CS396

Using a Liveness Based GC

Milind Luthra (150363)
Supervisor: Prof. Amey Karkare

April 24, 2018

Overview

- Garbage collectors, used to free up memory during program execution, collect those objects which will not be used later in the program.

Overview

- Garbage collectors, used to free up memory during program execution, collect those objects which will not be used later in the program.
- The commonly used metric to approximate this (reachability) leaves much to be desired in terms of space saving.

Overview

- Garbage collectors, used to free up memory during program execution, collect those objects which will not be used later in the program.
- The commonly used metric to approximate this (reachability) leaves much to be desired in terms of space saving.
- Using a liveness-based GC solves this problem, but at the cost of much reduced speeds. We try to come up with a practical LGC using a profile-guided compilation step.

Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

Heap Reference Analysis Using Access Graphs

- Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs.

Heap Reference Analysis Using Access Graphs

- Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs.
- An access graph provides liveness information about the heap at any point of the program.

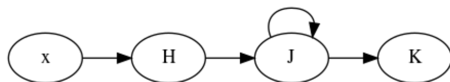
Heap Reference Analysis Using Access Graphs

- Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs.
- An access graph provides liveness information about the heap at any point of the program.
- The idea behind an liveness-based garbage collector is that when invoked, it uses access graphs to find objects which are cannot be live, and discards them.

Heap Reference Analysis Using Access Graphs

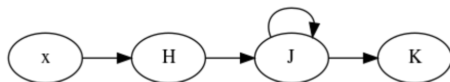
- Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs.
- An access graph provides liveness information about the heap at any point of the program.
- The idea behind an liveness-based garbage collector is that when invoked, it uses access graphs to find objects which are cannot be live, and discards them.
- Two implementations of such garbage collectors: Nikhil Pangarkar's MTP, Vilay Kandi's MTP.

Using Access Graphs



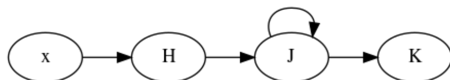
- An access graph looks like this, but I will represent it with $x.H.J+.K$ (as a regular expression).

Using Access Graphs



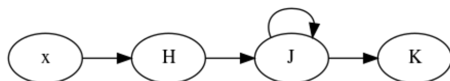
- An access graph looks like this, but I will represent it with $x.H.J+.K$ (as a regular expression).
- By finding the paths along the object graph which match this, we can find live objects.

Using Access Graphs



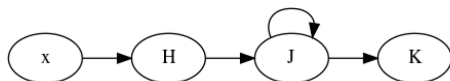
- An access graph looks like this, but I will represent it with $x.H.J+.K$ (as a regular expression).
- By finding the paths along the object graph which match this, we can find live objects.
- For instance, $x.H.J.J.K$ is live. If any object is found to be live at the end of the path, its entire path is live.

Using Access Graphs



- An access graph looks like this, but I will represent it with $x.H.J+.K$ (as a regular expression).
- By finding the paths along the object graph which match this, we can find live objects.
- For instance, $x.H.J.J.K$ is live. If any object is found to be live at the end of the path, its entire path is live.
- The first MTP models this as a CFL reachability problem (the access graph defining a CFL).

Using Access Graphs



- An access graph looks like this, but I will represent it with $x.H.J + .K$ (as a regular expression).
- By finding the paths along the object graph which match this, we can find live objects.
- For instance, $x.H.J.J.K$ is live. If any object is found to be live at the end of the path, its entire path is live.
- The first MTP models this as a CFL reachability problem (the access graph defining a CFL).
- Another way to model this is through regular path queries.

Table of Contents

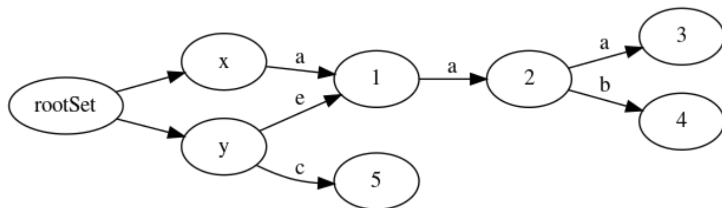
1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

Multiple Visits to a Node

- In a traditional RGC, we perform a traversal of the object graph, and mark all the nodes that are visited.

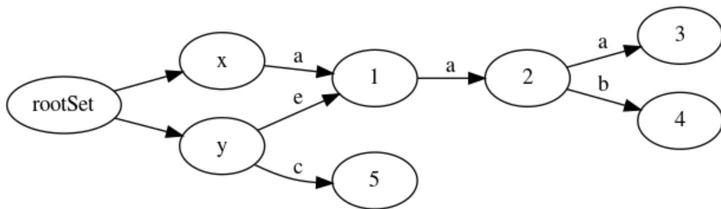
Multiple Visits to a Node

- In a traditional RGC, we perform a traversal of the object graph, and mark all the nodes that are visited.
- However, this might not be the case for an LGC.



