VM-Level Memory Monitoring for Resolving Performance Problems

Philipp Lengauer

Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems,
Johannes Kepler University Linz
philipp.lengauer@jku.at

Abstract

Memory anomalies, such as memory leaks, floating garbage, and excessive garbage collection pauses, impact application performance considerably. Sadly, these anomalies often remain inexplicable as detecting and locating them is a tedious task for which only little automated tool support exists. We propose to design a Java virtual machine extension that exposes parts of its internal memory state and allows memory monitoring tools to access this state at runtime. Furthermore our goal is to automate the tuning of the Java virtual machine to counteract memory anomalies. Together with domain experts from our industrial partner, Compuware Austria, we plan to validate our approach on real-world applications.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurements; D.2.8 [*Software Engineering*]: Metrics—Performance measures

General Terms Measurements, Performance

Keywords Memory monitoring, memory anomalies, garbage collection

1. Introduction and Motivation

The way how applications allocate objects and manage object references greatly impacts their performance. While this is obviously true for applications that use unmanaged memory, it is also true for applications that use garbage-collected memory. We research memory anomalies, such as leaking memory, floating garbage, and spikes in garbage collection times, which cause performance problems that are hard to locate and even harder to resolve.

In unmanaged applications, a *memory leak* occurs if the programmer clears all references to an object without freeing it explicitly. Although in managed memory the garbage collector automatically detects unreferenced objects and reclaims the occupied memory, a memory leak can still occur; namely if an application falsely continues to reference objects that it actually does not use anymore, the garbage collector cannot free them. As memory leaks can accumulate during execution, the garbage collector compromises performance by wasting more and more time scanning unused memory. An additional (and even worse) problem is that eventually the application will run out of memory and crash.

Floating garbage are objects that are no longer referenced, but have not yet been collected. This happens if a dead object lives in a heap space A, and another dead object, which keeps a reference to it, lives in a heap space B, and both heap spaces are collected independently. During a collection of heap space A, the dead object cannot be freed. Only after the dead object in heap space B has been collected, the floating garbage in A can be collected as well. The time spent to unnecessarily scan floating garbage degrades performance. Floating garbage makes out-of-memory scenarios more frequent, thus more garbage collection cycles are necessary, degrading performance even more.

During garbage collection an application must be suspended, thus *excessive garbage collection times* make an application unresponsive. Possible reasons for such spikes are full heaps, garbage collection algorithms that are unsuited for the given heap content, or badly configured garbage collectors for the application's allocation behavior. However, choosing a suitable garbage collector and configuring it properly is by no means trivial, particularly without exhaustively trying them all.

This paper presents our research goals in locating and resolving memory anomalies in applications with garbage-collected memory. We propose a Java virtual machine extension, enabling external tools to detect and locate memory anomalies. Furthermore we intend to build a knowledge base for recommending an application-tailored garbage collector configuration.

We conduct our research in cooperation with Compuware Austria GmbH. Compuware develops leading-edge performance monitoring tools for multi-tier Java and .NET applications. In applications of their customers, sporadically occurring high GC times and permanently increasing memory consumption are problems that currently cannot be resolved with Compuwares' tools.

2. State of the Art

Memory leaks are a well-researched area: Xu et al. [1] propose to annotate coarse-grained transactions in the source code that define the life time of associated objects. Their tool monitors the program execution and detects objects that exceed the life time defined by the transaction; such objects are considered to be memory leaks. The sound partition of a program into transactions, although critical for good results, is left to the programmer. Aftandilian et al. [2] propose programmer-written assertions to declare when objects are supposedly dead. At run time their modified garbage collector checks the assertions and reports violations. This approach requires programmers to specify object lifetimes, which partly takes away the convenience of using automatic memory management in the first place. Xu et al. [3] propose a profiler that instruments calls

SPLASH '13, October 26–31, 2013, Indianapolis, IN, USA.

ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2508076

to Java's collection classes in order to track unused data objects. During garbage collection their tool calculates leakage confidence values based on the time at which an object was last retrieved from the collection. This approach is limited to objects stored in standard library collections and likely will produce false positives, as it solely relies on the last time of retrieval, which is typically influenced by user input. Printezis et al. [4] describe an architectural framework that helps tracking memory leaks by visualizing large heaps as well as changes of the heap. The framework can easily be adapted to any technology and memory manager.

In contrast to memory leaks, there is only little research about floating garbage and spikes in GC times, however, there is some research that covers allocation behavior and GC performance in general. For example, Singer et al. [5] worked on how to chose the proper heap size for an application. This is relevant for our research, because choosing an improper heap size can cause spikes in GC times. Dieckmann and Hölzle [9] analyzed the allocation behavior and object properties of the SPECjvm98 benchmarks and proposed optimizations based on their observations. Blackburn et al. [11] describe the garbage collection costs as a function of the heap size and identify key algorithmic features influencing performance.

Singer et al. [8] determine application-specific garbage collectors for a particular program without exhaustively profiling the program. They use machine learning techniques to build a prediction model and recommend a garbage collector based on a single profiling run of the program. However, they only recommend the garbage collector itself, but no configuration (e.g., heap size) for it.

Other research, partly relevant for our work, is about memory anomalies in a more general way: Chis et al. [6] define eleven patterns of inefficient memory use in Java programs, e.g., boxed primitive types in collections, sparsely populated collections, nested collections instead of flat collections. Their analysis framework monitors the execution of a Java program and counts how often the patterns occur. Pauw and Sevitsky [7] provide a tool that visualizes large heaps with relatively small graphs, making it easy to locate the root reference of a memory leak.

Some proprietary Java virtual machines (i.e., JRockit [10] and IBM JVM) are shipped with tools to retrieve and visualize statistics, e.g., garbage collection pause times, allocated bytes for each thread and memory usage of individual spaces. They are able to visualize object statistics, such as number of bytes and number of objects for each type, and suggest heap sizes tailored to the exact needs of a given application. However, most of the object statistics are gathered by suspending the virtual machine and snapshotting the heap, which considerably decreases performance.

3. Problem and Research Questions

We derived the following research questions (see Figure 1) based on the problems described in Section 1:

(1) Which monitoring capabilities are needed in Java virtual machines to support locating and resolving performance degradations in garbage-collected memory? Current Java virtual machines expose a native interface for debuggers and profilers: the Java Virtual Machine Tool Interface (JVMTI). This interface partly exposes the internal state of the executed Java application and of the virtual machine itself, e.g., the state of all objects, all threads and their according call stacks, and loaded classes. However, JVMTI does not expose other information that is required to detect memory anomalies, such as the actual layout of all objects on the heap, statistical data about garbage collection (e.g., number and age of objects collected, number of objects promoted, and total number of objects survived), nor reasons for garbage collector decisions (e.g., why objects have been not collected although they are dead or why a garbage collection was necessary). Our goal is to specify a richer tool interface that exposes the memory and the garbage collector.

(2) How can we methodically detect and locate memory anomalies that degrade performance in garbage-collected memory? As our industrial partner reports, detecting memory anomalies is a tedious task which involves labor-intensive manual analysis, with only little automated tool support available. Our goal is to define run-time metrics (measurable with the richer tool interface), such as live objects aggregated by allocation site, that indicate performance-degrading memory anomalies and help locating their cause (e.g., the allocation site, the root object, or the source line). Furthermore, we want to provide tool automation for detecting and locating memory anomalies.

(3) How can we reduce the performance impact of garbage collection by recommending garbage collection settings tailored to a given application? Current Java virtual machines support a variety of garbage collector algorithms, which can be individually configured. Some configuration options are common to all garbage collectors, but others are individual for specific ones. Finding a well-suited configuration (i.e., choosing a garbage collector and selecting a configuration from all parameter combinations) for a given application is difficult. Singer et al. [8] propose a method for selecting a garbage collector based on a single profiling run of an application. Our goal is to additionally recommend a configuration for the garbage collector that is well-suited for a given application, because we expect further performance improvements from that.

