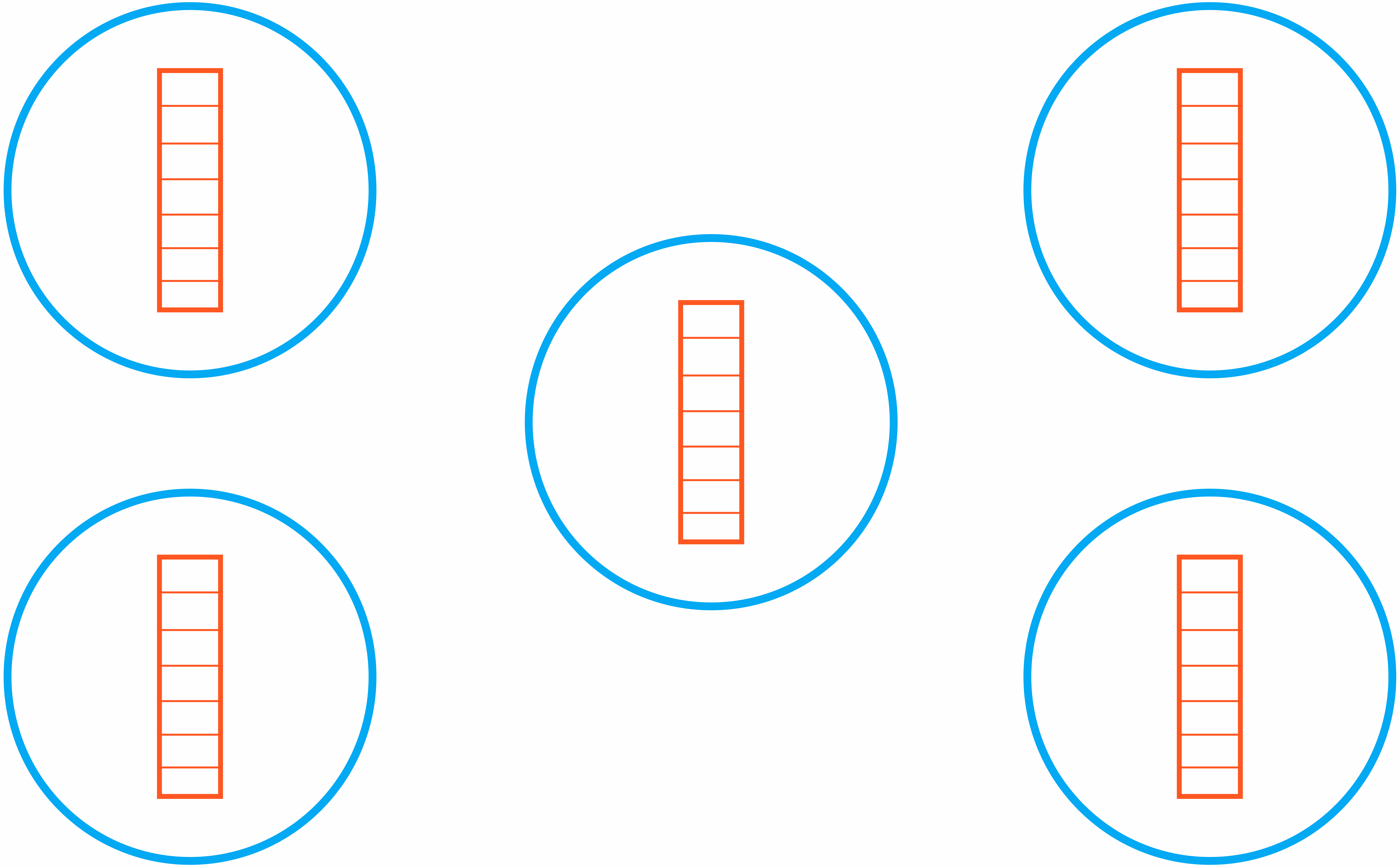
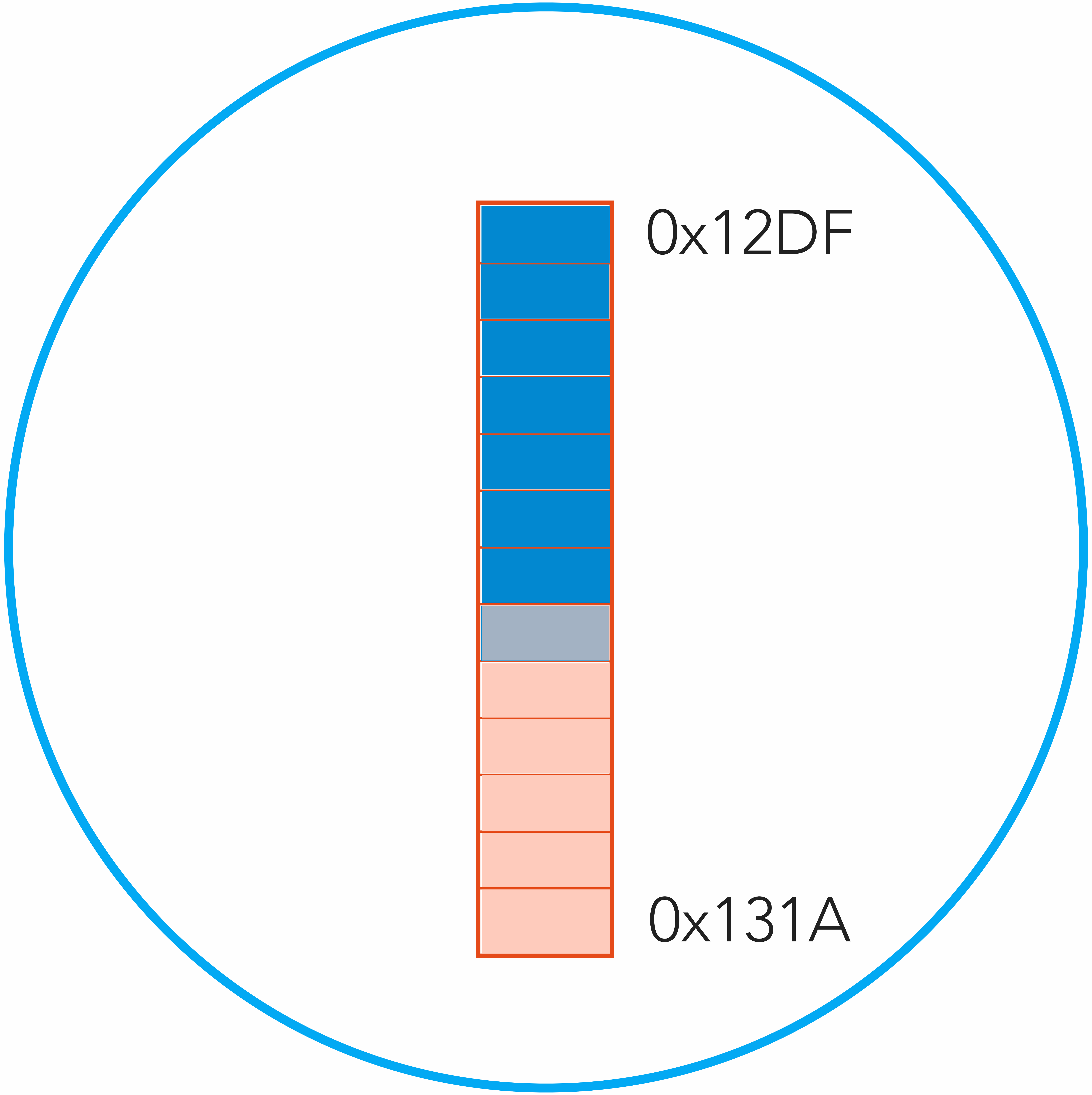
DIGGING THROUGH THE GARBAGE

