CS396: Using a Liveness Based GC

Milind Luthra
Supervisor: Prof. Amey Karkare

2017/18-II

Contents

1 Introduction **2**
1.1 Reachability . . . 2
1.2 A Liveness-Based Garbage Collector . . . 2

2 Previous Work **2**
2.1 Heap Reference Analysis Using Access Graphs . . . 2
2.2 Liveness Based Garbage Collectors (MTP) . . . 3
2.3 Problems . . . 3
2.4 Potential Solution: Applying LGC Selectively . . . 3
2.4.1 Generational Garbage Collection . . . 3
2.4.2 Maintaining Last Use for Large Objects . . . 4
2.4.3 Changing to RGC after a Certain Depth . . . 4
2.4.4 Profile Guided Optimization . . . 4

3 Implementation **4**
3.1 Overview . . . 4
3.2 Metric . . . 5
3.3 Details . . . 5
3.3.1 The Notion of Time . . . 6
3.3.2 Modified Drag . . . 6
3.3.3 Ideal Attributes for Our Profiler . . . 6
3.4 Problems . . . 6
3.4.1 Speed . . . 6
3.4.2 Tracking Boot Classes . . . 7
3.4.3 Parameters . . . 7
3.4.4 Other Issues . . . 8
3.5 Results . . . 8
3.5.1 Speed . . . 8
3.5.2 Visualization . . . 8

4 Future Work **9**
4.1 Completing the Compiler . . . 9
4.2 Improvements . . . 9

5 Browser Extension **9**

1 Introduction

Most garbage collectors use reachability as their metric for approximating the liveness of objects. This leads to a lot of objects occupying memory after they will not be used anymore. A possible solution to this is using a liveness-based garbage collector, but it is not practical to collect all objects in this way due to speed considerations. We propose the use of profile-guided compilation step to decide which objects to collect based on liveness.

1.1 Reachability

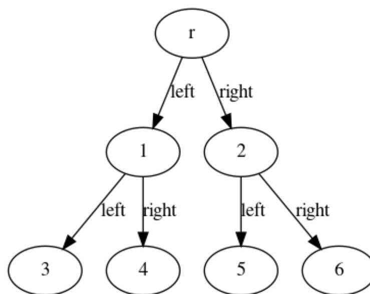

Figure 1: Example object graph for binary tree

Reachability is the usual method employed by most garbage collectors to approximate liveness. Consider a pointer `r` to the root of a binary tree. We can access the left and the right subtree using `r.left` and `r.right`. We can access the children of the subtrees in a similar way.

In this way, we can access the entire tree through `r`, and all the nodes are said to be *reachable* by `r`.

Most garbage collectors maintain a *root set* of objects - for instance, all the objects pointed to be stack variables - and any object that is reachable from this root set is safe from being collected. Any object that is not is eligible for collection.

A directed graph with objects as nodes and edges marked with the labels is called an *object graph*. An example of an object graph for the binary tree is given in 1.

1.2 A Liveness-Based Garbage Collector

A liveness-based garbage collector, that uses future use information to collect an object, can collect more garbage. But it is slower than the reachability based collector since it does not guarantee that each object will be visited only once.

The goal of this project is to come up with a practical garbage collector which can mix liveness-based garbage collection with reachability-based garbage collection for the appropriate objects.

2 Previous Work

2.1 Heap Reference Analysis Using Access Graphs

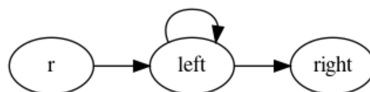

Figure 2: Example access graph

Access graphs provide a way to reason about the liveness information at point in the program. This paper [3] describes the analysis of the heap to obtain these access graphs.

Consider, for instance, the access graph provided in 2. What is denotes is that objects of the form `r.left.right`, `r.left.left.right`, `r.left.left.left.right` and so on are accessible.

An access graph can be conveniently represented as a regular expression, in this case, `r.left+.right`.

2.2 Liveness Based Garbage Collectors (MTP)

These are two implementations of a liveness-based garbage collectors [4] [2]. Both the implementations use the same basic idea: whenever the garbage collector is invoked, it first creates the object graph using all the elements in the root set, and then uses the access graphs to determine which objects are live.

In the example given in 1 along with the access graph in 2, we will be able to mark exactly one path live, that is, the path from `r->1->4`. All the other nodes which do not lie along this path can be collected, this incurring *massive* saving of space.