Multiple Visits to a Node

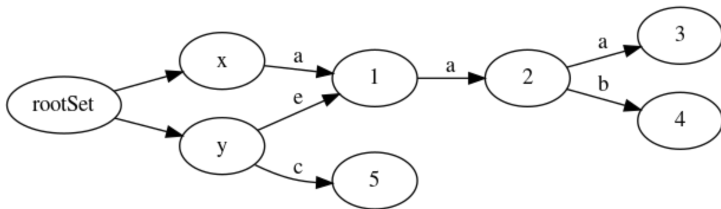
- In a traditional RGC, we perform a traversal of the object graph, and mark all the nodes that are visited.
- However, this might not be the case for an LGC.



- With the access graphs $x.a.a.b$ and $y.e.a*$.

Multiple Visits to a Node

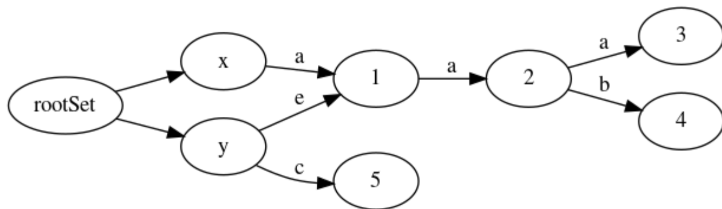
- In a traditional RGC, we perform a traversal of the object graph, and mark all the nodes that are visited.
- However, this might not be the case for an LGC.



- With the access graphs $x.a.a.b$ and $y.e.a*$.
- Node 2 is visited twice.

Multiple Visits to a Node

- In a traditional RGC, we perform a traversal of the object graph, and mark all the nodes that are visited.
- However, this might not be the case for an LGC.



- With the access graphs $x.a.a.b$ and $y.e.a*$.
- Node 2 is visited twice.
- This causes a massive slowdown, making this scheme impractical.

Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

Applying LGC Selectively

- With the cost of liveness based collection being rather high in terms of time taken, we should use it only when we need to save big on space.

Applying LGC Selectively

- With the cost of liveness based collection being rather high in terms of time taken, we should use it only when we need to save big on space.
- The problem is finding those objects which are eligible for LGC.

Applying LGC Selectively

- With the cost of liveness based collection being rather high in terms of time taken, we should use it only when we need to save big on space.
- The problem is finding those objects which are eligible for LGC.
- We need to make a trade-off between the slowdown and the memory usage of the program.

Applying LGC Selectively

- **Generational GC**
  - A generational GC has two or more areas where objects exist. One of them is the "nursery", where a small-number of newly created objects reside.

Applying LGC Selectively

- **Generational GC**
  - A generational GC has two or more areas where objects exist. One of them is the "nursery", where a small-number of newly created objects reside.
  - We LGC the nursery, and RGC when we go for a complete garbage collection.

Applying LGC Selectively

- **Generational GC**
  - A generational GC has two or more areas where objects exist. One of them is the “nursery”, where a small-number of newly created objects reside.
  - We LGC the nursery, and RGC when we go for a complete garbage collection.
  - This may work well for code which assigns a large number of objects, and will use them only a few times.

Applying LGC Selectively

- **Generational GC**
  - A generational GC has two or more areas where objects exist. One of them is the "nursery", where a small-number of newly created objects reside.
  - We LGC the nursery, and RGC when we go for a complete garbage collection.
  - This may work well for code which assigns a large number of objects, and will use them only a few times.
- **Last-use for large objects**
  - For every use of an object, we update a "lastUsed" field in the header, and in every mark phase of the GC, we check this field.

Applying LGC Selectively

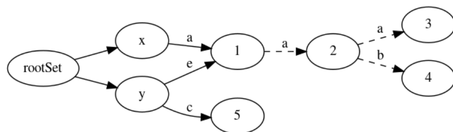
- **Generational GC**
  - A generational GC has two or more areas where objects exist. One of them is the "nursery", where a small-number of newly created objects reside.
  - We LGC the nursery, and RGC when we go for a complete garbage collection.
  - This may work well for code which assigns a large number of objects, and will use them only a few times.
- **Last-use for large objects**
  - For every use of an object, we update a "lastUsed" field in the header, and in every mark phase of the GC, we check this field.
  - Depending on the size/last-use of the object, we might choose to check the liveness of the object.

Applying LGC Selectively

- **Generational GC**
  - A generational GC has two or more areas where objects exist. One of them is the "nursery", where a small-number of newly created objects reside.
  - We LGC the nursery, and RGC when we go for a complete garbage collection.
  - This may work well for code which assigns a large number of objects, and will use them only a few times.
- **Last-use for large objects**
  - For every use of an object, we update a "lastUsed" field in the header, and in every mark phase of the GC, we check this field.
  - Depending on the size/last-use of the object, we might choose to check the liveness of the object.
  - This is basically checking whether a path from the root set exists with the labels matching a regular expression.

