A Practical and Extensible Framework for Garbage Collection Tracing

Extended Abstract

Xuanrui (Ray) Qi
Tufts University
Medford, Massachusetts
xqi01@cs.tufts.edu

ABSTRACT

This extended abstract presents a new tool for memory tracing, Elephant Tracks II. Elephant Tracks II (or ET2) is a portable, modular and extensible memory tracing tool designed for practical memory tracing of garbage-collected programs, producing precise traces of the program's heap operations, including allocation, pointer mutation, procedure entry & exit, and object deaths, using the Merlin algorithm to compute death times. Unlike all previous tools, however, ET2 is capable to support multiple programming languages by decoupling the tracing phase and the death time computation phase. We describe some of the high-level design and low-level implementation strategies employed to support this extensibility and portability.

CCS CONCEPTS

• **Software and its engineering → Garbage collection**; *Software performance*.

KEYWORDS

Tracing; garbage collection; Merlin algorithm; inter-language operability

ACM Reference Format:
Xuanrui (Ray) Qi. 2018. A Practical and Extensible Framework for Garbage Collection Tracing: Extended Abstract. In *Proceedings of SPLASH 2018 Student Research Competition (SPLASH SRC'18).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 MOTIVATION

It is hard to trace the memory usage and behavior of programs written in programming languages with automatic memory management semantics. Most of the time, this is not a problem. However, there are cases where we want to know the details of memory usage, such as when we want to debug memory leaks [2].

To trace the memory usage of such programs, one may use dynamic analysis techniques. Specifically, the method of *program tracing*, or "tracing" information about the execution of programs using textual logs, is of special interest to us. With program tracing, we can record the allocation of objects, the flow of pointers (i.e. pointer updates), pointer access, and data updates by instrumenting the running program.

2 CHALLENGES

However, even with program tracing, the memory usage behavior of automatically memory managed programs is not completely clear. Specifically, the death time of objects, or the last time at which the object is last accessible, is not obvious from the program's execution trace. Fortunately, the Merlin algorithm [1] gives us a method to compute those death times, by doing a depth-first search on the graph of dead objects, and propagating known information about death times through the graph.

However, this pose us a serious problem in the implementation of GC tracers: up until now, GC tracers have to perform death-time computations during program execution, accessing the program's memory graph from time to time. This is a huge concern, because it slows down the GC tracer significantly, rendering GC tracers much less powerful than they should be. Our previous tool, Elephant Tracks [6], can slow down programs by 500 to 1000 times. Moreover, as Elephant Tracks works closely with the Java Virtual Machine, it could only be used as a analysis tool for Java programs. A successor to Elephant Tracks, AntTracks [3], is much more efficient, but unfortunately it is built into the HotSpot Java Virtual Machine, making its approach tedious to migrate to other programming languages; moreover, it does not perform death time computation. It is difficult to have both efficiency and inter-system operability within a memory tracer.

Efficiency and extensibility of GC tracers are, thus, both major challenges in garbage collection research. Without efficient GC tracers, it is difficult to collect data about GC performance, for example. Without extensible GC tracers, one has to implement from scratch a tracer for each program runtime system, making the engineering process tedious. Our main goal is, thus, to devise a new architecture and, eventually, a GC tracing tool that is both *efficient* and *extensible*. We believe that our new GC tracing framework, Elephant Tracks II, is the only tracing tool that could easily be extended to different programming platforms and different programming models.

3 IMPROVING EFFICIENCY

One could use the Merlin algorithm to compute the death time of objects, but running the Merlin algorithm is costly because it requires traversing the graph of dead objects at least once, on every garbage collection [6]. This drastically slows down the GC tracer. Moreover, the GC tracer must also keep a "shadow" copy of the heap graph, drastically increasing memory usage.

SPLASH SRC'18, November 2018, Boston, Massachusetts USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Nevertheless, it is possible to create an otherwise complete program trace and generate death times from the otherwise complete data [1]. In Elephant Tracks II, we use this approach to optimize the time and space requirements of the GC tracer. For our Java GC tracer, we use the JVM Tooling Interface [5] and the JNIF Java bytecode manipulation library [4] to dynamically rewrite all classes and instrument all code by adding calls to methods that generate traces, from which we could compute the death times of objects.

To compute the death times, however, one needs to record extra information beyond what is generally expected in GC traces. Specifically, we need *witness* records, which tell us when objects were accessed. When an object is no longer accessed and subsequently garbage collected, we may assume safely that the last access timestamp recorded is the idealized death time of the object. For example, if a object is last accessed at timestamp 20, we record 20 as the death time of the object, even if it is not actually collected until timestamp 47. We add these witness records whenever an object is accessed: in Java, for instance, whenever the object is directly accessed, passed into a method, or when a instance method is invoked on the object.