The first MTP [4] changes this to a CFL (Context Free Language) reachability problem. It uses the fact that regular expressions (and thus access graphs) are strictly less powerful than CFLs, and thus, we can treat them as such. Then, we look for paths in the object graph whose labels, when concatenated, are a part of the CFL, and these paths are live.

We can also model this as a regular path query, which is a common problem for making queries on certain databases and XML trees. This works by converting the access graph to a regular expression, as I have already done for my ease.

2.3 Problems

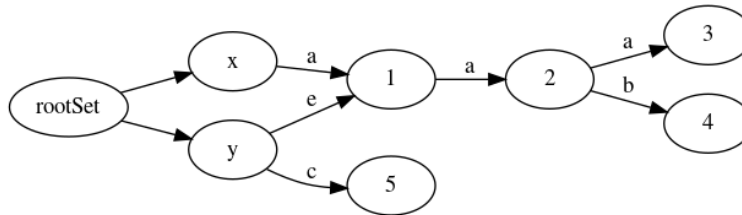

Figure 3: Object graph with potential multiple scans

LGC is slower than RGC because it needs to scan the graph again and again. While reachability ensures that we do no more than one traversal of the graph, an LGC makes no such guarantees. Consider the figure 3, and two access graphs, `x.a.a.b` and `y.e.a*`. The first access graph causes us to traverse `x`, `1`, `2`, and `4`, while the second access graph traverses `y`, `1`, `2` and `3`.

As can be seen, the nodes `1` and `2` are visited twice. Thus, even though we can save a lot of memory using LGC, this causes an unreasonable slowdown if we try to LGC *every* object. Thus, we need to focus on those objects which impact the memory the most.

2.4 Potential Solution: Applying LGC Selectively

There are quite a few ways to select what objects we will apply LGC on.

2.4.1 Generational Garbage Collection

A generational GC [1] is one that partitions the heap into several "generations". This is based on the empirical observation that a lot of objects are created and used only a few times before becoming unreachable. Thus, newly created objects are added to a "nursery". On use, an object moves to a "tenured" generation. We collect on the nursery far more often (called a minor GC) than we collect on the tenured objects (called a major GC).

It might make sense to collect the short-lived "nursery" using liveness during the minor GC cycle, since the number of objects is likely to be smaller over there, and then have occasional major GCs which still use

reachability. This works well for that code which assigns a large number of objects, and either uses them once or does not use them at all.

2.4.2 Maintaining Last Use for Large Objects

For every use of an object, we update a "last used" field in the header, and in every mark phase of our reachability-based GC, we check this field. If the object is large, and the last use is very long ago (as compared to other objects on the heap), then it might make sense to check liveness. If the object is found to be live, we update its last-used and the last-used of all the objects in its path to avoid a re-traversal the next time.

2.4.3 Changing to RGC after a Certain Depth

We traverse the object graph only to a certain depth using LGC, after which we start using RGC. In other words, if a node at a certain depth is marked visited, then its subtree will not be traversed again. This is a trade-off, a larger depth means more accurate collection at the cost of some slowdown.

2.4.4 Profile Guided Optimization

We can look to our actual use cases (profiles) to decide where to collect using liveness, and where to collect using reachability.

1. Compile-time PGO

   This involves three stages. Building a binary, then running it (on a possibly instrumented JVM) against a set of typical use cases, and noting down the allocation sites which have the largest objects reachable, but dead, for long intervals of time.

   The final step is compiling again, this time using information from the instrumentation phase to determine what areas will use liveness-based collection and which areas can use reachability.

   The advantage to this method is no runtime costs in the final program (except those incurred by using LGC on a select set of objects), but at the same time a decent "representative" input set is needed to obtain correct information. This is the main focus of this UGP.

2. Runtime PGO

   This approach seeks to combine the three stages of the above system into the JVM itself. It is possible to know when the workload of the application changes. This is called adaptive optimization, and in JikesRVM, this is done by finding "hot" methods (those which are frequently used).

   Whenever the workload changes, we can also begin instrumenting code for a while, especially in those areas which are "hot". After a few (slower than usual) executions, we will have a dynamically generated list of allocation sites with the sorted by size of dead, reachable objects into the time they exist for, and we can decide where we need to use LGC.