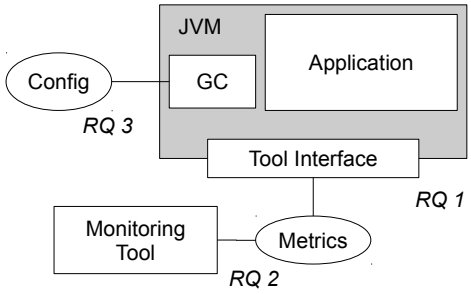

Figure 1. Key Elements and Research Questions (RQ)

4. Approach

Our approach is to define and measure run-time metrics that reflect the memory and garbage collector behavior and to compare the measurements of programs with and without memory anomalies. Based on these measurements, we plan to detect memory anomalies, locate their cause, and recommend steps to resolve them. In detail we plan the following steps:

1) Identify run-time metrics that describe the memory usage of an application. We plan to monitor object allocation and garbage collection, and thereby measure metrics that allow classifying applications by their memory usage behavior, e.g., number of objects allocated and freed, object lifetime and size, number of references traversed, and number of objects survived or promoted.

2) Specify and implement a Java virtual machine extension that exposes the data necessary to calculate the run-time metrics. We plan to extend the Java virtual machine tool interface by creating a custom virtual machine, based on the OpenJDK 8. Our extensions focus on exposing statistical heap and garbage collector data.

3) Find correlations between measurements and memory anomalies. We plan to find statistical correlations between the measurements and the existence of memory anomalies.

We plan the following additional steps for resarch question 3:

4) Measure garbage collector performance and run-time metrics for real-world applications with different garbage collector configurations. The Hotspot Java Virtual Machine (JVM) provides four different collectors with a multitude of configuration options. We

plan to define and measure reasonable combinations, e.g., the scavenging collector with varying tenuring thresholds, survivor ratios, and space resizing policies.

5) Build a knowledge base by mapping the measured performance together with run-time metrics to the according garbage collector configuration. We plan to use machine learning techniques (e.g., k-nearest neighbors, binary decisions trees, or support vector machines) to build a knowledge base from real-world applications and make it publicly available. We plan to include applications from our industrial partner and other third-parties as well, and widely used benchmarks.

6) Recommend a garbage collector configuration for a given application by measuring its run-time metrics and retrieving the best configuration of similar applications from the knowledge base.

5. Research Methodology and Evaluation

We use an iterative approach: (1) we analyze the problem together with domain experts from our industrial partner; (2) develop a prototype, and (3) evaluate the prototype with benchmarks and real-world case studies from our industrial partner.

We plan to evaluate research questions 1 and 2 from Section 3 in an laboratory experiment where developers apply our method in order to detect and locate prior-known memory anomalies in real-world applications. Moreover, we will measure run-time metrics in applications with and without memory anomalies. In applications with known memory anomalies, we will locate and fix the anomaly manually, so that we can compare two versions of the same application, one with the anomaly and one without it.

For research question 3, we plan to use two application sets, a training set and an evaluation set. We will train our recommender system on the training set and evaluate the recommendation by means of the evaluation set. In order to rate the recommendation, we will compare the run time of the recommended configuration with a reasonable amount of alternate configurations as well as a configuration from a domain expert.

6. Ongoing Research

In the last few months, we have been working on a transactional memory leak detector. Transactions are key for our industrial partner, as our partner's monitoring software aggregates most data on transactions, e.g., on a customer checkout in a web shop application. A transaction comprises a set of activities (possibly in parallel) in reaction to a user input. The start and end of a transaction is automatically detected when methods (specified by the operator) are entered, e.g., a method representing the customer checkout, or a web request is received. We observed that in such applications some transactions leak memory whereas others do not. Our prototypical transactional memory leak detector can spot such transactions, which our industrial partners monitoring software could not. In order to develop the detector, we defined a set of run-time metrics (e.g., allocated and freed objects per class and per transaction) extended the tool interface of the OpenJDK 8 virtual machine (VM) so that it provides our run- time metrics, and implemented an agent that retrieves and processes the metrics.

