. We define the minimum size as 10% of all reachable memory data points in the long window and the maximum size as 50% of all reachable memory data points in the long window. In any case, the minimum size must be at least two data points.

The algorithm takes the first data point in the long window as a starting point and builds all possible windows that (1) start at this data point, (2) end at another data point and (3) meet the size constraints. For all these windows, it then calculates the reachable memory growth per second and chooses the one with the quickest growth. This time window is remembered and the process is repeated with the next data point as starting point. This is repeated until all data points in the long window have been used as starting point. As a result, the algorithm remembered one window for each data point. Among all these windows, it again chooses the one with the quickest growth.

3.2.3 Examples

As already mentioned, the heuristic-based algorithm has been developed to overcome the flaws of the linear-regression-based algorithm which does not always fulfill the desired characteristics defined in Section 3.1. Figure 3 illustrates this problem. While both algorithms detect the same time window in the first two examples on the top half, in the third example the linear-regression-based algorithm includes a presumably irrelevant spike. The linear-regression-based algorithm performs even worse in the three examples on the bottom half of Figure 3. In these examples, it includes the

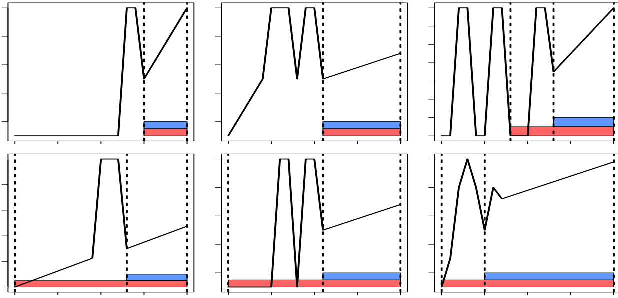


Figure 3. Six exemplary memory evolutions and the detected time windows (red = linear regression, blue = heuristic).

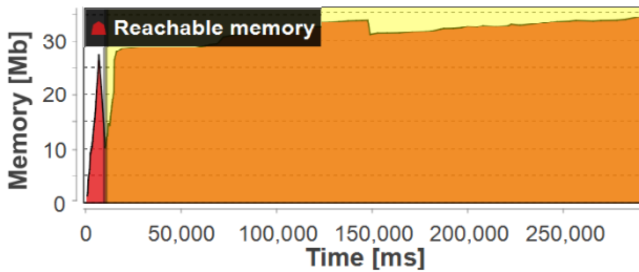


Figure 4. EasyTravel's memory evolution and the memory leak time window (yellow) detected by the heuristic-based algorithm.

whole application in the detected time window instead of just the final period of suspicious growth. Depending on the analysis technique that should be applied on the time window, the noise included in these time windows may make it difficult to recognize the root cause of the suspicious growth.

3.2.4 Case Study: Dynatrace EasyTravel

To show how our approach can be used in AntTracks, we apply it on the *Dynatrace easyTravel* application [11]. Dynatrace focuses on application performance monitoring (APM) and distributes easyTravel as their state-of-the-art demo application. It is a multi-tier application for a travel agency, using a Java backend. An automatic load generator can simulate accesses to the service. When easyTravel is started, different problem patterns can be enabled and disabled, one of which is a hidden memory leak somewhere in the backend.

Figure 4 shows the memory evolution of the application and the time window that has been detected automatically using the heuristic-based memory leak time window detection algorithm. After an initial peak (which is not included in the final time window), the memory mostly grows, except for a drop at around $150,000ms$, which is small enough to be tolerated by the algorithm.

This time window can be inspected with different analysis techniques. For example, the time window could be checked for growing data structures, as done in [51]. The less noise the window contains, the easier it becomes for users to spot those data structures that are involved in the memory leak.

3.3 High GC Overhead Analysis

The *garbage collection overhead* is the ratio between the time spent on garbage collections and the application's overall run time. The duration of a garbage collection partly depends on the number of surviving objects. The more objects survive, the more moves have to be executed by the garbage collector. This leads to increased garbage collection times.

