A low-latency garbage collector for GHC

Ben Gamari   Laura Dietz

Well-Typed

Motivation

```
$ ghc -threaded EditDist.hs
$ ./EditDist +RTS -s

...

  16,168,836,784 bytes allocated in the heap
   5,417,286,976 bytes copied during GC
   1,745,510,392 bytes maximum residency (13 sample(s))
       3,260,424 bytes maximum slop
            3416 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     15520 colls,     0 par    1.695s   1.702s     0.0001s    0.0010s
  Gen  1        13 colls,     0 par    7.320s   7.328s     0.5637s    3.5480s

...
```

The cause of the pause…

Garbage collection in GHC's existing collector:

- performs $O(\mathrm{live\ heap\ size})$ work during major collection
- stops program execution for entirety of collection

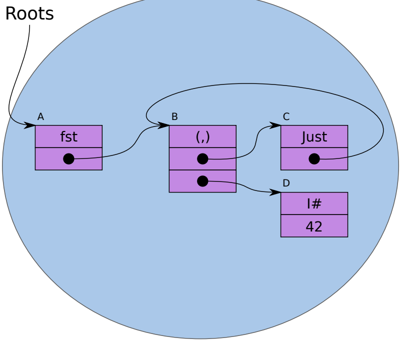
The cause of the pause...

Garbage collection in GHC's existing collector:

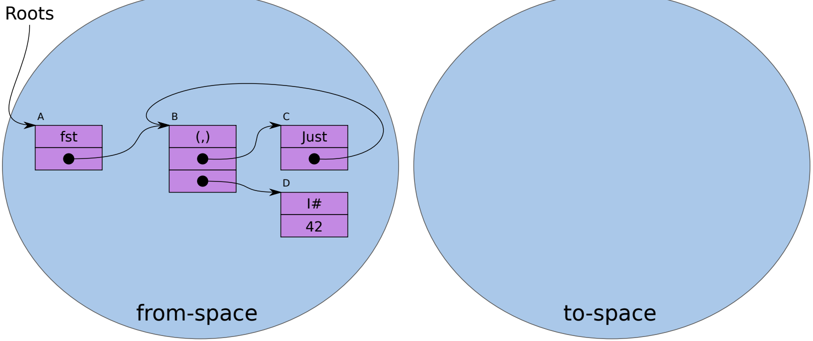
- performs $O(\text{live heap size})$ work during major collection
- stops program execution for entirety of collection

Copying GC remarkably difficult to incrementalize.