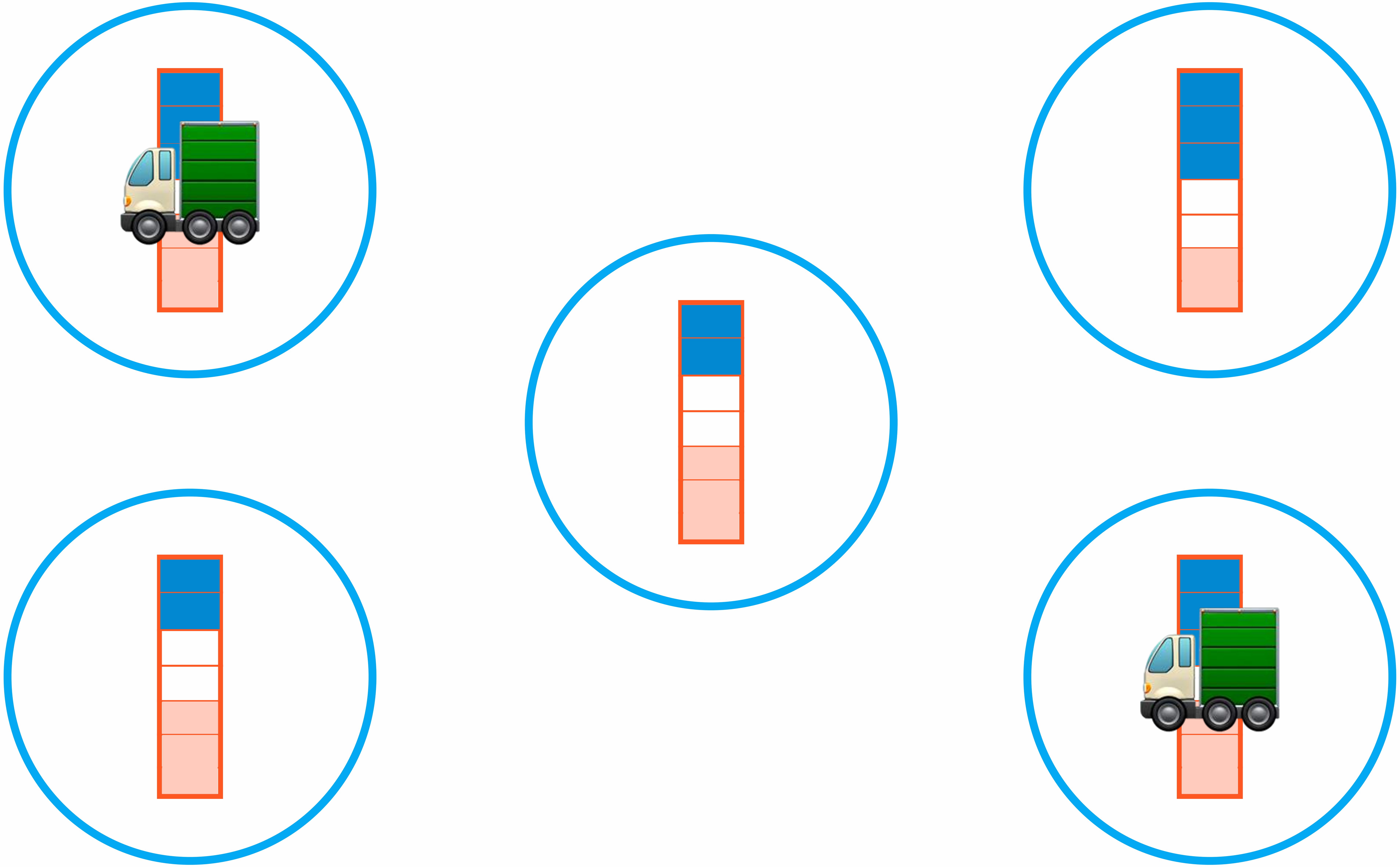
FURY
HYPERX
DOMINATOR PLATINUM
BALLISTIX
VENGEANCE
VENGEANCE LPX
DDR4
RIPJAWS V
NIGHT HAWK
VENGEANCE RGB
FURY
HYPERX
DDR4
RGB
RGB
DARK PRO
G.SKILL