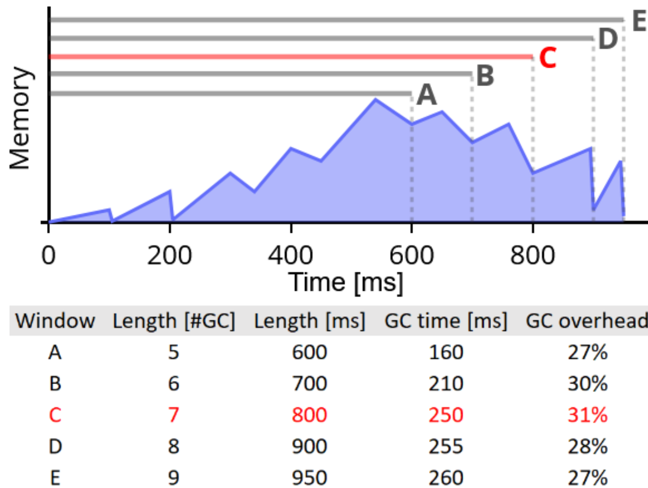
To reduce an application's garbage collection overhead, users should inspect the time window that exhibits the highest GC overhead. In this time window, analysis techniques to identify those objects that survive and thus slow down the collections could be applied. Yet, according to our experience, most novice users disregard problematic garbage collector behavior and only focus on the memory evolution when inspecting an application. Thus, we support them by detecting the time window with the highest GC overhead automatically.

3.3.1 Time Window Detection

On the other hand, windows that cover a very large number of garbage collections might take too long to analyze. They also do not provide more insight because a shorter window will still reveal the reasons for the high garbage collection overhead. Thus, algorithms should only looks for windows that cover at least 5 garbage collections and do not cover more than 50 garbage collections. The numbers used as smallest and largest window size have proven to work well in most scenarios.

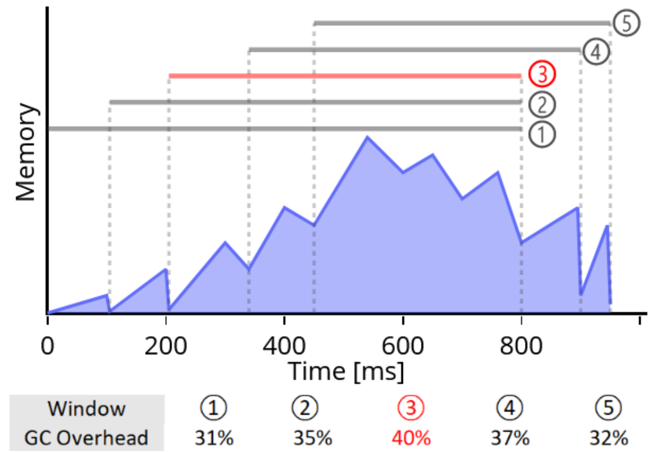
We assume that we know the start time and the end time of each garbage collection in the application. To find the window with the highest garbage collection overhead that meets the window size constraint, our algorithm performs the following steps: First, it selects a start timestamp for the time window. In the first iteration, the start timestamp is the trace's very first timestamp, i.e., the timestamp that marks the start of the application. It then builds all windows that (1) start at this timestamp, (2) meet the size constraint, and (3) end at the end of a garbage collection. Figure 5 demonstrates the first iteration of the algorithm where windows A to E are built from the initial timestamp. For each of these windows, the garbage collection overhead is calculated by dividing the time spent in garbage collections over the window by the duration of the window. Among all windows constructed this way, the algorithm remembers the one with the highest garbage collection overhead. In the example in Figure 5, this would be window C.

In the following iterations, the start timestamp moves forward such that every end of a garbage collection serves as start timestamp once. For every start timestamp then again all valid windows are built and the one with the highest garbage collection overhead is remembered. Figure 6 shows the remembered window for each start timestamp (windows ① to ⑤). Among these windows, the algorithm



Window	Length [#GC]	Length [ms]	GC time [ms]	GC overhead
A	5	600	160	27%
B	6	700	210	30%
C	7	800	250	31%
D	8	900	255	28%
E	9	950	260	27%

Figure 5. Windows A-E are all the valid windows that start at the first timestamp. Window C is the one with the highest garbage collection overhead.



Window	①	②	③	④	⑤
GC Overhead	31%	35%	40%	37%	32%

Figure 6. Windows ①-⑤ are the windows with the highest garbage collection overhead for each start timestamp. Window ③ is the one with the highest overhead overall.

finally chooses the one with the highest overhead. This final window has the highest overhead of all possible windows that meet the size constraint. In the example in Figure 6, this would be window ③.

The final window is only accepted if it has a overhead of at least 10%. This threshold prevents us from detecting a window with a generally low garbage collection overhead.