Copying garbage collection

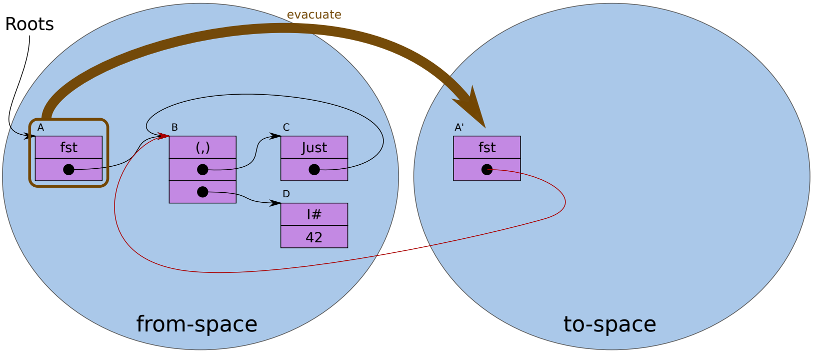


Copying garbage collection



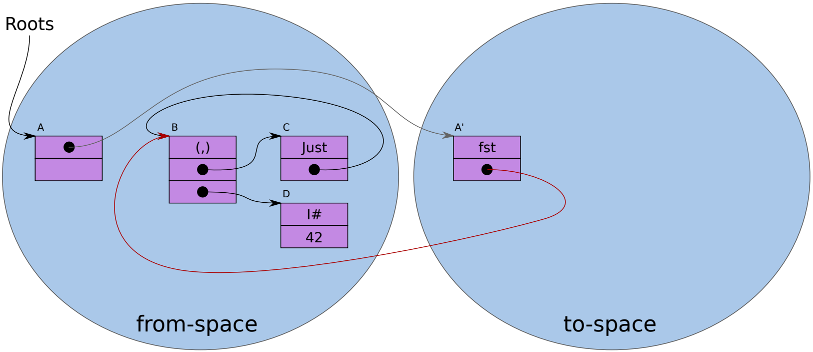
Well-Typed

Copying garbage collection

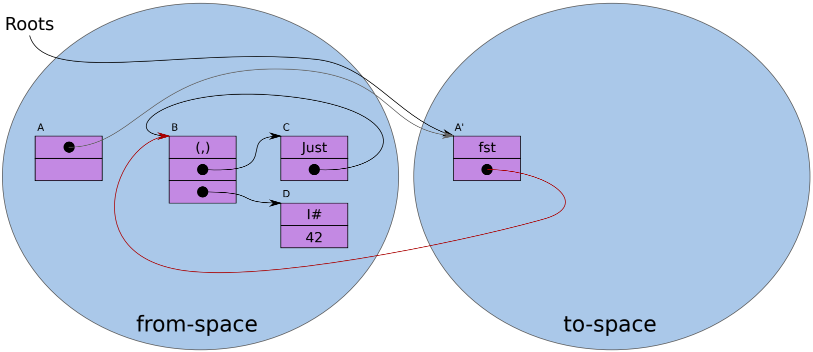


Well-Typed

Copying garbage collection

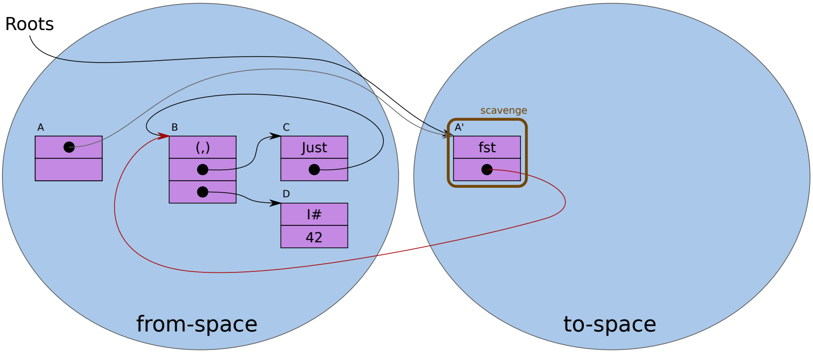


Copying garbage collection



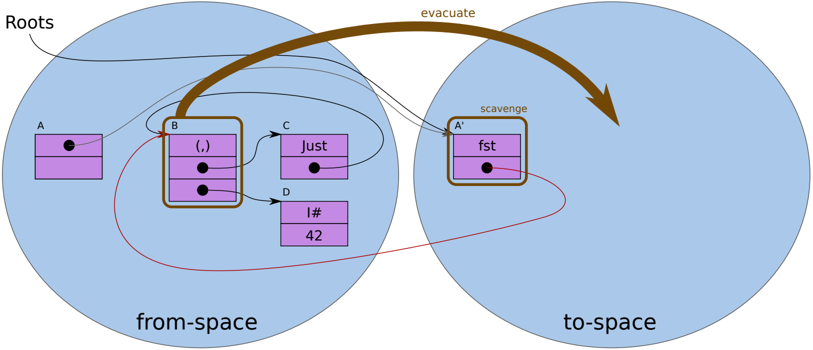
Well-Typed

Copying garbage collection



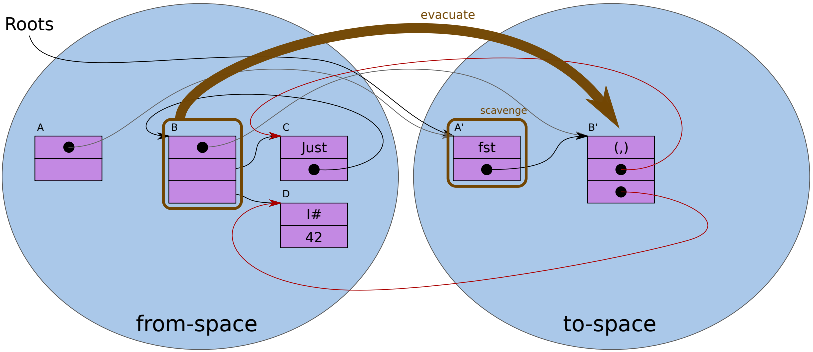
Well-Typed

Copying garbage collection

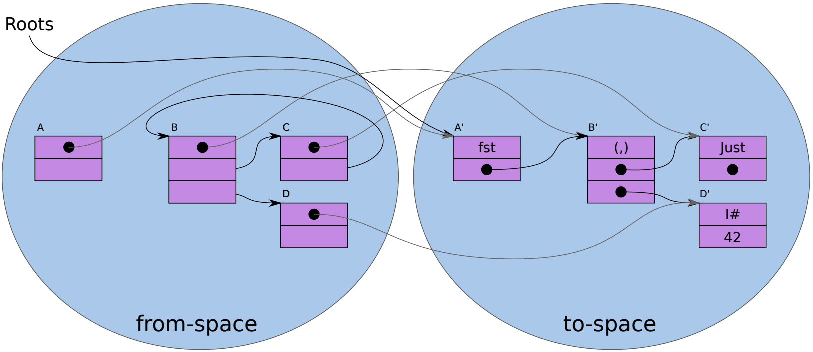