   The disadvantage is the cost of instrumentation. Also, if we mistakenly add LGC to an allocation site where it is not needed, it may lead to a slow down of the code. The advantage is that we can deal with a large number of workloads without prior knowledge.

3 Implementation

3.1 Overview

This is the final (compile-time PGO) based scheme. First, the source Java files are compiled using a standard compiler, like `javac`. After that, they are run on an instrumented JVM against a set of representative test cases. This yields some data indicating what objects cause the largest impact on memory, and we can annotate the source files accordingly.

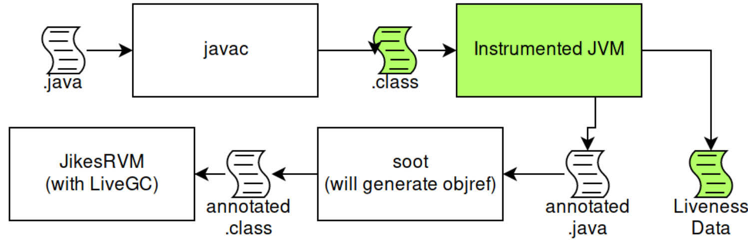

Figure 4: Overview of final scheme (green highlights the focus of this UGP)

Then, a custom compiler (to be written most likely using `soot`) is used to compile these annotated source files with liveness information, and finally, they can be run with `JikesRVM`, which (in the two MTPs [4] [2]) implements a liveness-based GC.

3.2 Metric

The metric to measure how much an object impacts memory after not being collected is called **drag** [5]. It needs to take into account both the size of the object, and the time it persists on the heap after it is used last. Thus, it becomes

$$drag = objSize \times (collectionTime - lastUseTime)$$

Similarly, our profiler also measures something called **lag** [5], which is a measure of how early an object is created. Lag has no bearing on the LGC, but makes for a useful addition for the profiler to be used as a standalone tool.

$$lag = objSize \times (firstUseTime - creationTime)$$

3.3 Details

The JVM works by having a small amount of native code loading certain critical Java classes (the so-called "boot classes"). After this, the boot classes load the application classes, and the program execution starts. We use a JavaAgent along with a library called ByteBuddy to instrument the class loading step. As soon as the classes are to be loaded, we modify the bytecode of the classes and then load them.

We need to instrument **allocation**, so we instrument the constructors. At the end of every constructor (where `this` is well-defined), we insert a call to our own code, storing certain information about the object, and uniquely tagging the object using the `hashCode`, which is (usually) a unique property associated with each Java object.

We need to instrument **use**, so we instrument method calls. At every method call, we insert a call to `useObject`, which updates the last usage time associated with that object.

We need to instrument **deallocation**, which is a bit trickier. Java does not have destructors - the closest thing it has is something called a `finalize` method. However, it is not guaranteed to be called after every garbage collection. Thus, whenever we allocate an object, we also create a `WeakReference` to it. A `WeakReference` allows us to maintain a link to the object without preventing its collection (which a normal reference would do). We check the list of `WeakReference` objects at the end of every GC, and those objects whose underlying references have become `null` are marked as being deallocated.

Usually, the GC is run only when we are low on memory. This can create a situation where an object is unreachable, but not deallocated, because the JVM has no immediate need for memory. This can give abnormally high drag for certain objects which stay on the heap far longer than even the reachability metric needs them to. Thus, we call `System.gc()` at regular intervals, each called a *cycle* .

This is essentially the resolution of our profiler, because if an object is eligible for RGC between collections, we will never know till the end of that cycle. We can keep the cycle size small to have a more accurate profiler, which will be slower as calling the GC is somewhat time-consuming.

3.3.1 The Notion of Time

Clock time is not a useful notion of time in the metric for drag. Consider for instance the code in 5. This code may not call GC after the `heavyObject` is set to `null`. Thus, it may persist on the heap for a long time while `expensiveIOOperation` is being called, and may have the same drag as an object which persists on the heap for a long time while the JVM is in need of memory.

```
int method() {
    this.heavyObject = null;
    this.expensiveIOOperation();
}
```

Figure 5: Notion of time

Thus, our notion of time is the bytes allocated since the start of the program. In this way, if an object is on the heap while the JVM is hungry for memory (is allocating heavily), then it will have a larger drag than an object on the heap when the JVM does not require much memory.

This notion of time is similar to that presented in [6] [5].

