A low-latency garbage collector for GHC

Ben Gamari    Laura Dietz
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)

<table>
<thead>
<tr>
<th>Tot time (elapsed)</th>
<th>Gen 0</th>
<th>Gen 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15520</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>0 par</td>
<td>0 par</td>
</tr>
<tr>
<td>1.695s 1.702s 0.0001s 0.0010s</td>
<td>7.320s 7.328s 0.5637s 3.5480s</td>
<td></td>
</tr>
</tbody>
</table>

...
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
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

Well-Typed
Copying garbage collection
Copying garbage collection

Well-Typed

```plaintext
A
B
C
D
A'

Roots

from-space

to-space

fst
Just
I#
L# 42

fst

(,)

Well-Typed
```
Copying garbage collection
Copying garbage collection

from-space

A

B

(,)

C

Just

D

I#

42

to-space

Roots

A'

fst

A'

Well-Typed
Copying garbage collection

```
Well-Typed
```

```
A
B
C
D

fst
Just
I#
42

Roots

from-space

A'

Scavenge

to-space

Well-Typed
```
Copying garbage collection

Well-Typed
Copying garbage collection

Roots

from-space

A
B
C
D

Just
I#
42

to-space

A'
B'
fst
(,

Well-Typed

Well-Typed
Copying garbage collection

Well-Typed
Copying garbage collection

Roots

Well-Typed
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

Roots

from-space

A
(,)
B
Just
C
I#
42
D

Roots

to-space

A'

Well-Typed
The challenge of copying collection

Well-Typed
The challenge of copying collection

from-space

 Roots

 to-space
The challenge of copying collection
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

- Start of major GC
- Enable write barrier
- Final sync.
- Begin sweep

Concurrent Collector

Capability 1

Capability 2

Preparatory Pause

Minor Garbage Collection

Pre-sweep Pause

Concurrent Phase

Stop-the-World Pause

STW GC Task

Marking by Mutator

Mutator Execution

Concurrent Marking
Garbage Collection Lifecycle

- **Start of major GC**
- **Enable write barrier**
- **Final sync.**
- **Begin sweep**

- **Concurrent Collector**
  - Concurrent Marking
  - Concurrent Sweeping

- **Capability 1**
  - Preparatory Pause

- **Capability 2**
  - Minor Garbage Collection
  - Pre-sweep Pause

- **Stop-the-World Pause**
- **STW GC Task**
- **Marking by Mutator**
- **Mutator Execution**
- **Concurrent Marking**

Well-Typed
Garbage Collection Lifecycle

- **Start of major GC**
- **Enable write barrier**
- **Final sync.**
- **Begin sweep**

**Concurrent Collector**
- **Concurrent Marking**
- **Final Marking**
- **Concurrent Sweeping**

**Capability 1**
- **Preparatory Pause**
- **Minor Garbage Collection**

**Capability 2**
- **Pre-sweep Pause**

Legend:
- Stop-the-World Pause
- STW GC Task
- Marking by Mutator
- Mutator Execution
- Concurrent Marking

Well-Typed
How to use it?

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

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

...
Benchmarks: Response time

The graph compares the response times of different scenarios. The y-axis represents latency (seconds) on a logarithmic scale, ranging from $10^{-4}$ to $10^0$. The x-axis shows the percentile, ranging from 0% to 99.9999%. Four scenarios are compared:

- **non-moving-A** (solid black line)
- **moving-A** (solid pink line)
- **non-moving-B** (dashed black line)
- **moving-B** (dashed pink line)

The graph illustrates how each scenario varies with respect to the percentile, indicating the distribution of response times.
Benchmarks: How much memory is lost to fragmentation?

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

For further implementation details see our paper at ISMM 2020.

Email: ben@well-typed.com