Well-Typed

Copying garbage collection

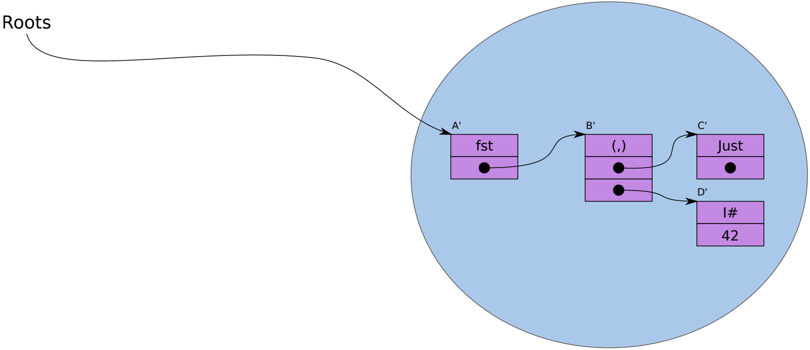


Copying garbage collection



Well-Typed

Copying garbage collection



Well-Typed

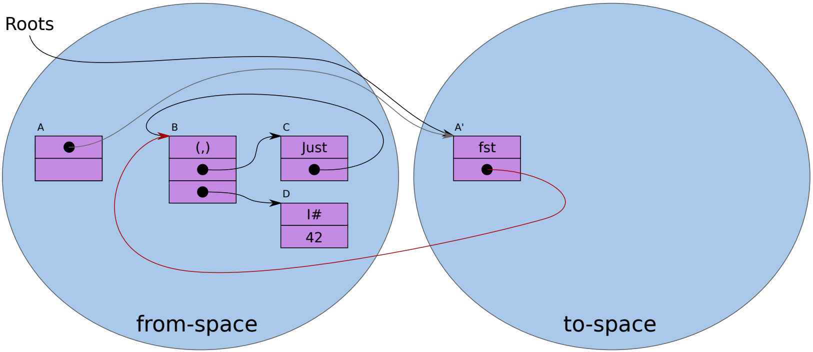
Copying garbage collection

Benefits:

- Cheap allocation
- Efficient: Scavenging has excellent locality
- Compacting: Avoids fragmenting heap over successive collections
- Easily implemented, parallelized

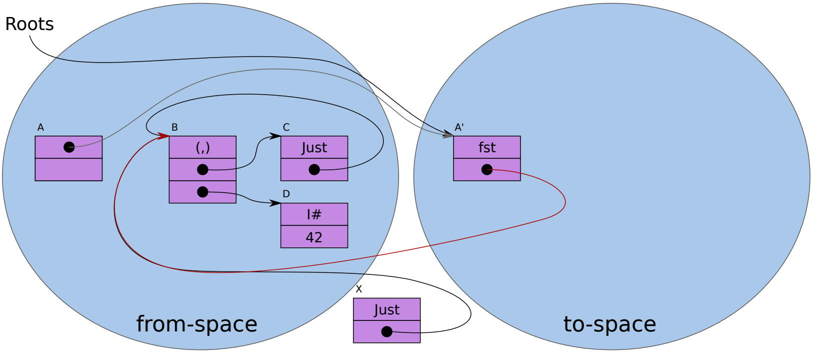
However, hard to perform without stop-the-world pause.