3.3.2 Case Study: AntTracks

We applied AntTrack's garbage collection overhead window detection mechanism on AntTracks itself. As shown in Figure 7, it was able to detect the time window with the highest garbage collection overhead that meets the window size constraints. The time window covers the most intense part of a garbage collection overhead hotspot. By analyzing which objects had to be moved by the GC most often during this window, we were able to find and fix a bottleneck involving `long[]` objects that were created in AntTracks when pointer information was read from trace files.

3.4 High Memory Churn Analysis

As we have seen before, garbage collections are slower the more objects survive. Analogously, they are fast when many objects die. Yet, even these fast garbage collections have to pause the application. These *stop-the-world pauses* require all application threads to halt at so-called safepoints, i.e., at specific instructions that block the executing thread if necessary, before the GC can start to work.

Even though modern garbage collectors such as Shenandoah [12] or the Z GC [48, 49] perform certain operations concurrently to the running application, nearly all garbage collectors still have to use stop-the-world pauses at some points. When many garbage collections occur over a short time span, these pauses can lead to a significant overhead. A typical cause for frequent garbage collections are objects that are allocated in great numbers and turn into garbage shortly after their allocation.

In the next section, we present an algorithm that detects suspicious time windows based on an application's *memory churn rate*. The memory churn rate is the frequency at which the application discards memory within a certain time window. This hints at a wasteful use of objects, i.e., an unnecessarily high amount of short-living object allocations. Often, algorithms can be adjusted to use fewer temporary objects, which leads to two improvements: (1) Less time is spent for the allocation of objects, and (2) the number of garbage collections is reduced. A common case for the excessive use of temporary objects in Java is the boxing of primitives.

3.4.1 Time Window Detection

To detect the time window with the highest memory churn rate we basically use the algorithm described in Section 3.3.1. The only difference is that this time the algorithm searches for the window with the highest memory churn rate instead

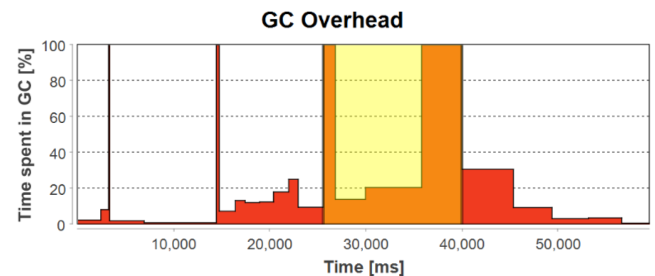


Figure 7. Automatically detected GC overhead window in AntTracks, highlighted in yellow from around 26, 000ms to 40, 000ms.

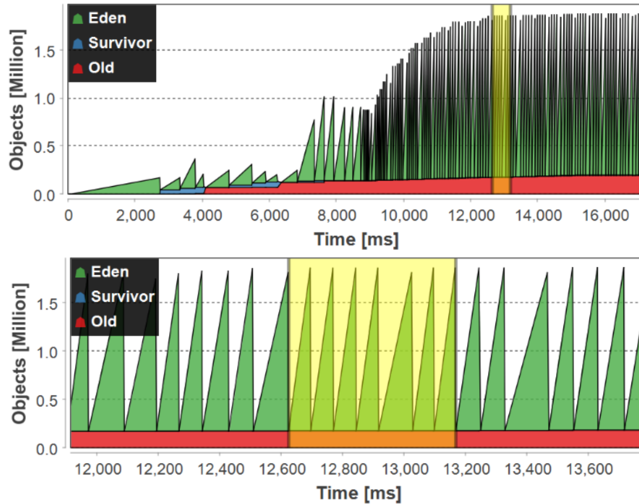


Figure 8. Automatically detected time window with high memory churn in the `finagle-http` benchmark (global view and zoomed-in view).

```
1 val response: Future[http.Response] = client(request)
2 for (i <- 0 until NUM_REQUESTS) {
3   Await.result(response.onSuccess { rep: http.Response =>
4     totalLength += rep.content.length })}
```

Listing 1. Problematic part of the method `FinagleHttp.runIteration`.

of the highest garbage collection overhead. To do so, it calculates the churn rate of a window by dividing the total number of bytes freed within the window by the window's duration. To calculate the number of freed bytes for a given garbage collection, all we need to know is the size of the heap before the collection and after the collection.

3.4.2 Example: Finagle-http

Renaissance [33] is a benchmark suite composed of modern, real-world, concurrent, and object-oriented workloads that exercise various concurrency primitives of the JVM. Since this benchmark suite is rather new, it has not yet been the subject of a memory study [26]. Thus, it is perfectly suited to test whether AntTracks is able find memory problems in real-world applications unknown to the inspector.