3.3.2 Modified Drag

We can thus suitably modify our idea of drag;

$$drag = objSize \times (cycleSize \times (collectionCycle - lastUsedCycle))$$

3.3.3 Ideal Attributes for Our Profiler

In this section, we look at certain attributes which make our profiler different from the idea presented in the profiler from this paper [6]. Their profiler was supposed to be run one-time to analyze the heap, while we plan to make our profiler a standalone tool which can be a regular part of the compiling process. Keeping this in mind, our profiler is the following:

- A **fast** profiler, since it aims to be a part of the compilation process, we cannot have it slow things down to a large extent. It might be run locally or on a CI, and in both cases, we do not want the build times to exceed the unprofiled build times by more than 20 times or so.
- A **standalone** profiler, not baked into the JVM. If it is baked into the JVM, the compilation process would require a separate JVM. This is not a practical scheme for something that we intend to deploy with minimal effort. Since JavaAgents are a technique of instrumentation exposed to developers by Oracle, it is likely to have a consistent API, the same cannot be said about instrumenting the JVM. We also have the benefit of composing several JavaAgents together if we need to.
- A **configurable** profiler, in case we need to adjust granularity/speed of the profiler, we should be able to.

3.4 Problems

3.4.1 Speed

The largest source of problems in my profiler was trying to improve its speed.

1. Tracking Deallocation

   As mentioned above, my first attempt at tracking deallocation was based on traversing the list of all `WeakReference` objects that I knew to be alive in the previous cycle. This could add up to a lot of objects, which needed to be traversed every cycle.

Fortunately, recent versions of Java provide something called a `ReferenceQueue`. By associating each `WeakReference` with the queue, it is automatically pushed to the queue when the GC determines that the underlying reference is eligible for collection. I run a separate collection thread, which polls the queue and does post-processing on dead objects.

2. Tracking Use

   Java programs are essentially objects interacting with each other - and this interaction is through method calls. This meant that there was a very large amount of time spent calling `useObject`. In fact, in certain programs, the profiler was spending over 70% of CPU time inside `useObject`.

   However, since we don't care about each access (we only care about the last access), it is not necessary to record all the accesses for a particular object. Further, the time we are using is the total number of bytes allocated till now, so we do not even need to record usage more than once if we have not allocated any new object (that is, we need to record usage no more than once per GC cycle).

   Concretely, in certain benchmarks from SPECjvm2008 3.5, like `compress`, the same set of 25-30 objects are accessed thousands/hundreds of thousand times, without any new objects being allocated in that interval. We need to record their access only once. Keeping this in mind, we came up with several improvements. Note that the variable `usage` is incremented each time we use an object, and set to zero when we finally record all the uses in some interval.

   - Recording only when we need to: Instead of immediately setting the last used time for each object as soon as it is used, we just store its `hashCode` in a collection, and record the usage at the end, that is, whenever usage exceeds `usageLimit` and we have allocated some new object (time has incremented).
     - Discarding redundant records: the collection we are using is a Set - thus, even if we have a thousand uses of one object, it is recorded only once since duplicates are automatically discarded.
     - The above conditions are checked only once every `usageSkip` object uses.

   The trade-off that one faces is the loss of granularity that the profiler offers.

3.4.2 Tracking Boot Classes

A lot of important Java classes (like a lot of `java.*` or `javax.*` classes) are a part of the boot class path. Thus, they are loaded before any of the application classes. There are certain problems in instrumenting boot classes (for instance, lots of native code/irregularities). They are used very commonly, so they must be instrumented (or so we thought). This creates a dilemma.

While examining the old LGC [4], it was noticed that it does not instrument these boot class files at all, only the 'new' class files that have been created to test the old LGC out. We realized that it might be reasonable to test how much the boot class path objects actually contribute to memory usage (as opposed to the 'new' classes we define) in a typical Java application.

As seen in 6, which uses certain benchmarks from 3.5, the size of the boot classes used is much more than application classes (the Y-axis denotes the sum of the size of all the objects created of boot/application class, it has no unit because JavaAgent instrumentation only provides a relative estimate of size).

The compromise that we came up with was using a parameter to decide what classes to instrument.