The challenge of copying collection

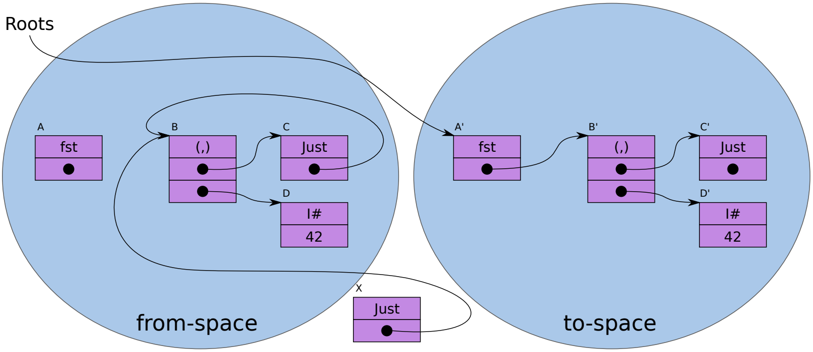


Well-Typed

The challenge of copying collection

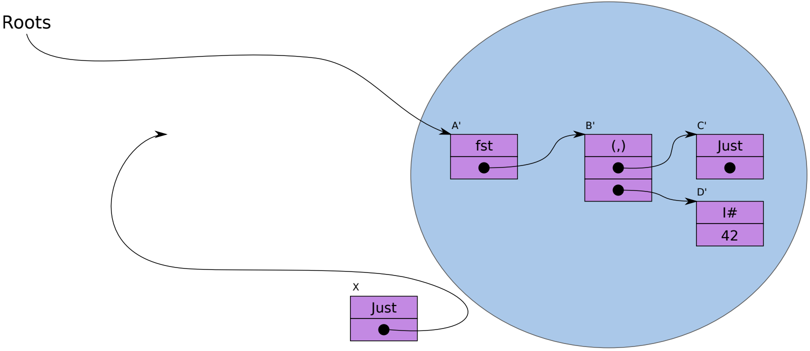


The challenge of copying collection



Well-Typed

The challenge of copying collection



Well-Typed

A new collector design

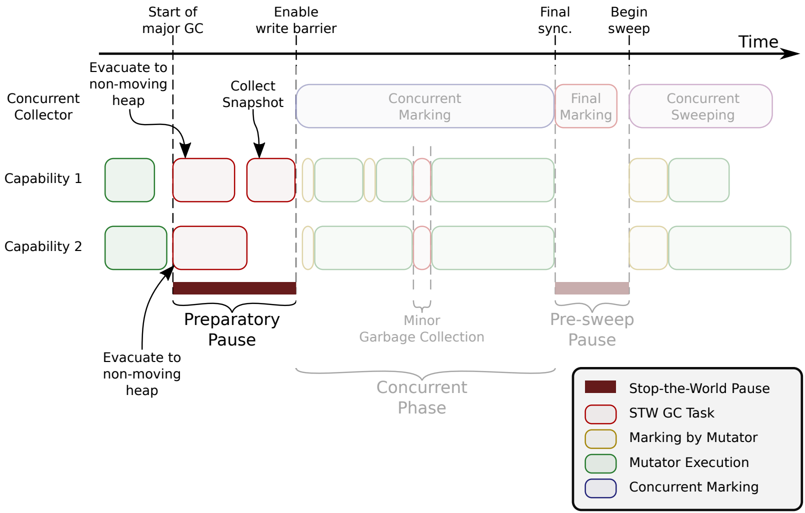
Generational collector:

- Retain moving collection for (bounded-size) young generations
- Non-moving heap with mark & sweep collection for oldest generation

Eliminates long pauses:

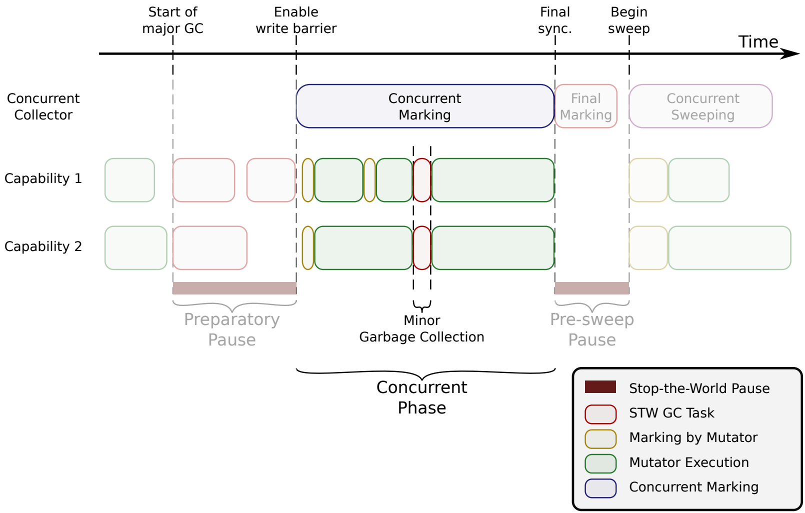
- Young generations: STW collection with bounded duration
- Oldest generation: concurrent collection