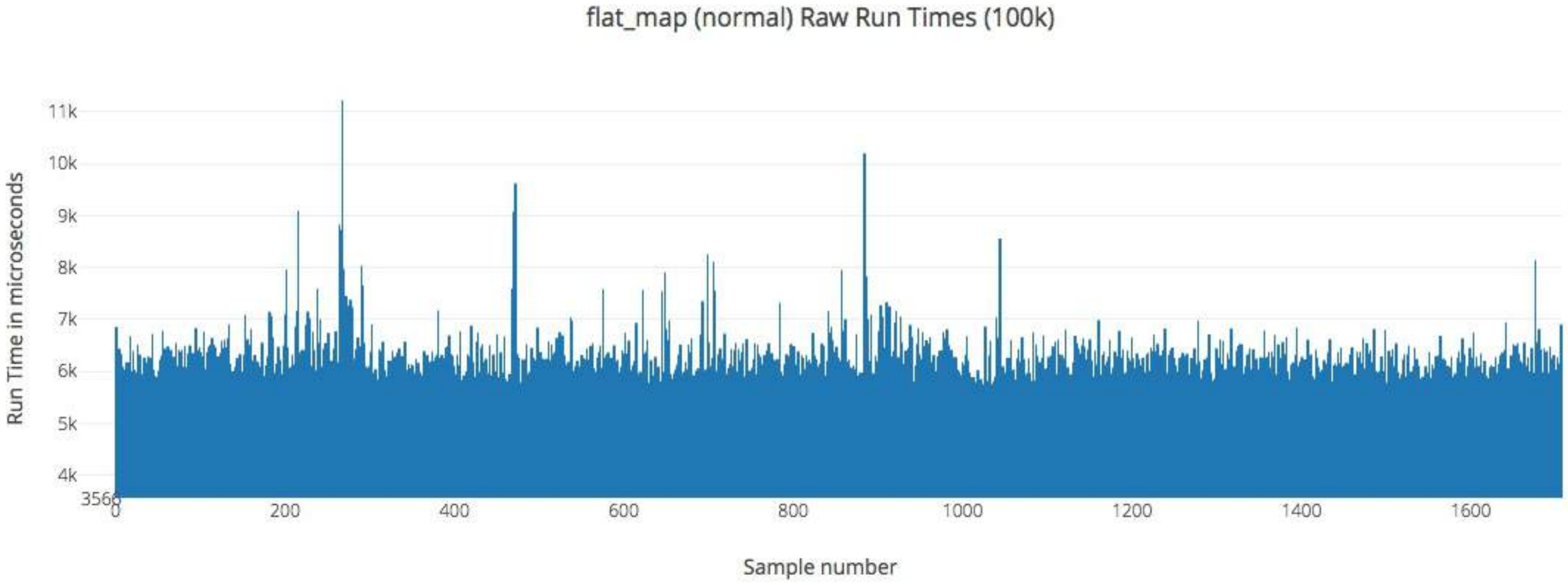
```
GC.disable()
starting_memory = Process.get_memory(self())
fun.()
ending_memory = Process.get_memory(self())
total = ending_memory - starting_memory
GC.enable()
```