Our modified VM collects these metrics as follows: it adjusts the necessary counters whenever an object is allocated or freed by the garbage collector. When an object is allocated, the VM checks if a transaction is currently active and adjusts the allocation counter for the respective transaction and class. It stores the transaction in the object header, which we extended to carry the additional information. As soon as the garbage collector frees the object, the VM uses the information stored in the object header to identify the allocating transaction and to adjust the corresponding free counter. When a transaction completes, our agent detects potential memory leaks by checking the allocation and free counters for each class in order to determine how many objects have survived. For transactions with survivors, the agent locates the surviving objects on the heap and retrieves their allocation site so that the user can decide which survivors comprise an actual memory leak. As our solution involves the garbage collector, the user must wait for the next collection cycle in order to get accurate results for a transaction.

As we progress with our research of correlations between metrics and memory anomalies, we plan to include the other metrics that our VM already provides in the detector.

7. Conclusion

This paper presented our planned and ongoing research on detecting and locating memory anomalies such as memory leaks, floating garbage, and garbage collection spikes, which we consider open research issues. We propose a virtual machine extension that provides run-time metrics reflecting the memory usage and the heap content of applications. We intend to measure these metrics in applications with known memory anomalies and to find correlations between the measurements and memory anomalies. Furthermore, we propose building a knowledge base for recommending application-tailored garbage collector configurations.

Acknowledgments

This work has been supported by the Christian Doppler Forschungsgesellschaft, Austria, and Compuware Austria GmbH.

References

[1] G. Xu, M. D. Bond, F. Qin, and A. Rountev, *LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks*, Proceedings of the 32nd Conference on Programming Language Design and Implementation, 270 - 282, 2011

[2] E. E. Aftandilian and S. Z. Guyer, *GC Assertions: Using the Garbage Collector to Check Heap Properties*, Proceedings of the 2009 Conference on Programming language Design and Implementation, 235 - 244, 2009

[3] G. Xu and A. Rountev, *Precise Memory Leak Detection for Java Software Using Container Profiling*, Proceedings of the 30th International Conference on Software Engineering, 151 - 160, 2008

[4] T. Printezis and R. Jones, *GCspy: an Adaptable Heap Visualization Framework*, Proceedings of the 17th Conference on Object-oriented Programming, Systems, Languages, and Applications, 343 - 358, 2002

[5] J. Singer, R. E. Jones, G. Brown, and M. Lujn, *The Economics of Garbage Collection*, Proceedings of the 2010 International Symposium on Memory Management, 103 - 112, 2010

[6] A. E. Chis, N. Mitchel, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy, *Patterns of Memory Inefficiency*, Proceedings of the 25th European Conference on Object-oriented Programming, 383 - 407, 2011

[7] W. D. Pauw and G. Sevitsky, *Visualizing Reference Patterns for Solving Memory Leaks in Java*, Proceedings of the European Conference on Object-oriented Programming, 1999

[8] J. Singer, G. Brown, I. Watson, and J. Cavazos, *Intelligent Selection of Application-specific Garbage Collectors*, Proceedings of the 6th International Symposium on Memory Management, 91 - 102, 2007

[9] S. Dieckmann and Urs Hölzle, *A Study of Allocation Behavior of the SPECjvm98 Java Benchmarks*, Proceedings of European Conference on Object-oriented Programming, 1999

[10] M. Hirt and M. Lagergren *Oracle JRockit - The Definitive Guide*, ISBN 978-1-847198-06-8, 2010

[11] S. M. Blackburn, P. Cheng, and K. S. McKinley *Myths and Realities: the Performance Impact of Garbage Collection*, Proceedings of the Joint International Conference on Measurement and Modelling of Computer Systems, 25 - 36, 2004