Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance

Markus Weninger[⋆] , Elias Gander[⊗] , Hanspeter Mössenböck[⋆]
{firstname.lastname@jku.at}
[⋆] Institute for System Software, Johannes Kepler University, Linz, Austria
[⊗] Christian Doppler Laboratory MEVSS, Johannes Kepler University, Linz, Austria

Abstract

High memory churn occurs when many temporary objects are created and shortly thereafter collected by the garbage collector. Such excessive dynamic allocations negatively impact an application's performance because (1) a great number of objects has to be allocated on the heap and (2) an increased number of garbage collections is required to collect them.

In this paper, we present ongoing research on how to support developers in detecting, understanding and resolving high memory churn in order to improve their application's performance. Based on a recorded memory trace, an algorithm automatically searches for memory churn hotspots and calculates the age at which objects die within it, since objects that die young are the major contributors to memory churn. Information about these objects, for example their types and allocation sites, can then be inspected in order to locate the problematic code locations.

To demonstrate the feasibility and applicability of our approach, we implemented and present a new *memory churn analysis* feature in AntTracks, our trace-based memory monitoring tool.

1 Introduction

A common performance anti-pattern [1] is high memory churn. High memory churn, also known as excessive dynamic allocations [6], denotes the frequent creation and collection of objects. The time it takes to allocate these objects, as well as the time spent on collecting them during garbage collection, both negatively impact an application's performance. Even though temporary objects often turn out to be superfluous and avoidable through minor adjustments of the underlying algorithms, most state-of-the-art memory monitoring tools do not provide analysis features to inspect memory churn in greater detail but rather focus on the analysis of memory leaks.

In this paper, we describe an approach to support developers during the investigation of high memory churn in garbage-collected languages. To motivate our work, we present typical causes for high memory churn in Section 2. In Section 3, we discuss how our approach automatically detects *memory churn hotspots*, i.e., time windows in which unusual amounts of garbage is collected, based on the evolution of an application's memory footprint. If a memory churn hotspot is detected, we calculate the *lifetime* of each object that died within the hotspot, as explained in Section 4. We perform this calculation since objects that die shortly after their creation are the main contributors to high memory churn. In Section 5, we discuss how lifetime information can be combined with information on other heap object properties (such as type or allocation site) to point users to code locations that should be inspected to reduce memory churn.

All concepts presented in this work have been implemented using our memory monitoring tool AntTracks [1]. AntTracks encompasses two parts: (1) a modified Java VM that collects memory traces containing information about memory events such as allocations or garbage collections [5], and (2) an offline analysis tool that can reconstruct the monitored application's heap states, i.e., the contents of the heap at different points in time, based on such a trace [7].

2 Motivation

The careless allocation of objects can lead to high memory churn that results in run-time overhead that could easily be prevented. A typical situation leading to high memory churn is the allocation of short-living temporary objects within heavily executed loops. Every iteration allocates new objects that quickly turn into garbage. Another typical problem is the use of boxed primitives as generic types, e.g., `ArrayList<Integer>`. Every time a primitive is added to such a structure, it is wrapped into a heap object, which causes unnecessary memory overhead. One last example is the careless use of streams. Often, multiple `map` operations (or similar) are used unnecessarily, causing many short-living intermediate objects to be created. Another classic mistake is to use `map` when working with primitives instead of using the respective memory-efficient operation such as `mapToInt`.

3 Memory Churn Hotspot Detection

The first step when checking an application for high memory churn is to look for *memory churn hotspots*. Figure 1 shows an application that exhibits frequent

[1] AntTracks available at: http://mevss.jku.at/AntTracks

tall spikes in its memory footprint, a typical memory churn pattern. The plot depicts the monitored application's memory footprint at the beginning and at the end of every garbage collection. Since the memory occupied at the start of a garbage collection is much higher than at its end, each garbage collection appears as the falling edge of a spike.