0x12DF
0x131A

```
starting_memory = :erlang.memory
fun.()
ending_memory = :erlang.memory
total = ending_memory - starting_memory
```




flat_map (normal) Raw Run Times (100k)
Run Time in microseconds
11k
10k
9k
8k
7k
6k
5k
4k
3568
0
200
400
600
800
1000
1200
1400
1600
Sample number

```
spawn_opt(test, test, [], [{min_heap_size, 65535}]).

Process.spawn(Test, :test, [], min_heap_size: 65535)
```

Packard Bell

486DX2/66MHz Multimedia Computer

- 720 MB Hard Drive
- 8 MB Memory
- 1 MB Video Memory
- Double-Speed CD-ROM Drive
- 16-Bit Sound Card and Speakers
- Internal Fax/Modem
- Telephone Answering System (24CD)

14" Super VGA Non-Interlaced Monitor (1411SL) $287.87

Includes Over $800 Worth of Software

Windows for Workgroups, MS Money, Prodigy, MS Productivity Pack, MS Entertainment Pack, MS Works, New Grolier Encyclopedia, 3D Dinosaur Adventure, Undersea Adventure, Space Adventure, Speed, World Atlas, Sports Illustrated Almanac and More!

o Payments

erest† Until April 1995

uired. No finance charge if paid in full within 6 months from the date of purchase
hin 6 mont... Offer is for individuals, not businesses. Offer ends October 29, 1994.