First, we downloaded the benchmark suite [1] in version 0.9 and created a trace file of every benchmark. Then, we loaded these trace files into AntTracks and inspected the automatically detected time windows. One benchmark that attracted our attention was `finagle-http`. According to the benchmark's documentation, it *sends many small Finagle HTTP requests to a Finagle HTTP server and awaits response.* This benchmark exhibits a high memory churn. Its memory evolution and the automatically detected memory churn window can be seen in Figure 8.

[1]Renaissance benchmark suite: https://renaissance.dev/

```
1 val response: Future[http.Response] = client(request)
2 val h = { rep: http.Response => totalLength += rep.content.length }
3 for (i <- 0 until NUM_REQUESTS) Await.result(response.onSuccess(h))
```

Listing 2. Fixed version of the method `FinagleHttp.runIteration`.

Inspecting this memory churn using AntTracks's short-lived objects analysis feature revealed that the type `FinagleHttp$$anonfun$runIteration$1$$...` has a high churn rate. The naming pattern reveals that these are Scala objects, more specifically, anonymous functions, which are allocated in the method `runIteration` of the benchmark's main class `FinagleHttp`. Since such a rapid allocation and collection of anonymous functions is unlikely to be intentional, we looked up the method's source code. The problematic part can be seen in Listing 1. In the loop, a lot of anonymous function objects are created, waiting for an HTTP request to succeed to increment the counter `totalLength`. Listing 2 shows our fix for this problem. Only a single response handler is created which is reused for every HTTP request. This fix reduces the overall amount of allocated temporary objects by about 25%.

3.5 Window Detection Performance

The complexity of all algorithms is $O(n)$, where n is the number of garbage collections covered by the trace file. For example, applied on the trace of the Dynatrace EasyTravel application that has been shown in Section 3.2.4, which covers about 700 garbage collections, the different time window detection algorithms take between 5ms and 30ms on average. Thus, our algorithms can scale up to process data reconstructed from traces that cover thousands of garbage collections.

4 Related Work

Memory Leak Detection To support memory leak detection, various approaches and tools have been developed over the last years. Šor and Srirama [45] classify these approaches into the following groups:

1. *Online approaches* that actively monitor and interact with the running virtual machine, separated into approaches that
   a. *measure staleness* [3, 14, 35, 36, 38, 59]. The longer an object is not used, the staler it becomes. Staleness analysis tries to reveal objects that do not get collected by the GC but become stale, since they are most likely to be leaking. The challenge these approaches face is that tracking object accesses is extremely expensive.
   b. *detect growth* [4, 20, 21, 43, 44]. These approaches group the live heap objects (usually either by their types or allocation sites) and detect growth using various metrics.

2. *Offline approaches* that collect information about an application for later analysis, separated into approaches that
   a. *analyze heap dumps and other kinds of captured state* [22, 27–29]. Compared to online approaches, offline approaches often perform more complicated analyses based on the object reference graph, involving graph reduction, graph mining and ownership analysis.
   b. *use visualization* to aid leak detection [7, 31, 37].
   c. *employ static source code analysis* [9, 60].
3. *Hybrid approaches* that combine online features as well as offline features [10, 39, 58].

In this taxonomy, AntTracks would be classified as a *hybrid approach*. It collects detailed memory traces *online* using its VM, while the processing of these traces happens *offline*. Its offline analysis tool mostly focuses on the visualization and automatic detection of heap growth.

Memory Churn Analysis For example, Peiris and Hill [32] presented EMAD, the *Excessive Memory Allocation Detector*. Compared to AntTracks, which detects memory churn offline using memory traces, EMAD uses dynamic binary instrumentation and exploratory data analysis to determine whether an application performs excessive dynamic memory allocations.

5 Limitations

One limitation of our work is its currently limited evaluation based on a small set of use cases. We plan to find more open-source projects that suffered from memory anomalies in the past which we can use to build a reference set of real-world applications. This collection could then be used to evaluate memory monitoring tools.

In addition to that, we are currently conducting a user study. One goal of this study is to see how well people with different backgrounds are able to detect suspicious time windows themselves. Preliminary results suggest that novice users are not always able to recognize suspicious memory growth in an application. Also, it seems that most users also underestimate the possible severity of high memory churn.

6 Future Work

User Study As stated in the previous section, we are currently conducting a user study with university students having various levels of expertise. The aim of the study is to gain insights on how well the study participants are able to analyze memory anomalies with AntTracks and which features they would expect from a memory monitoring tool in general. This could help the community to improve the quality of memory monitoring tools.