3.4.3 Parameters

- `usageSkip` 2
- `usageLimit` 2
- `cycleSize` 3.3
- `infix`: Only track classes with these infixes (comma separated)
- `jsonSize`: Total number of objects to save (ordered by largest drag first)

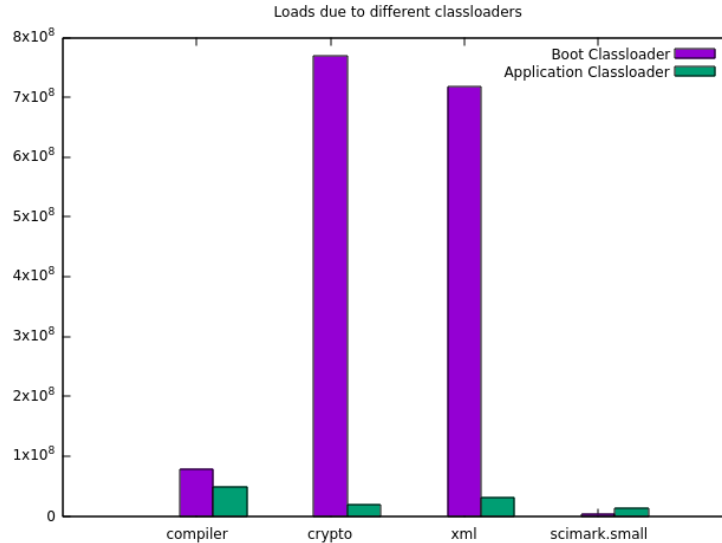

Figure 6: Impact of boot classes

3.4.4 Other Issues

1. StackMaps In the later versions of Java, a StackMap (maintaining type information) is required at every target of a jump. Its hard to construct a StackMap by hand for our instrumented code, so we enforce using Java 5.

2. Some Classes Cannot Be Tracked Some very basic classes,like `Object`, cannot be tracked, along with classes which themselves are used for instrumentation (that creates a 'cyclic' tracking issue).

3. Non-Existence of Hashcode For several boot classes, the `hashCode` does not seem to exist, causing the program to crash. This would mean that we use some inefficient representation to store objects. However, we found that a method, `System.identityHashCode` always exists, giving us a unique representation of any object.

3.5 Results

3.5.1 Speed

We used SPECjvm2008 to benchmark our profiler. The results are shown below. The X-axis has the name of the SPECjvm2008 benchmark, followed by what infixes 3.4.3 we were profiling for.

As seen, the profiler, without speedups applied 7 performs unacceptably slow on certain benchmarks, while with speedups 8 performs decently.

3.5.2 Visualization

The current profiler dumps a JSON file at the end of execution, containing liveness information. A web-based visualization mechanism9 was created for this, where one can find the objects with the largest amount of lag/drag, and see where the objects are created. (The Y-Axis is the drag, again in arbitrary units because that is what the instrumentation capability of the JVM promises us).

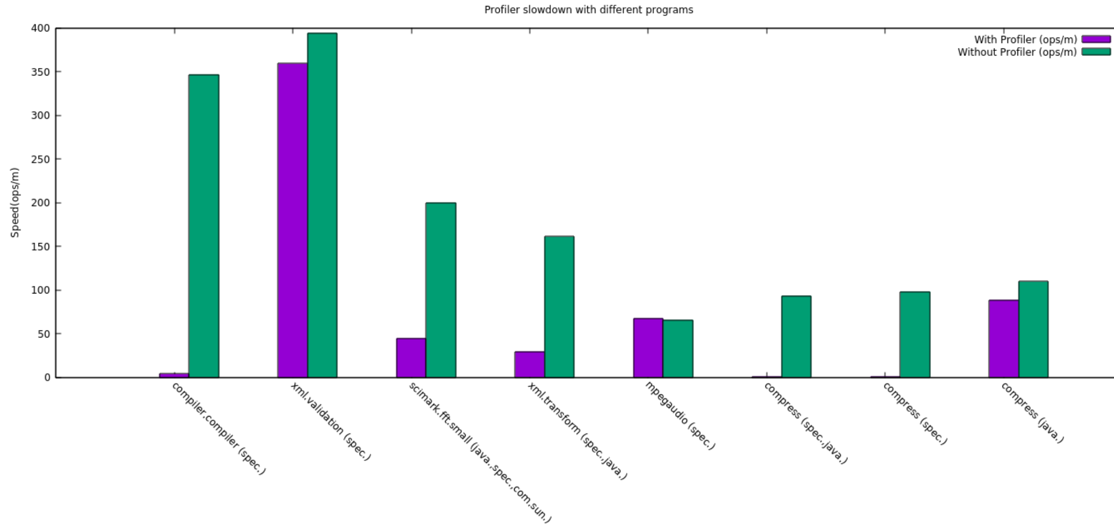

Figure 7: Profiler speed (without speedups applied)

4 Future Work

4.1 Completing the Compiler

As described in 4, there are several steps left to make the compiler complete. We need to use the liveness data to annotate the source files instead of dumping it into a JSON file, and implement the later parts of the compiler in `soot`, `JikesRVM`.