In [9], we presented algorithms to automatically detect suspicious patterns in an application's memory footprint that hint at memory anomalies such as high memory churn. This feature aims to help novice users that would otherwise struggle to recognize problematic patterns on their own. Currently, we only present the most critical anomalies, e.g., the strongest memory churn hotspot, to not overwhelm the (novice) users with too much information. In general, the following steps are performed to find an application's strongest memory churn hotspot (as done in Figure 1):

- Construct all possible time windows that cover between 5 and 50 garbage collections.
- Calculate each window's *garbage per second* by summing the bytes collected within the window and dividing them by the window's duration. Graphically speaking, sum the heights of all falling edges within the window and divide them by the window's width.
- Finally, select the window with the highest garbage per second. If its garbage per second is significantly higher than the application's average garbage per second, i.e., if it is a hotspot, report it (e.g., by highlighting in the plot).

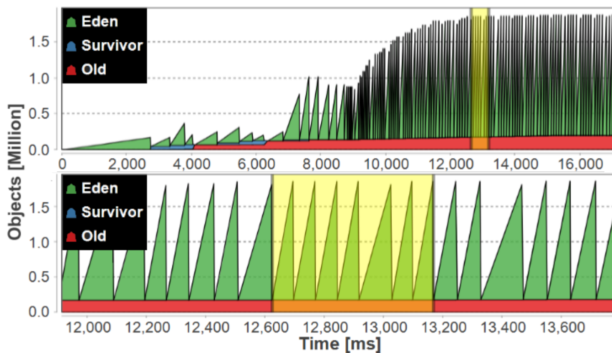

Figure 1: Automatically detected time window with high memory churn (global view and zoomed-in view).

4 Object Lifetime Calculation

To detect which objects die young, we need to know the time at which a given object was *born* and when it *died*. The *Merlin* algorithm [2] used by *Elephant Tracks* [3] could be used to calculate very exact object death times, yet it causes a several 100-fold increase in the analyzed application's run time [4]. Instead, we use less exact object ages, namely the *number of garbage collections an object survived*. This way, it is sufficient to know for each object (1) the first garbage collection following its allocation and (2) during which garbage collection it was collected. Like most memory tracers, AntTracks records events at the start and the end of garbage collections, where garbage collections are assigned consecutive IDs. As shown in Figure 2, the *birth time* of each object is set to the ID of the garbage collection following the allocation. When the garbage collector reclaims an object, it is assigned the ID of the currently running garbage collection as its *free time*. It is then straightforward to calculate the age of a died object by subtracting the two IDs.

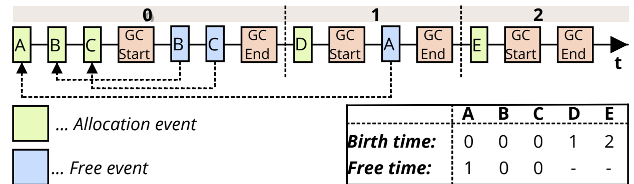

Figure 2: For every heap object, a *Birth Time* and *Free Time* is reconstructed.

5 Memory Churn Suspect Inspection

Many memory analysis tools, including AntTracks, group heap objects based on one or more criteria (such as their types) and display the number of objects and the number of bytes per heap object group [7]. This typically happens during heap state analysis, i.e., during the inspection of the heap at a given point in time.

We suggest a similar approach for memory churn analysis. Yet, instead of grouping the *live objects* at a given point in time, we group all objects that *died within a given time window*. This time window can be manually selected by the user or automatically detected, as explained in Section 3. We use a new grouping criterion, the *object lifetime grouping*, in addition to existing ones such as type and allocation site. As shown in Figure 3, this criterion aggregates the died objects into groups named *"<x> GCs survived"*. Next to each group we display the number of objects and the number of bytes that have been collected by the garbage collector within the selected time window. Major memory churn contributors can be revealed by drilling down into the largest object groups that did not survive a single garbage collection.

