Imp: Object Impact Performance on GC by Profiling

Jake Roemer, Hailong Zhang, Ganga Reddy

December 10, 2015

1 Introduction

Memory leaks have a big impact on performance by wasting memory resources, and by increasing the frequency and amount of work garbage collection does. Many techniques have been developed to detect memory leaks and improve program performance by tolerating memory corruption. Some tools use user annotations to help accurately and precisely detect memory leaks and give useful diagnostic information [5, 4]. Other techniques try to tolerate memory leaks when memory corruption manifests by eliminating or moving the object leak [1, 2]. What many of these techniques fail to report is how useful fixing a memory leak will be.

Since memory leaks impact memory resources and garbage collection performance and frequency, then not only is the detection and culprit information of a memory leak needed, but also its impact on these metrics. This is because an object can be considered a memory leak, but may not be worth fixing.

We see this disparity arise because the lifetime of an object can not be captured exactly. Either garbage collectors over-approximate object liveness by tracking the reachability of an object from heap roots. Or memory leak detectors under-approximate object deadness with staleness based tracking of how many garbage collections an object has survived, but has since been used. Therefore, without precise object lifetime information it becomes a difficult tasks to distinguish harmful leaks from benign leaks.

To try and improve the estimation of object deadness from staleness tracking we implemented *Imp*, a memory leak impact profiler. Imp tracks object staleness to obtain candidates for memory leaks and measures the impact the stale object has on garbage collection performance. Since Imp tracks the staleness of an object, it is easy to provide useful diagnostic information from any staleness detector in addition to Imp's impact profile.

2 Related Works

To achieve impact evaluation of different objects, *Imp* uses the prior work Leak Pruning [2] which predicts and reclaims significantly stale objects. It preserve semantics of programs by poisoning references to reclaimed objects.

In addition to reclaiming objects, *Imp* also employs the prior work CCU [3] for a state of the art staleness detector. Both CCU and leak pruning approximates logarithmic staleness of objects.

3 Overview

We propose *Imp*, a memory leak impact profiling technique, which detects potential memory leaks via object staleness tracking and provides impact information the memory leak has on garbage collection. Since GC performance relies on both frequency and pause time the memory leak must be removed to accurately measure both. By removing the object, the GC performance comparison must be done in separate runs. After the first set of trials running the program, GC performance is tracked without removing any memory leaks. This gives a snapshot of the base GC performance with the memory leak. Then a second set of trials running the program is performed, but now the memory leak is removed when identified and the program continues execution. This gives a snapshot of GC performance without the memory leak. Lastly, the two sets of trials are compared and the memory leak's impact on GC performance is calculated.

3.1 Staleness

To measure the impact on garbage collection, objects are chosen as memory leak candidates by tracking staleness. To detect staleness, each object tracks a staleness level as part of the object's header. The staleness level will reflect a range of GC calls that the object has not been accessed during. Once the object is determined stale enough, as in its staleness level is significantly high, it is ready to be selected for profiling. The staleness tracking is done by the CCU [3] project and was not changed by Imp.

3.2 Removal

Once an object is considered stale enough it becomes a candidate for removal from the heap. Since staleness is tracked in a range instead of an exactly number of survived GC calls, many objects can be considered significantly stale during the same removal trace. In the event of a tie, the largest object is selected for deletion. The object removal is merged into CCU from the Leak Pruning [2] project and has been modified by *Imp*.

Our modifications restrict the number of deleted objects per run to a single type descriptor. So instead of a single object being impacted, *Imp* considers a source to target type descriptor from the candidate set. Therefore, multiple objects can be removed per run, but only if they are of the same type descriptor.

The second modification, effects how objects are removed. Leak Pruning originally poisoned a reference from a source node to a target node in a heap trace. The poisoned reference would prevent a transitive closure from marking the target objects of the poisoned reference which would in turn be swept away. *Imp* instead nullifies the reference from source to target to achieve the same effect as sweeping the target objects, but results in null pointer exception errors if the object is accessed later.

Imp did not change when objects are removed from how Leak Pruning is originally set up. To maintain correctness, but still removing objects Leak Pruning waits until an out of memory error occurs before selecting and removing an object. Even though *Imp* could change when selecting and removal occurs, we decided to wait until out of memory arose to limit the potential errors of removing an object that will be used again.