Garbage Collection Lifecycle



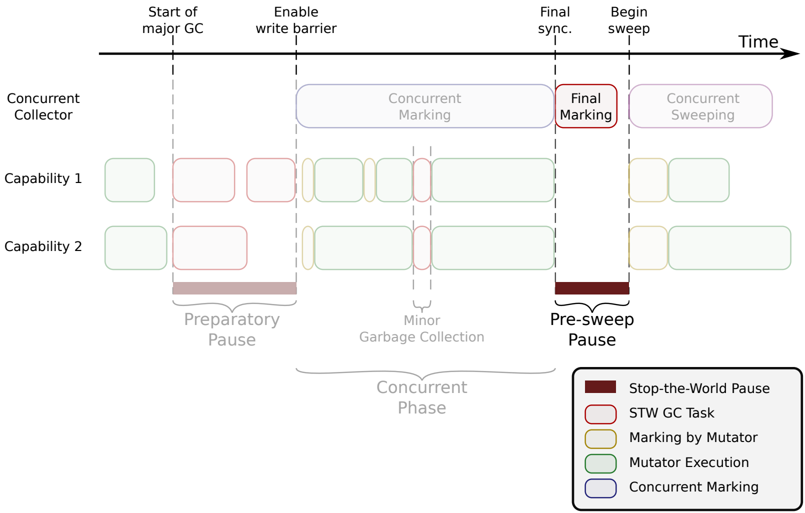
Well-Typed

Garbage Collection Lifecycle



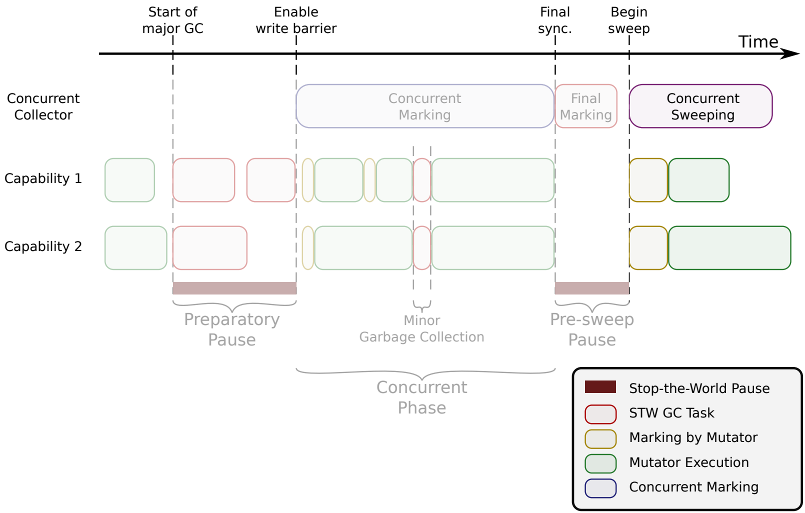
Well-Typed

Garbage Collection Lifecycle



Well-Typed

Garbage Collection Lifecycle



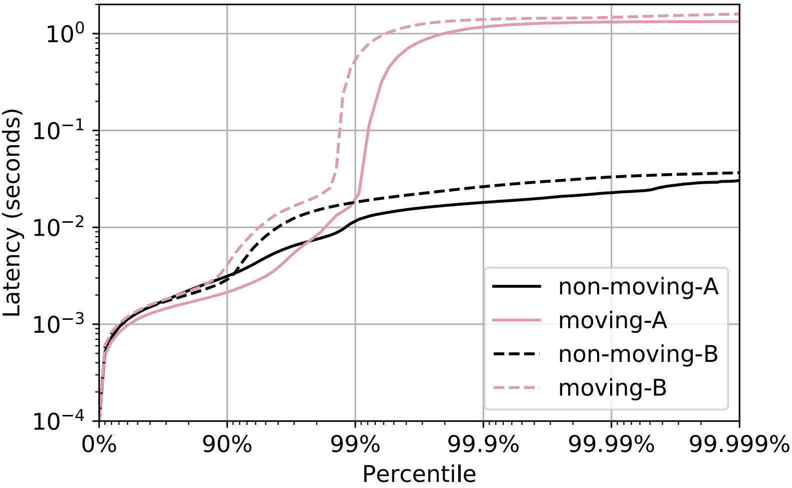
Well-Typed

How to use it?