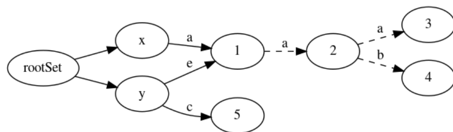
Applying LGC Selectively

- **Changing to RGC after a certain depth**
  - We only traverse the object-graph till a certain depth using LGC strategy.



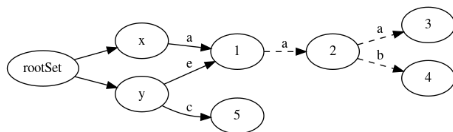
Applying LGC Selectively

- **Changing to RGC after a certain depth**
  - We only traverse the object-graph till a certain depth using LGC strategy.
  - After that, we do not visit a subtree again if its root is marked visited.



Applying LGC Selectively

- **Changing to RGC after a certain depth**
  - We only traverse the object-graph till a certain depth using LGC strategy.
  - After that, we do not visit a subtree again if its root is marked visited.
  - The deeper we go, the better collection we'll do, at the cost of larger collection times.



Applying LGC Selectively

- **Profile Guided Optimization of the GC**
  - We look at our actual use cases (profiles) to determine which objects are the most eligible for LGC.

Applying LGC Selectively

- **Profile Guided Optimization of the GC**
  - We look at our actual use cases (profiles) to determine which objects are the most eligible for LGC.
  - Large objects which linger for a long time after their death are candidates for LGC, and other objects are RGC.

Applying LGC Selectively

- **Profile Guided Optimization of the GC**
  - We look at our actual use cases (profiles) to determine which objects are the most eligible for LGC.
  - Large objects which linger for a long time after their death are candidates for LGC, and other objects are RGC.
  - We can do this at runtime. We instrument code for "hot" methods, and find sites to LGC. We instrument code every time the workload changes.

Applying LGC Selectively

- **Profile Guided Optimization of the GC**
  - We look at our actual use cases (profiles) to determine which objects are the most eligible for LGC.
  - Large objects which linger for a long time after their death are candidates for LGC, and other objects are RGC.
  - We can do this at runtime. We instrument code for "hot" methods, and find sites to LGC. We instrument code every time the workload changes.
  - We can do this at compile time. We instrument a binary against a set of profiles, and them recompile it, marking the objects which will use LGC. This is the focus of this project.

Questions

Any questions?

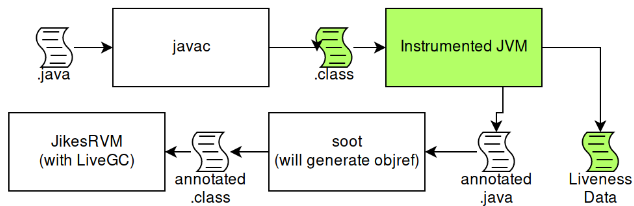
Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

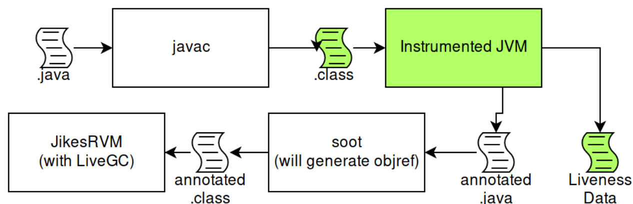
Table of Contents

1. Previous Work
    - Overview
    - Problems
    - Possible Solutions
2. Implementation
    - Overview
    - Metric
    - Details
    - Problems
    - Results
3. Future Work
4. Bonus Content

Overview

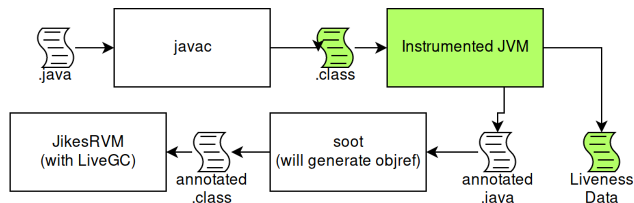


Overview



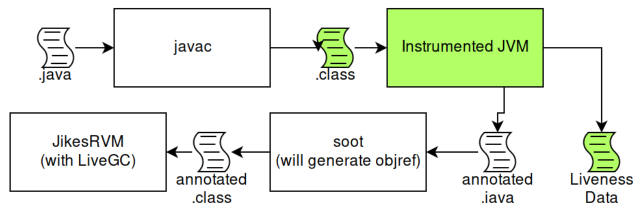
- The compiled program is run on a JVM with RGC. We can use a number of representative test cases. Object lifetimes and usage is monitored to provide liveness information.