A potential problem with this is those LGC nodes which have RGC parents. These nodes will not be visited more than once. A possible solution is to mark every ancestor of an LGC node as LGC. Another possible solution is evaluating objects based on the sum of their drag and the drag of their subtree in the object graph.

4.2 Improvements

There are certain improvements that can make the profiler into a standalone tool, suitable for use without the rest of the compiler.

- Generating **StackMaps** by hand, so we can support recent versions of Java.
- Using JVMTI (JVM Tooling Interface), a JavaAgent like mechanism in C++ to speedup the profiler. We tried doing this initially, but failed because we felt that bytecode manipulation in C++ did not have sufficient library support.

5 Browser Extension

During the course of this UGP, we also came up with a browser extension which profiles tab usage, and suspends tabs to save memory. The strategy used is innovative, keeping both recency information and usage time in mind. The results of the profiler 10 encode tab use/creation information.

Memory usage goes down by around 750MB 11 12 while using the suspend function of the extension. The experiment was done using Firefox 60 (Nightly) with 50 tabs open for half an hour.

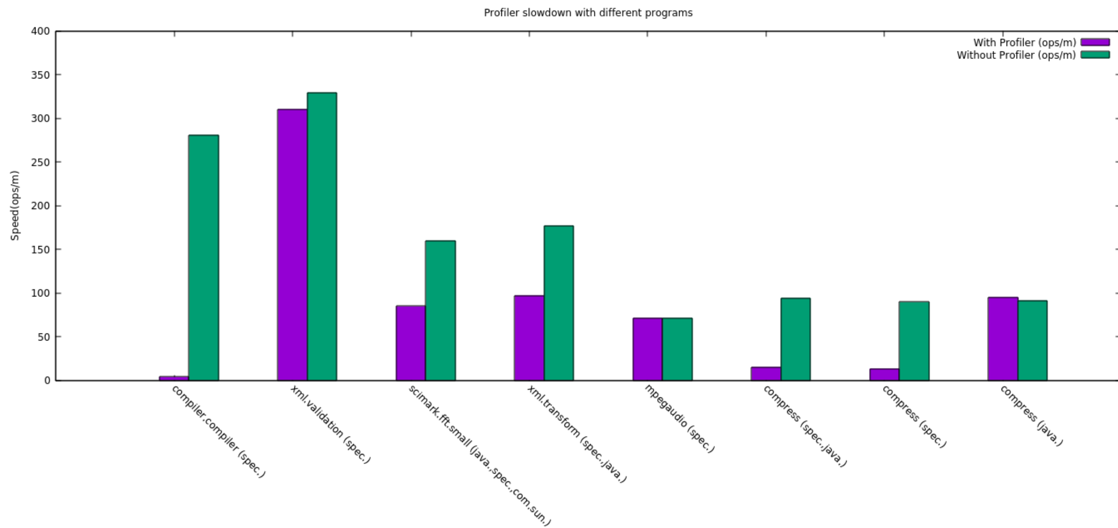

Figure 8: Profiler speed (with speedups applied)