3.3 Impact Profile

After an object is selected and removed, the program will either complete execution or officially run out of memory. At this time, GC time and mutator time can be collected from the run. *Imp* also performs a second run without removing objects as the comparison run, which also collects GC time and mutator time. From this information *Imp* takes the relative percentage of time GC performed by following equation 1.

$$gcTime = \frac{gcTime}{gcTime + muTime} \tag{1}$$

Imp uses relative percentage since the base GC performance will be from program start to the first out of memory error and the GC performance without the memory leak will be from program start to the a later memory error. Even though this gives different GC times, the relative GC time compared to mutator time should be an accurate measurement.

Then *Imp* takes the difference between base gcTime and gcTime without the memory leak to get the Impact Score for a type descriptor. Once we accumulate an Impact Score for all significant type descriptors a profile of the Impact Scores is given. As we mentioned before, any staleness detector will work for tracking object staleness and so any diagnostic information that is tracked in addition to the profile can also be reported at this time.

4 Implementation

4.1 Configuration

We implement *Imp* in JikesRVM 3.1.1, targeting IA32 architecture and building with the FastAdaptive configuration. The experiments are done on a dual-core 2.2 GHz Intel i5-5200U system with 8GB memory. Each processor has a 32KB-L1 instruction cache, 32KB-L1 data cache, and 256KD-L2 cache. All configurations for each benchmark are run for 10 trials. We use a Generation Mark Sweep garbage collector and set the heap size to 25M.

4.2 Benchmark

We profiled object GC impact scores using the DaCapo version 2006-10-MR1 benchmark bloat.

5 Results

To generate impact profiles, *Imp* runs an application once without deleting objects while tracking staleness to produce a list of candidates for profiling. Once the candidate list is generated, we prune the list down to three candidates for individual profiling. Then a set of 10 trials is run with BaseConfig for base gcTime and ImpFirst,Second,ThirdCand configuration for type descriptor gcTime. The different Imp*Cand configurations target one of three different candidates for profiling. Once the 10 trials are completed, impact scores are then reported for the First, Second, and Third candidates based on equation 1. Since candidates are type descriptors, they read source; to target;.

5.1 First Candidate

LEDU/purdue/cs/bloat/file/ClassFileLoader;
Ljava/util/HashMap;

Impact Score: -0.01884402706845817%

5.2 Second Candidate

[Ljava/util/HashMap$HashEntry;
Ljava/util/HashMap$HashEntry;

Impact Score: -0.0200397489910745 24%

5.3 Third Candidate

Ljava/util/zip/ZipFile;
Ljava/util/LinkedHashMap;

Impact Score: -0.04372875041890267%

We can see here the impact of these candidates is negligible on total percentage of garbage collection performance. These results are not too surprising. First we are taking a difference between Base GC performance and GC performance after removing a memory leak. This causes a discrepancy between the length of the application, resulting in the need for a relative percentage instead of a more accurate difference in gcTime. Second, the life of the program after the memory leak is removed can greatly affect the impact score. This is because GC needs to run for long enough to get an accurate representation of the savings from removing the memory leak.

We can also see that the impact scores are negative. This is because by removing the memory leak total GC performance grew worse. This is also related to the fact that we continue executing the application after out of memory error occurs. With only removing one object, we are leaving the heap in an almost full state. This causes more frequent GC calls and extends the pause time of collection to process the almost full heap.

6 Conclusion

We have implemented *Imp*, a memory leak impact profiling technique, to give programmers a useful understanding of how memory leaks are directly impacting GC performance. We were able to successfully detect and remove significantly stale objects to evaluate their impact on GC performance. We believe with further improvements, the impact profile can reveal more interesting results. Currently, only three candidates are considered for impact profiling, but this can be extended to systematically profiling all type descriptors who become significantly stale. In addition to evaluating more candidates, a broader set of benchmarks would produce a stronger set of profiling results. We believe this to be true due to the large variation in how much applications are effected by memory leaks. To directly impact the evaluation of the impact scores, using object lifetime profiling and removing an object before an out of memory error occurs would give a much more precise measurement of gcTime and the benefit GC performance has from removing the memory leak.

References

[1] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 109–126, 2008.

[2] M. D. Bond and K. S. McKinley. Leak pruning. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 277–288, 2009.

[3] J. Huang and M. D. Bond. Efficient context sensitivity for dynamic analyses via calling context uptrees and customized memory management. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 53–72, 2013.

[4] G. H. Xu. Resurrector: a tunable object lifetime profiling technique for optimizing real-world programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 111–130, 2013.

[5] G. H. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 270–282, 2011.