Build program with `-threaded`, run with `+RTS --nonmoving-gc`:

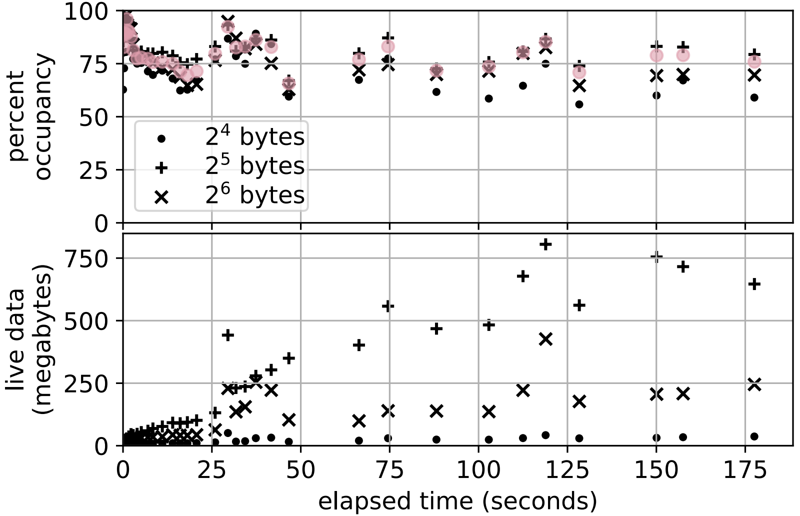
```
$ ghc -threaded EditDist.hs
$ ./EditDist +RTS -s --nonmoving-gc

...

  16,168,831,672 bytes allocated in the heap
   1,871,197,048 bytes copied during GC
   1,962,037,024 bytes maximum residency (10 sample(s))
     489,828,576 bytes maximum slop
            3001 MiB total memory in use (9 MB lost due to fragmentation)

                                      Tot time (elapsed)  Avg pause  Max pause
  Gen  0     15523 colls,     0 par    2.946s   2.962s     0.0002s    0.0040s
  Gen  1        10 colls,     0 par    0.004s   0.004s     0.0004s    0.0014s
  Gen  1        10 syncs,                       0.001s     0.0001s    0.0004s
  Gen  1       concurrent,             2.940s   5.888s     0.5888s    3.7217s

...
```

Benchmarks: Response time



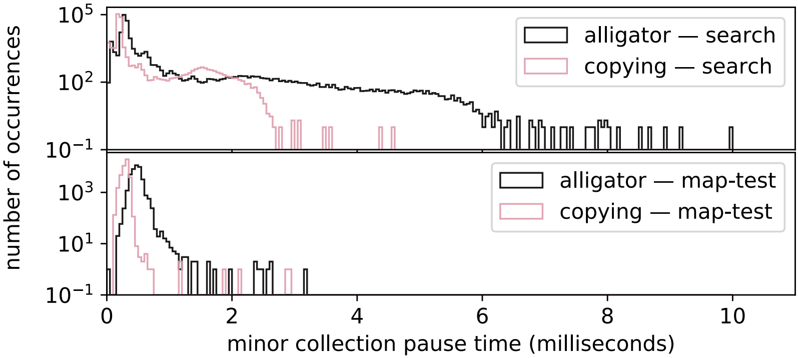
Well-Typed

Benchmarks: How much memory is lost to fragmentation?



- Roughly 25% steady-state storage overhead due to fragmentation and overhead.

Benchmarks: Allocation cost



- Allocation cost increases, particularly with fragmentation
- Manifests as longer minor GCs

What can you expect?

- Much lower latencies for most programs (major collections comparable to minor)
  - Especially in the tail
- Throughput reduction around 10%
  - Due to locality, write barrier overhead
- Memory footprint: increase of between 10% and 25%
  - Due to allocation overhead, conservative marking
- Other things to keep in mind:
  - Unsafe foreign calls can introduce pauses

Future work

- Optimization:
  - Parallel marking
  - Use address-space partitioning to reduce cost of generation checks
  - Improve allocator bitmap representation to lower allocation cost
- Pause reduction:
  - Abort final synchronization on long pre-sweep pause; back-pressure
  - Tune promotion heuristics to
- Allocation of pinned objects directly into non-moving heap
  - Addresses problem of fragmentation due to pinned objects

Summary

Questions?

For further implementation details see our paper at ISMM 2020.

Email: ben@well-typed.com