Moreover, previously, foreign function calls via the Java Native Interface were used for instrumentation calls [6], which results in the abstraction barriers being broken frequently, and thus a lot of overhead. However, in our implementation of the Java GC tracer, we use Java to implement all instrumentation calls, which results in less overhead and better efficiency overall. As a result of the aforementioned improvements, our Java GC tracing tool has sped up GC tracing by 5 to 50 times on a number of small benchmarks.

4 MODULARITY AND EXTENSIBILITY

Previously available GC tracing tools are not only inefficient, but also not extensible. Particularly, most of them work only for one (or a few) programming language implementations. However, given that most programming languages implementations have a very similar memory model at a low-level, one should be able to use the same processor to compute death times and analyze traces.

Thus, we propose a modular GC tracing architecture, in which a GC tracing tool consists of a language-specific frontend and a language-agnostic backend. Separating the program tracing phase ("frontend") with the computationally heavy trace analysis phase ("backend") improves the performance of our tool, as expensive computations are no longer computed at the tracing phase. This design also allows for better modularity, which will, in turn, allow us to increase the usability of our framework.

The frontend, as described above, is bundled to a specific runtime system, and emits a complete trace of memory events (of course, excluding death events) during an execution of a program. The backend executes the Merlin algorithm on the trace, computes death times, and could also be modularly extended to complete other analysis tasks. With this modular design, we can support GC tracing for more programming languages with much less effort, as only a tracing frontend needs to be implemented. More specifically, as long as access to the underlying object and pointer model in the runtime system is available, it is technically possible to implement an Elephant Tracks II frontend for that runtime system. For example, Haskell as a programming language does not have pointers, but the Glasgow Haskell Compiler uses pointer structures to represent programming constructs such as thunks and closures; as a result, we may perform memory tracing on these structures using the Elephant Tracks II framework.

5 FUTURE WORK

Although our work on Elephant Tracks II has given promising preliminary results, multiple future work directions remain. First of all, more engineering work is required, especially on the frontend, as it is tricky to instrument many Java language features (such as reflection). We are also interested in porting Elephant Tracks II to other programming languages, as well as implement more features for the backend.

On the other hand, we are also interested in making sense of our memory traces. Specifically, we hope to look for more domain-specific patterns in memory traces, and using those traces to improve garbage collectors. We are also interested in devising machine learning methods to learn from GC traces. Overall, we believe that memory traces contain a great wealth of knowledge, and Elephant Tracks II can help us uncover this wealth more effectively, but current methods of mining knowledge from traces are still inadequate.

6 CONCLUSION AND CONTRIBUTIONS

In this research, we present Elephant Tracks II, an extensible, efficient and practical framework for GC tracing, and present the main techniques used in the implementation of Elephant Tracks II. Although our research is still in a preliminary stage, we have reached some positive results with our current implementation.

Overall, our contributions do not only include devising a new GC tracing framework and methods for its implementation, but also include a new way to think about the architecture of program tracers in general. Using our new modular architecture, we could potentially make many program tracers more extensible and modular with a moderate amount of engineering. This will shed a new light on the implementation of dynamic program analyzers, in general.

ACKNOWLEDGMENTS

I would like to thank my research advisor, Professor Samuel Guyer, as well as external collaborators on the Elephant Tracks project: JC Beyler, Steve Blackburn, Man Cao, Wessam Hassanein, Kathryn McKinley, Ryan Rose, and Leandro Watanabe, in alphabetical order. I would also like to thank the National Science Foundation for supporting this work in part.

REFERENCES

[1] 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. https://doi.org/10.1145/1133651.1133654

[2] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. MemInsight: Platform-independent Memory Debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 345–356. https://doi.org/10.1145/2786805.2786860

[3] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/2668930.2688037

[4] Luis Mastrangelo and Matthias Hauswirth. 2014. JNIF: Java Native Instrumentation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines,*

Languages, and Tools (PPPJ '14). ACM, New York, NY, USA, 194–199. https://doi.org/10.1145/2647508.2647516

[5] Oracle Corporation. 2013. JVM Tool Interface 1.2.3. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html. (2013). Accessed March 16, 2018.

[6] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant Tracks: Portable Production of Complete and Precise GC Traces. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 109–118. https://doi.org/10.1145/2464157.2466484