Overview



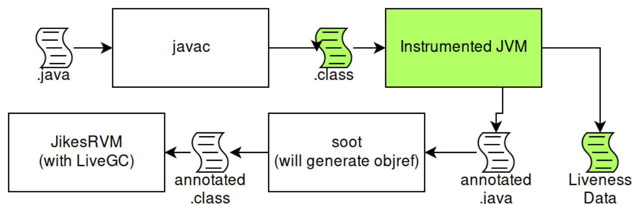
- The compiled program is run on a JVM with RGC. We can use a number of representative test cases. Object lifetimes and usage is monitored to provide liveness information.
- This liveness information can be used to generate an annotated source file, which can be later compiled into a suitably annotated executable for LGC.

Overview



- The compiled program is run on a JVM with RGC. We can use a number of representative test cases. Object lifetimes and usage is monitored to provide liveness information.
- This liveness information can be used to generate an annotated source file, which can be later compiled into a suitably annotated executable for LGC.
- Low additional cost (at runtime) to find which objects are LGC.

Overview



- The compiled program is run on a JVM with RGC. We can use a number of representative test cases. Object lifetimes and usage is monitored to provide liveness information.
- This liveness information can be used to generate an annotated source file, which can be later compiled into a suitably annotated executable for LGC.
- Low additional cost (at runtime) to find which objects are LGC.
- Requires a set of representative test cases.

Table of Contents

1. Previous Work
    - Overview
    - Problems
    - Possible Solutions
2. Implementation
    - Overview
    - Metric
    - Details
    - Problems
    - Results
3. Future Work
4. Bonus Content

Metric

- **Drag**
  - We want to measure the impact of both the time the object persists after being dead, and how large the object is.

Metric

- **Drag**
  - We want to measure the impact of both the time the object persists after being dead, and how large the object is.
  - For this, we use **drag**.

Metric

- **Drag**
  - We want to measure the impact of both the time the object persists after being dead, and how large the object is.
  - For this, we use **drag**.
  - $drag = objSize \times afterDeadTime$

Metric

- **Drag**
  - We want to measure the impact of both the time the object persists after being dead, and how large the object is.
  - For this, we use **drag**.
  - $drag = objSize \times afterDeadTime$
- **Lag**
  - Additionally, we can also measure **lag**.

Metric

- **Drag**
  - We want to measure the impact of both the time the object persists after being dead, and how large the object is.
  - For this, we use **drag**.
  - $drag = objSize \times afterDeadTime$
- **Lag**
  - Additionally, we can also measure **lag**.
  - This depends on the amount of time that the object is created before it is used.

Metric

- **Drag**
  - We want to measure the impact of both the time the object persists after being dead, and how large the object is.
  - For this, we use **drag**.
  - $drag = objSize \times afterDeadTime$
- **Lag**
  - Additionally, we can also measure **lag**.
  - This depends on the amount of time that the object is created before it is used.
  - $lag = objSize \times beforeUseTime$

Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

How the Profiler Works

- The JVM loads classes that it needs to execute.

How the Profiler Works

- The JVM loads classes that it needs to execute.
- We use a JavaAgent to intercept these class loads and instrument the bytecode using ByteBuddy.

How the Profiler Works

- The JVM loads classes that it needs to execute.
- We use a JavaAgent to intercept these class loads and instrument the bytecode using ByteBuddy.
- We instrument constructors, to track **allocation.**

How the Profiler Works

- The JVM loads classes that it needs to execute.
- We use a JavaAgent to intercept these class loads and instrument the bytecode using ByteBuddy.
- We instrument constructors, to track **allocation.**
- We instrument method calls to track **use**.

How the Profiler Works

- The JVM loads classes that it needs to execute.
- We use a JavaAgent to intercept these class loads and instrument the bytecode using ByteBuddy.
- We instrument constructors, to track **allocation.**
- We instrument method calls to track **use**.
- The closest thing to a destructor (to track **deallocation**) in Java is a finalizer, and is not necessarily called on time. Instead, we create a `WeakReference` to the object. At the end of every GC, we check these to determine whether the underlying object is alive.

How the Profiler Works

- The JVM loads classes that it needs to execute.
- We use a JavaAgent to intercept these class loads and instrument the bytecode using ByteBuddy.
- We instrument constructors, to track **allocation.**
- We instrument method calls to track **use**.
- The closest thing to a destructor (to track **deallocation**) in Java is a finalizer, and is not necessarily called on time. Instead, we create a `WeakReference` to the object. At the end of every GC, we check these to determine whether the underlying object is alive.
- We run `System.gc()` at regular intervals (called a *cycle*), to make sure that unreachable objects don't persist.

Notion of Time

- Clock time is not a useful notion of time.

Notion of Time

- Clock time is not a useful notion of time.
- For instance, an object created at the start of an expensive IO and collected immediately afterwards may have the same drag as a long-dead object of a similar size.