6 Example

Figure 3 through Figure 5 show a complete example of how AntTracks has been used to investigate a memory churn hotspot in order to improve the *finagle-http* benchmark in the Renaissance benchmark suite [8] version 0.9.0. As shown in Figure 3, inspecting the automatically detected memory churn hotspot reveals that over 99.9% of the died objects (10, 012, 077 out of 10, 019, 784) did not even survive a single garbage collection. Inspecting the types of these objects in Figure 4 reveals that most of them are divided almost equally among four types (since finagle-http is a

Scala application, its type names are typically longer than Java type names). Next, we inspected the allocation sites of these four types and found out that the allocation sites of the first three types are within library methods which we cannot modify. Yet, Figure 5 shows that the allocation site of the fourth type (which are anonymous function objects) is located in the `FinagleHttp` class, the benchmark's main class. Since such a rapid allocation and collection of anonymous function objects is unlikely to be intentional, we inspected the method's source code. In a loop, a lot of anonymous function objects were created waiting for an HTTP request to succeed before incrementing a counter. In our fixed version, only a single response handler is created which is reused for every HTTP request. This reduced the overall amount of allocated temporary objects by about 25% and sped up the application by about 5%.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
4 GCs survived	7,686
1 GCs survived	21

Figure 3: Grouping objects by the number of survived GCs facilitates high memory churn analysis.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
Promise$WaitQueue$$anon$4	2,494,576
Promise$Monitored	2,494,393
Future$$anonfun$onSuccess$1	2,494,362
FinagleHttp$$anonfun$runIteration$1$$anon2	2,494,361
char[]	3,196

Figure 4: Inspecting the types of the frequently dying objects reveals major suspect types.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
...	
FinagleHttp$$anonfun$runIteration$1$$anon2	2,494,361
FinagleHttp$$anonfun$runIteration$1$$anon$	2,494,361

Figure 5: The allocation sites of frequently dying objects lead to methods that have to be inspected.

7 Conclusion and Future Work

As high memory churn can have a substantial negative impact on an application's performance, tool support to inspect such memory anomalies is essential. In this work, we discussed common causes for memory churn, we showed how to automatically detect memory churn hotspots, we presented how to detect objects that die shortly after their allocation, and suggested a way how to utilize and visualize this information for memory churn analysis. To showcase the applicability of our approach, we implemented it in our memory monitoring tool AntTracks and presented an example on how the tool's new memory churn analysis feature has been used to improve a real-world benchmark application.

For future work, we currently focus on making AntTracks (including its new memory churn analysis) more accessible to novice users. As we evaluated AntTracks's various capabilities [10], we observed a need for more guidance during memory anomaly analysis tasks. This lead us to elaborate recommendations for memory monitoring tool developers including '*Use automation to relieve users from complex tasks*' as well as '*Provide guidance and explanations to support exploratory learning of analysis capabilities*'.

8 Acknowledgement

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] C. U. Smith and L. G. Williams. "Software Performance Antipatterns". In: *WOSP*. 2000, pp. 127–136.

[2] M. Hertz et al. "Generating Object Lifetime Traces with Merlin". In: *ACM Trans. Program. Lang. Syst.* 28.3 (May 2006), pp. 476–516.

[3] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. "Elephant Tracks: Portable Production of Complete and Precise GC Traces". In: *ISMM*. 2013, pp. 109–118.

[4] G. Xu. "Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs". In: *OOPSLA*. 2013, pp. 111–130.

[5] P. Lengauer, V. Bitto, and H. Mössenböck. "Accurate and Efficient Object Tracing for Java Applications". In: *ICPE*. 2015, pp. 51–62.

[6] M. Peiris and J. H. Hill. "Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern". In: *ICPE*. 2016, pp. 237–248.

[7] M. Weninger and H. Mössenböck. "User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring". In: *ICPE*. 2018, pp. 115–126.

[8] A. Prokopec et al. "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". In: *PLDI*. 2019, pp. 31–47.

[9] M. Weninger, E. Gander, and H. Mössenböck. "Detection of Suspicious Time Windows In Memory Monitoring". In: *MPLR*. 2019, pp. 95–104.

[10] M. Weninger et al. "Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study". In: *Proc. ACM Hum.-Comput. Interact.* 4.EICS (June 2020), 75:1–75:37.