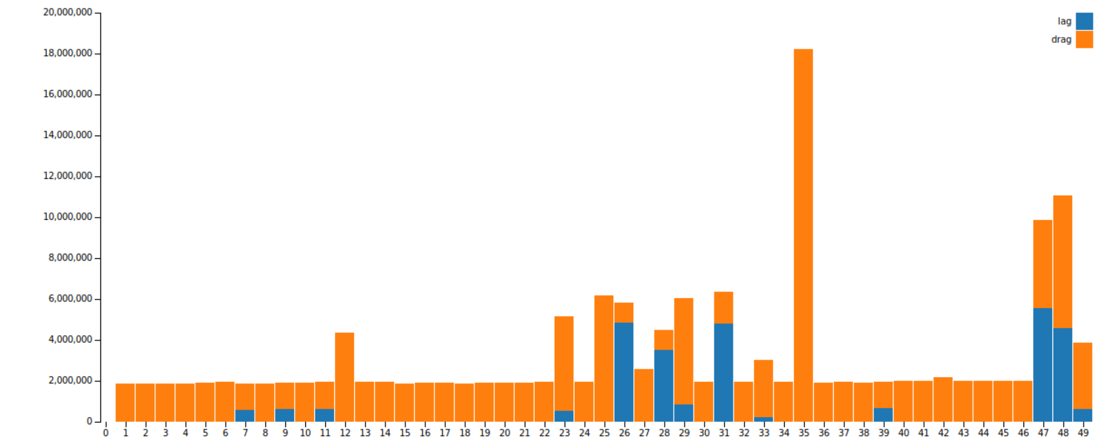

Figure 9: Profiler speed (with speedups applied)