Notion of Time

- Clock time is not a useful notion of time.
- For instance, an object created at the start of an expensive IO and collected immediately afterwards may have the same drag as a long-dead object of a similar size.
- Instead, we use the "total bytes allocated by JVM" as our notion of time.

Notion of Time

- Clock time is not a useful notion of time.
- For instance, an object created at the start of an expensive IO and collected immediately afterwards may have the same drag as a long-dead object of a similar size.
- Instead, we use the "total bytes allocated by JVM" as our notion of time.
- This results in a larger drag for objects when the JVM is hungry for memory, as opposed to when it is not allocating.

Modified Drag

We need to suitably modify our drag calculation keeping in mind the new notion of time, and the limitation on our ability to track deallocation.

Drag Calculation

$$drag = objSize \times cycleSize \times (lastReachableCycle - lastLiveCycle)$$

where

$cycleSize =$ "interval" in bytes at at which GC is called
$lastReachableCycle =$ cycle in which collection occurs
$lastLiveCycle =$ cycle in which last use of object occurs

Attributes for Profiler

- A fast profiler, since it aims to be a part of the compilation process, it needs to be reasonably fast.

Attributes for Profiler

- A fast profiler, since it aims to be a part of the compilation process, it needs to be reasonably fast.
- A standalone profiler (not baked into the JVM) to make deployment practical (otherwise we would need two JVMs, one during the compilation step, and one to actually run our program).

Attributes for Profiler

- A fast profiler, since it aims to be a part of the compilation process, it needs to be reasonably fast.
- A standalone profiler (not baked into the JVM) to make deployment practical (otherwise we would need two JVMs, one during the compilation step, and one to actually run our program).
- A configurable profiler, sometimes we might want to measure only objects of a certain package, or trade granularity for speed.

Attributes for Profiler

- A fast profiler, since it aims to be a part of the compilation process, it needs to be reasonably fast.
- A standalone profiler (not baked into the JVM) to make deployment practical (otherwise we would need two JVMs, one during the compilation step, and one to actually run our program).
- A configurable profiler, sometimes we might want to measure only objects of a certain package, or trade granularity for speed.
- We make any measurements using SPECjvm2008. Our runs are not compliant to their rules, because we do not generate XML reports or validate the checksum.

Attributes for Profiler

- A fast profiler, since it aims to be a part of the compilation process, it needs to be reasonably fast.
- A standalone profiler (not baked into the JVM) to make deployment practical (otherwise we would need two JVMs, one during the compilation step, and one to actually run our program).
- A configurable profiler, sometimes we might want to measure only objects of a certain package, or trade granularity for speed.
- We make any measurements using SPECjvm2008. Our runs are not compliant to their rules, because we do not generate XML reports or validate the checksum.
- The idea of drag and the general profiling method is inspired from Ran Shaham, Eliot K Kolodner, Mooly Sagiv. Heap profiling for space-efficient Java.

Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

Speed!

- **Tracking Deallocation**

Speed!

- **Tracking Deallocation**
  - Tracking whether an object has been deallocated or not involves going through the entire list of reachable objects at the end of every cycle.

Speed!

- **Tracking Deallocation**
  - Tracking whether an object has been deallocated or not involves going through the entire list of reachable objects at the end of every cycle.
  - Instead of this, we now use the `ReferenceQueue` type. The `WeakReferences` that are collected are automatically added to this queue, and we only need to do post processing.

Speed!

- **Tracking Deallocation**
  - Tracking whether an object has been deallocated or not involves going through the entire list of reachable objects at the end of every cycle.
  - Instead of this, we now use the `ReferenceQueue` type. The `WeakReferences` that are collected are automatically added to this queue, and we only need to do post processing.
- **Tracking Use**

Speed!

- **Tracking Deallocation**
  - Tracking whether an object has been deallocated or not involves going through the entire list of reachable objects at the end of every cycle.
  - Instead of this, we now use the `ReferenceQueue` type. The `WeakReferences` that are collected are automatically added to this queue, and we only need to do post processing.
- **Tracking Use**
  - Analysis showed that the `useObject` was taking too much CPU time, over 70% in one benchmark.

Speed!

- **Tracking Deallocation**
  - Tracking whether an object has been deallocated or not involves going through the entire list of reachable objects at the end of every cycle.
  - Instead of this, we now use the `ReferenceQueue` type. The `WeakReferences` that are collected are automatically added to this queue, and we only need to do post processing.
- **Tracking Use**
  - Analysis showed that the `useObject` was taking too much CPU time, over 70% in one benchmark.
  - This is invoked every time an object is accessed - since most Java programs consist of objects being accessed, it adds up to *a lot*.

Speed!