Visualization Many of AntTracks's analysis features communicate their results in form of tables, line charts or stacked area charts. We plan to evaluate alternative visualization approaches, for example, an application's memory evolution could also be displayed as *small multiples* [50] or as *a software city* [57]. To investigate keep-alive relationships i a heap state, we plan to support users by displaying aggregated heap objects as graphs [1].

Guided Exploration The aim of this work is to automate the first step of memory evolution analysis over time, namely the selection of a suspicious time window. Nevertheless, once a time window is selected, an appropriate analysis approach has to be selected, and the users are left on their own during this analysis. Thus, we are currently integrating features into AntTracks that we call *guided exploration*. The goal of guided exploration is to lead users through AntTracks's different analysis views. Within each view, those parts that contain the most information should automatically be detected, highlighted and explained to the user. This way, we want to achieve a learning-by-doing effect, with the goal that AntTracks should be usable by users without any prior memory monitoring experience.

7 Conclusion

In this paper, we presented an approach to automatically detect time windows that show typical behaviors of various memory anomalies. Freeing users from this non-trivial task enables them to focus more on finding the root cause of the problem. Especially inexperienced users that often struggle to recognize anomalies on their own profit from this feature.

The first type of memory anomaly that our approach is able to automatically detect is *continuous memory growth* caused by memory leaks.

The second type are windows that suffer from *high GC overhead*. High GC overhead can stem from two main root causes. Either the individual garbage collections within the time window took a long time, or a very high number of garbage collections had to be performed.

The third type of suspicious time windows we are able to detect are those that show *high memory churn*. High memory churn is caused by objects that are frequently allocated and freed shortly after their allocation. This also leads to a high number of garbage collections.

These algorithms to automatically select suspicious time windows have been integrated into AntTracks, a memory monitoring tool developed by us. Throughout the paper, their applicability was shown by applying them to different real-world applications.

Acknowledgments

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

References

[1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proc. of the 5th Int'l Symp. on Software Visualization (SOFTVIS '10)*.

[2] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*.

[3] Michael D. Bond and Kathryn S. McKinley. 2006. Bell: Bit-encoding Online Memory Leak Detection. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.

[4] K. Chen and J. Chen. 2007. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Proc. of the 31st Annual Int'l Computer Software and Applications Conf. (COMPSAC '07)*.

[5] Trishul Chilimbi, Richard Jones, and Benjamin Zorn. 2000. Designing a Trace Format for Heap Allocation Events. In *Proc. of the 2nd Int'l Symposium on Memory Management (ISMM '00)*.

[6] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. 2008. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software* 81, 12 (2008), 2252 – 2268.

[7] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP '99)*.

[8] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proc. of the 4th Int'l Symposium on Memory Management (ISMM '04)*.

[9] Dino Distefano and Ivana Filipović. 2010. Memory Leaks Detection in Java by Bi-abductive Inference. In *Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering (FASE 2010)*.

[10] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended Analysis for Performance Understanding of Framework-based Applications. In *Proc. of the 2007 Int'l Symposium on Software Testing and Analysis (ISSTA '07)*.

[11] Dynatrace. 2019. Demo Applications: easyTravel. https://community.dynatrace.com/community/display/DL/Demo+Applications+-+easyTravel

[12] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK. In *Proc. of the 13th Int'l Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.

[13] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Proc. of the 40th Int'l Conf. on Software Engineering: Comp. Proc. (ICSE '18)*.

[14] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*.

[15] Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, and Darko Stefanović. 2002. Error-free Garbage Collection Traces: How to Cheat and Not Get Caught. In *Proc. of the 2002 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*.

[16] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. 2006. Generating Object Lifetime Traces with Merlin. *ACM Trans. Program. Lang. Syst.* 28, 3 (May 2006), 476–516.

[17] Swaminathan Jayaraman, Bharat Jayaraman, and Demian Lessa. 2017. Compact Visualization of Java Program Execution. *Software: Practice and Experience* 47, 2 (2017), 163–191.

[18] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management.* Chapman and Hall/CRC.

[19] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*.

[20] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '07)*.

[21] Maria Jump and Kathryn S. McKinley. 2009. Detecting Memory Leaks in Managed Languages with Cork. *Software: Practice and Experience* 40, 1 (2009).

[22] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. 2010. Diagnosing Memory Leaks using Graph Mining on Heap Dumps. In *Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD '10)*.