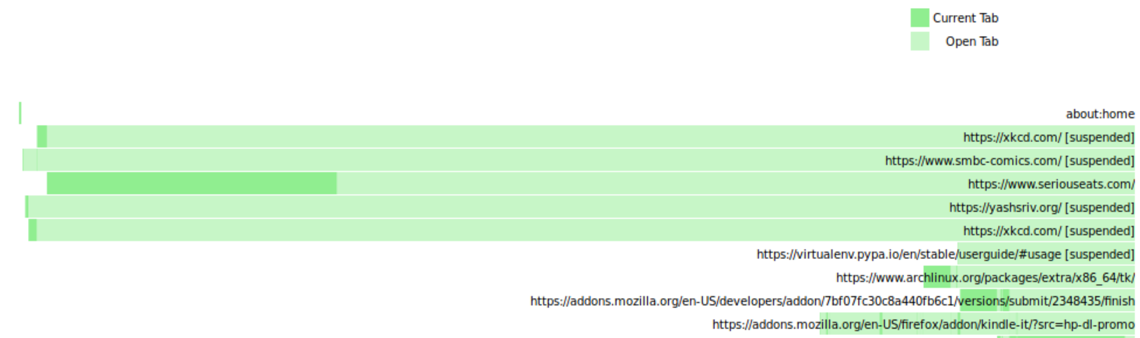

Figure 10: Browser tab profiling

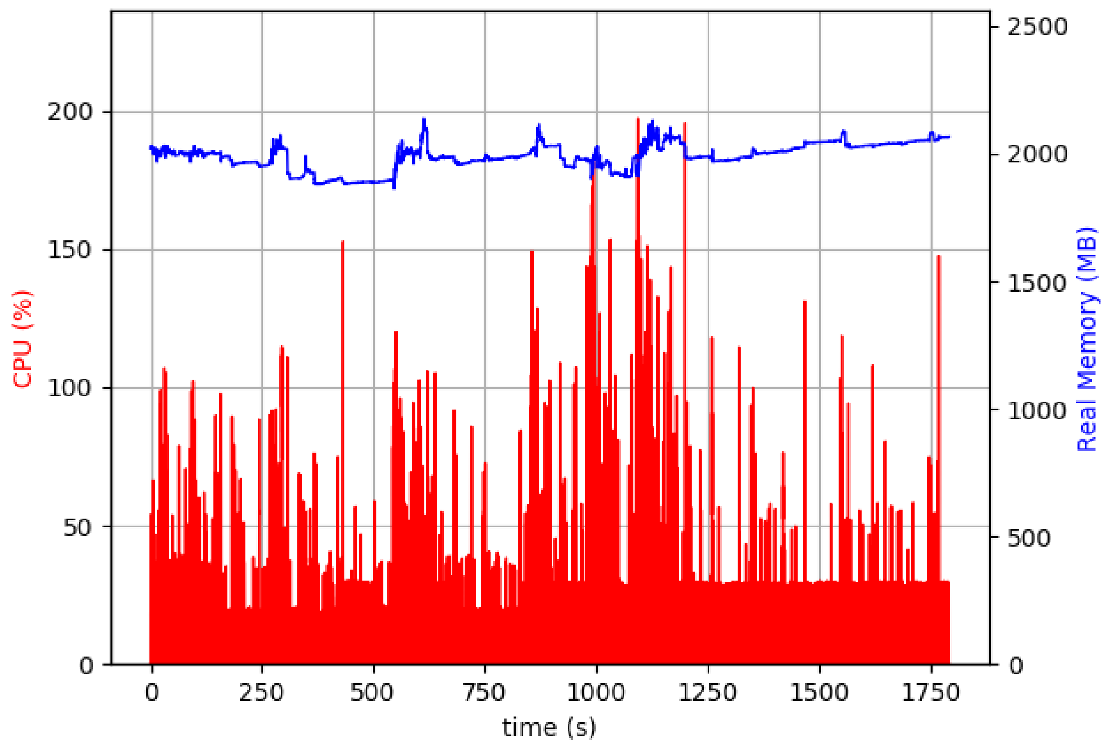

Figure 11: Memory usage with browser extension

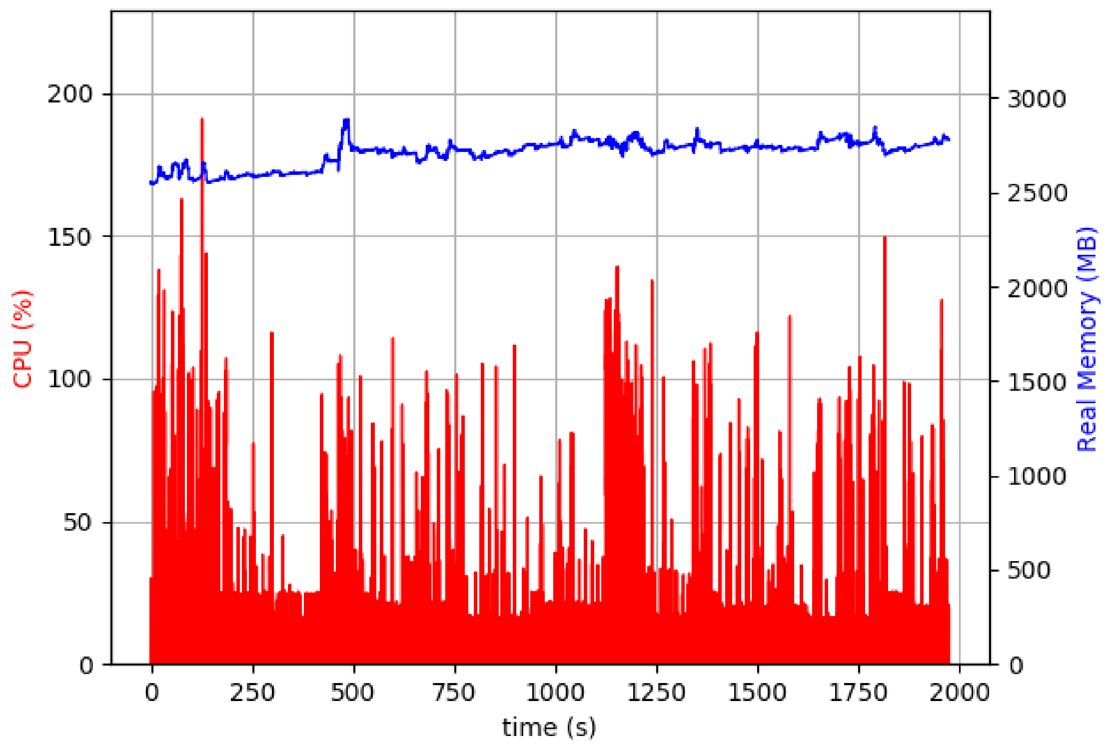

Figure 12: Memory usage without browser extension

References

[1] Fink, S., and Nyman, R. Generational garbage collection in firefox. `https://hacks.mozilla.org/2014/09/generational-garbage-collection-in-firefox/`, 2014.

[2] Kandi, V. Liveness based garbage collector for java. Master's thesis, Indian Institute of Technology, Kanpur, India, 2014.

[3] Khedker, U. P., Sanyal, A., and Karkare, A. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst. 30*, 1 (Nov. 2007).

[4] Pangarkar, N. Improving liveness based garbage collector in java. Master's thesis, Indian Institute of Technology, Kanpur, India, 2014.

[5] Röjemo, N., and Runciman, C. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. *SIGPLAN Not. 31*, 6 (June 1996), 34–41.

[6] Shaham, R., Kolodner, E. K., and Sagiv, M. Heap profiling for space-efficient java. *SIGPLAN Not. 36*, 5 (May 2001), 104–113.

- The code for this project is in a private repository, available on request.
- Weekly logs, more detailed technical documentation, more references
- Browser Extension