- **Tracking Deallocation**
  - Tracking whether an object has been deallocated or not involves going through the entire list of reachable objects at the end of every cycle.
  - Instead of this, we now use the `ReferenceQueue` type. The `WeakReferences` that are collected are automatically added to this queue, and we only need to do post processing.
- **Tracking Use**
  - Analysis showed that the `useObject` was taking too much CPU time, over 70% in one benchmark.
  - This is invoked every time an object is accessed - since most Java programs consist of objects being accessed, it adds up to *a lot*.
  - However, we don't care about each access (only care about the last access), it is not necessary to record all the accesses. Further, the time is the number of bytes allocated till now, so we do not need to record usage more than once if no new allocations have been made.

Tracking Boot Classes

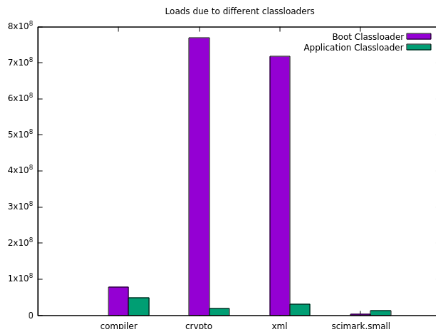
- Classes like `java.util.*`, `java.lang.*` are expected to be used in any ordinary program.

Tracking Boot Classes

- Classes like `java.util.*`, `java.lang.*` are expected to be used in any ordinary program.
- However, tracking them also causes a massive slowdown. Further, the MTP given above chooses not to use LGC on them.

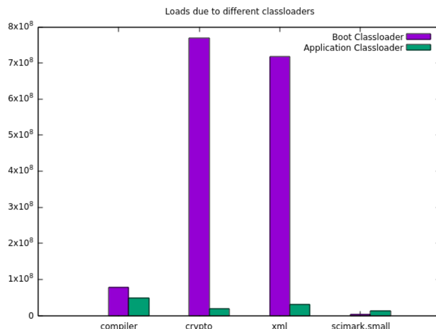
Tracking Boot Classes

- Classes like `java.util.*`, `java.lang.*` are expected to be used in any ordinary program.
- However, tracking them also causes a massive slowdown. Further, the MTP given above chooses not to use LGC on them.
- So, I wrote a small JavaAgent to compare usage.



Tracking Boot Classes

- Classes like `java.util.*`, `java.lang.*` are expected to be used in any ordinary program.
- However, tracking them also causes a massive slowdown. Further, the MTP given above chooses not to use LGC on them.
- So, I wrote a small JavaAgent to compare usage.



- **Parameters**: cycleSize, usageSkip, usageLimit, jsonSize, infixes

Other Issues

- **StackMaps**

Other Issues

- **StackMaps**
  - In the later versions of Java, a StackMap (maintaining type information) is required at every target of a jump.

Other Issues

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

Other Issues

- **StackMaps**
  - In the later versions of Java, a StackMap (maintaining type information) is required at every target of a jump.
  - It's hard to construct a StackMap by hand for our instrumented code, so we enforce using a Java 5.
- **Some Classes Cannot Be Tracked**
  - A small number of classes cannot be tracked, since they are critical parts of the class loading/tracking process.

Other Issues

- **StackMaps**
  - In the later versions of Java, a StackMap (maintaining type information) is required at every target of a jump.
  - It's hard to construct a StackMap by hand for our instrumented code, so we enforce using a Java 5.
- **Some Classes Cannot Be Tracked**
  - A small number of classes cannot be tracked, since they are critical parts of the class loading/tracking process.
- **Non-Existence of Hashcode**
  - Some boot classes seem to lack a `hashCode` method, which means we cannot uniquely identify them in a map unless we store the entire object (which would affect GC).

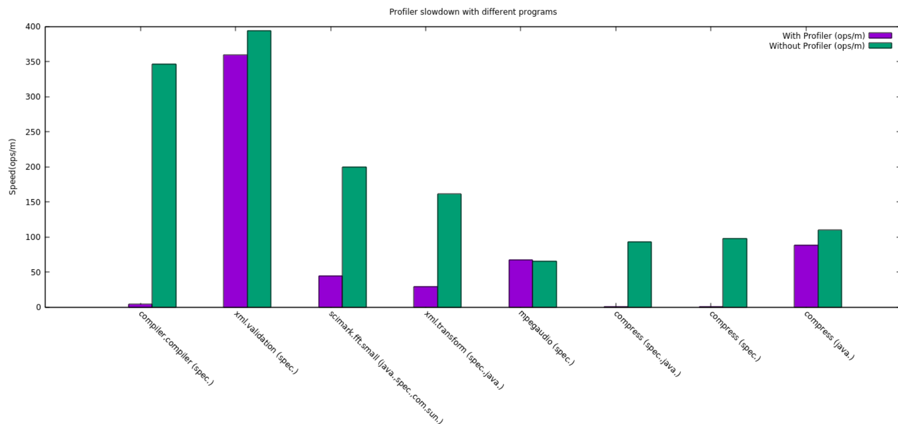
Other Issues

- **StackMaps**
  - In the later versions of Java, a StackMap (maintaining type information) is required at every target of a jump.
  - It's hard to construct a StackMap by hand for our instrumented code, so we enforce using a Java 5.
- **Some Classes Cannot Be Tracked**
  - A small number of classes cannot be tracked, since they are critical parts of the class loading/tracking process.
- **Non-Existence of Hashcode**
  - Some boot classes seem to lack a `hashCode` method, which means we cannot uniquely identify them in a map unless we store the entire object (which would affect GC).
  - Instead, we use `System.identityHashCode`.