[23] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.

[24] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE '15)*.

[25] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proc. of the 7th ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '16)*.

[26] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '17)*.

[27] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *Proc. of the 20th European Conf. on Object-Oriented Programming (ECOOP '06)*.

[28] Nick Mitchell and Gary Sevitsky. 2003. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP '03)*.

[29] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. In *Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications (OOPSLA '07)*.

[30] Raymond H Myers and Raymond H Myers. 1990. *Classical and modern regression with applications.* Vol. 2. Duxbury press Belmont, CA.

[31] Wim De Pauw and Gary Sevitsky. 2000. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency: Practice and Experience* 12, 14 (2000).

[32] Manjula Peiris and James H. Hill. 2016. Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern. In *Proc. of the 7th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '16)*.

[33] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2019)*.

[34] J. Qian and X. Zhou. 2012. Inferring weak references for fixing Java memory leaks. In *Proc. of the 2012 28th IEEE Int'l Conf. on Software Maintenance (ICSM '12)*.

[35] Derek Rayside and Lucy Mendel. 2007. Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks. In *Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '07)*.

[36] Derek Rayside, Lucy Mendel, and Daniel Jackson. 2006. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Proc. of the Int'l Workshop on Dynamic Systems Analysis (WODA '06)*.

[37] S. P. Reiss. 2009. Visualizing The Java Heap to Detect Memory Problems. In *Proc. of the 5th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '09)*.

[38] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. 2001. Heap Profiling for Space-efficient Java. In *Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 104–113. https://doi.org/10.1145/378795.378820

[39] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. 2000. Automatic Removal of Array Memory Leaks in Java. In *Proc. of the 9th Int'l Conference on Compiler Construction (CC '00)*.

[40] Connie U. Smith and Lloyd G. Williams. 2000. Software Performance Antipatterns. In *Proc. of the 2nd Int'l Workshop on Software and Performance (WOSP '00)*.

[41] Connie U. Smith and Lloyd G. Williams. 2002. New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. In *Intl. CMG Conf.*

[42] Connie U. Smith and Lloyd G. Williams. 2003. More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *Intl. CMG Conf.*

[43] V. Sor, P. Oü, T. Treier, and S. N. Srirama. 2013. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance (ICSM '13)*.

[44] Vladimir Šor, Nikita Salnikov-Tarnovski, and Satish Narayana Srirama. 2011. Automated Statistical Approach for Memory Leak Detection: Case Studies. In *On the Move to Meaningful Internet Systems (OTM 2011)*.

[45] Vladimir Šor and Satish Narayana Srirama. 2014. Memory leak detection in Java: Taxonomy and classification of approaches. *Journal of Systems and Software* 96 (2014).

[46] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. 2005. Extracting sequence diagram from execution trace of Java program. In *Proc. of the 8th Int'l Workshop on Principles of Software Evolution (IWPSE '05)*.

[47] Oracle. 2019. The HotSpot Group. http://openjdk.java.net/groups/hotspot/

[48] Oracle. 2019. ZGC - The Z Garbage Collector. http://openjdk.java.net/projects/zgc/

[49] Per Lidén & Stefan Karlsson. 2018. The Z Garbage Collector - An Introduction, FOSDEM 2018. http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf

[50] Stef van den Elzen and Jarke J. van Wijk. 2013. Small Multiples, Large Singles: A New Approach for Visual Data Exploration. *Comput. Graph. Forum* 32 (2013).

[51] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proc. of the 2019 ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '19)*.

[52] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proc. of the 9th Symposium on Software Performance (SSP '18)*.

[53] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proc. of the 15th Int'l Conf. on Managed Languages & Runtimes (ManLang '18)*.

[54] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '17)*.

[55] Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Companion of the 2019 ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '19)*.

[56] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proc. of the 9th ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '18)*.

[57] Richard Wettel and Michele Lanza. 2007. Visualizing Software Systems as Cities. In *Proc. of the 4th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '07)*.

[58] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '11)*.

[59] Guoqing Xu and Atanas Rountev. 2008. Precise Memory Leak Detection for Java Software Using Container Profiling. In *Proc. of the 30th Int'l Conf. on Software Engineering (ICSE '08)*.

[60] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *Proc. of the Annual IEEE/ACM Int'l Symposium on Code Generation and Optimization (CGO '14)*.

[61] H. Yu, X. Shi, and W. Feng. [n.d.]. LeakTracer: Tracing leaks along the way. In *Proc. of the 2015 IEEE 15th Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '15)*.