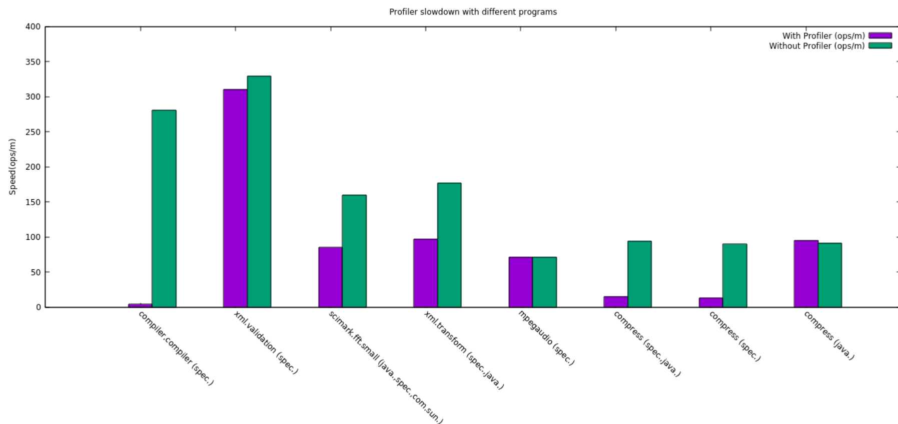
Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

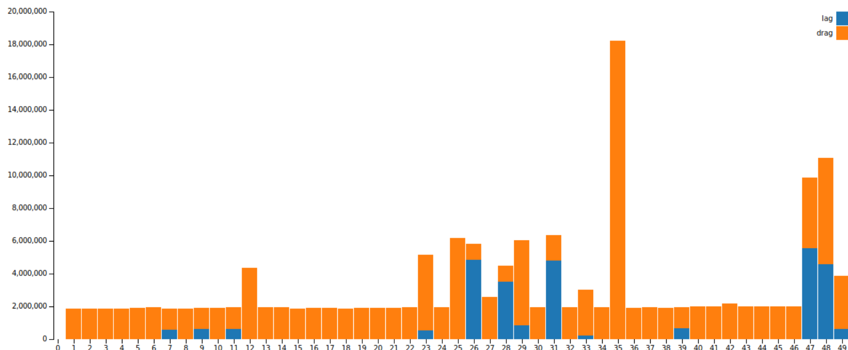
Profiler Speed



Profiler Speed



Visualization



Reference	Stack Trace
0	spec.benchmarks.compress.Decompressor$DeStack.(Compress.java:666) spec.benchmarks.compress.Decompressor.(Compress.java:522) spec.benchmarks.compress.Compress.performAction(Compress.java:80) spec.benchmarks.compress.Harness.runCompress(Harness.java:50) spec.benchmarks.compress.Harness.inst_main(Harness.java:63) spec.benchmarks.compress.Main.runBenchmark(Main.java:35) spec.benchmarks.compress.Main.harnessMain(Main.java:40) spec.harness.BenchmarkThread.runLoop(BenchmarkThread.java:170) spec.harness.BenchmarkThread.executeIteration(BenchmarkThread.java:132) spec.harness.BenchmarkThread.run(BenchmarkThread.java:59)

Questions

Any questions?

Table of Contents

1 Previous Work
- Overview
- Problems
- Possible Solutions

2 Implementation
- Overview
- Metric
- Details
- Problems
- Results

3 Future Work

4 Bonus Content

Completing the Compiler

- Currently, the profiler dumps information into a JSON file.

Completing the Compiler

- Currently, the profiler dumps information into a JSON file.
- We would like to use this data to annotate the original source file, to mark objects for LGC or RGC.

Completing the Compiler

- Currently, the profiler dumps information into a JSON file.
- We would like to use this data to annotate the original source file, to mark objects for LGC or RGC.
- A problem with this is those object graphs where an LGC node has an RGC parent.

Completing the Compiler

- Currently, the profiler dumps information into a JSON file.
- We would like to use this data to annotate the original source file, to mark objects for LGC or RGC.
- A problem with this is those object graphs where an LGC node has an RGC parent.
- These LGC nodes won't be visited more than once!

Completing the Compiler

- Currently, the profiler dumps information into a JSON file.
- We would like to use this data to annotate the original source file, to mark objects for LGC or RGC.
- A problem with this is those object graphs where an LGC node has an RGC parent.
- These LGC nodes won't be visited more than once!
- One possible solution is that we mark every ancestor of an LGC node LGC.

Completing the Compiler

- Currently, the profiler dumps information into a JSON file.
- We would like to use this data to annotate the original source file, to mark objects for LGC or RGC.
- A problem with this is those object graphs where an LGC node has an RGC parent.
- These LGC nodes won't be visited more than once!
- One possible solution is that we mark every ancestor of an LGC node LGC.
- Another possible solution is marking objects as LGC/RGC based on sum of drag of their children.

Other Future Work

- **Improvements**

Other Future Work

- **Improvements**
  - Generating StackMaps manually so that this can be used with Java 6 onwards.

Other Future Work

- **Improvements**
  - Generating StackMaps manually so that this can be used with Java 6 onwards.
  - Using JVMTI (C++) instead of a JavaAgent, for a possible speedup in profiling.

Other Future Work

- **Improvements**
  - Generating StackMaps manually so that this can be used with Java 6 onwards.
  - Using JVMTI (C++) instead of a JavaAgent, for a possible speedup in profiling.
- **Other Avenues to LGC** We've described several ways to use LGC practically, and explored only one. The others can also be explored.

Table of Contents

1. Previous Work
   - Overview
   - Problems
   - Possible Solutions
2. Implementation
   - Overview
   - Metric
   - Details
   - Problems
   - Results
3. Future Work
4. Bonus Content

A Memory Saving Browser Extension

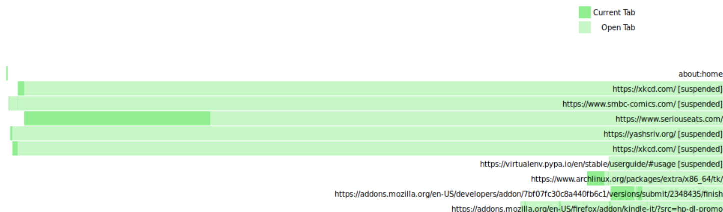
- During the course of this UGP, we also came up with a browser extension which profiles tab usage, and suspends tabs to save memory.

A Memory Saving 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.

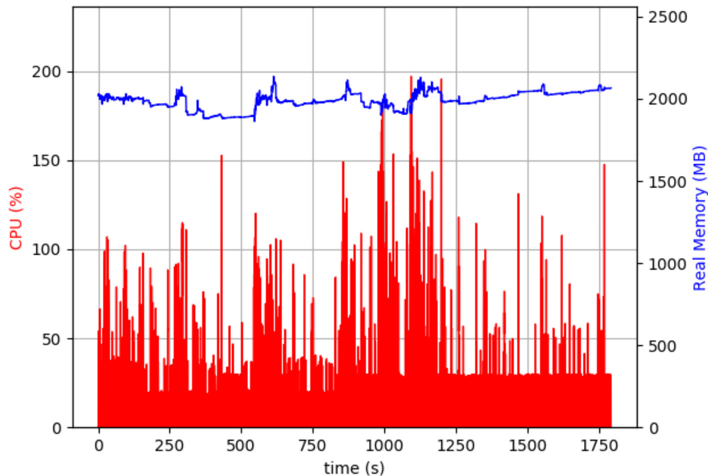
A Memory Saving Browser Extension

- 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.
- **Extension Profiler Results**



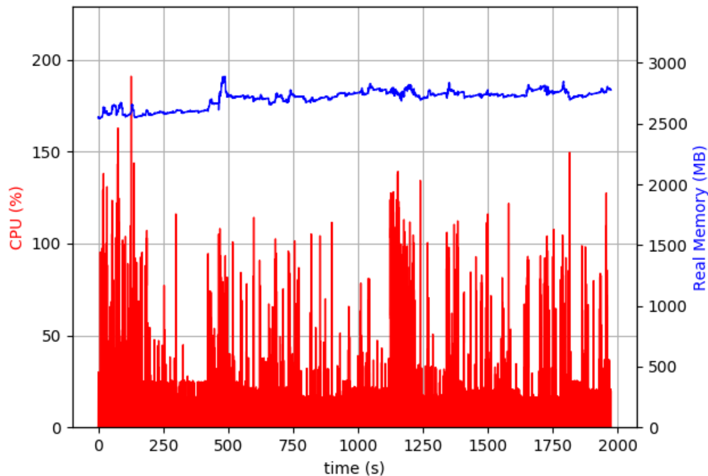
A Memory Saving Browser Extension

Usage with extension, over 30min and 50 tabs. 750MB saved.



A Memory Saving Browser Extension

Usage without extension, over 30min and 50 tabs.



Thank You!

- Weekly logs, more detailed technical documentation
- Code, private repository, access on request
- Browser Extension