$1798

Monitor Sold Separately

```
defmodule Test do
  def test do
    IO.inspect(Process.info(self(), :garbage_collection_info))









  end
end
```

```
defmodule Test do
  def test do
    IO.inspect(Process.info(self(), :garbage_collection_info))
    Enum.map((1..100000), fn num ->
      {:ok, num}
    end)







  end
end
```

```
defmodule Test do
  def test do
    IO.inspect(Process.info(self(), :garbage_collection_info))
    Enum.map((1..100000), fn num ->
      {:ok, num}
    end)
    IO.inspect(Process.info(self(), :garbage_collection_info))





  end
end
```

```
defmodule Test do
  def test do
    IO.inspect(Process.info(self(), :garbage_collection_info))
    Enum.map((1..100000), fn num ->
      {:ok, num}
    end)
    IO.inspect(Process.info(self(), :garbage_collection_info))
    :erlang.garbage_collect()




  end
end
```

```
defmodule Test do
  def test do
    IO.inspect(Process.info(self(), :garbage_collection_info))
    Enum.map((1..100000), fn num ->
      {:ok, num}
    end)
    IO.inspect(Process.info(self(), :garbage_collection_info))
    :erlang.garbage_collect()
    for num <- (1..100000) do
      {:ok, num}
    end

 end
end
```

```
defmodule Test do
  def test do
    IO.inspect(Process.info(self(), :garbage_collection_info))
    Enum.map((1..100000), fn num ->
      {:ok, num}
    end)
    IO.inspect(Process.info(self(), :garbage_collection_info))
    :erlang.garbage_collect()
    for num <- (1..100000) do
      {:ok, num}
    end
    IO.inspect(Process.info(self(), :garbage_collection_info))
  end
end
```

```
iex(1)> Process.spawn(Test, :test, [], min_heap_size: 65535)
{:garbage_collection_info,
 [
   # …
   heap_block_size: 75113,
   recent_size: 0,
   heap_size: 75111,
   # …
 ]}
{:garbage_collection_info,
 [
   # …
   heap_block_size: 833026,
   recent_size: 500000,
   heap_size: 833024,
   # …
 ]}
{:garbage_collection_info,
 [
   # …
   heap_block_size: 75113,
   recent_size: 6,
   heap_size: 75111,
   # …
 ]}
```

```
iex(1)> Process.spawn(Test, :test, [], min_heap_size: 65535)
{:garbage_collection_info,
 [
   # …
   heap_block_size: 75113,
   recent_size: 0,
   heap_size: 75111,
   # …
 ]}
{:garbage_collection_info,
 [
   # …
   heap_block_size: 833026,
   recent_size: 500000,
   heap_size: 833024,
   # …
 ]}
{:garbage_collection_info,
 [
   # …
   heap_block_size: 75113,
   recent_size: 6,
   heap_size: 75111,
   # …
 ]}
```

```
iex(1)> Process.spawn(Test, :test, [], min_heap_size: 65535)
{:garbage_collection_info,
 [
   # …
   heap_block_size: 75113,
   recent_size: 0,
   heap_size: 75111,
   # …
 ]}
{:garbage_collection_info,
 [
   # …
   heap_block_size: 833026,
   recent_size: 500000,
   heap_size: 833024,
   # …
 ]}
{:garbage_collection_info,
 [
   # …
   heap_block_size: 75113,
   recent_size: 6,
   heap_size: 75111,
   # …
 ]}
```

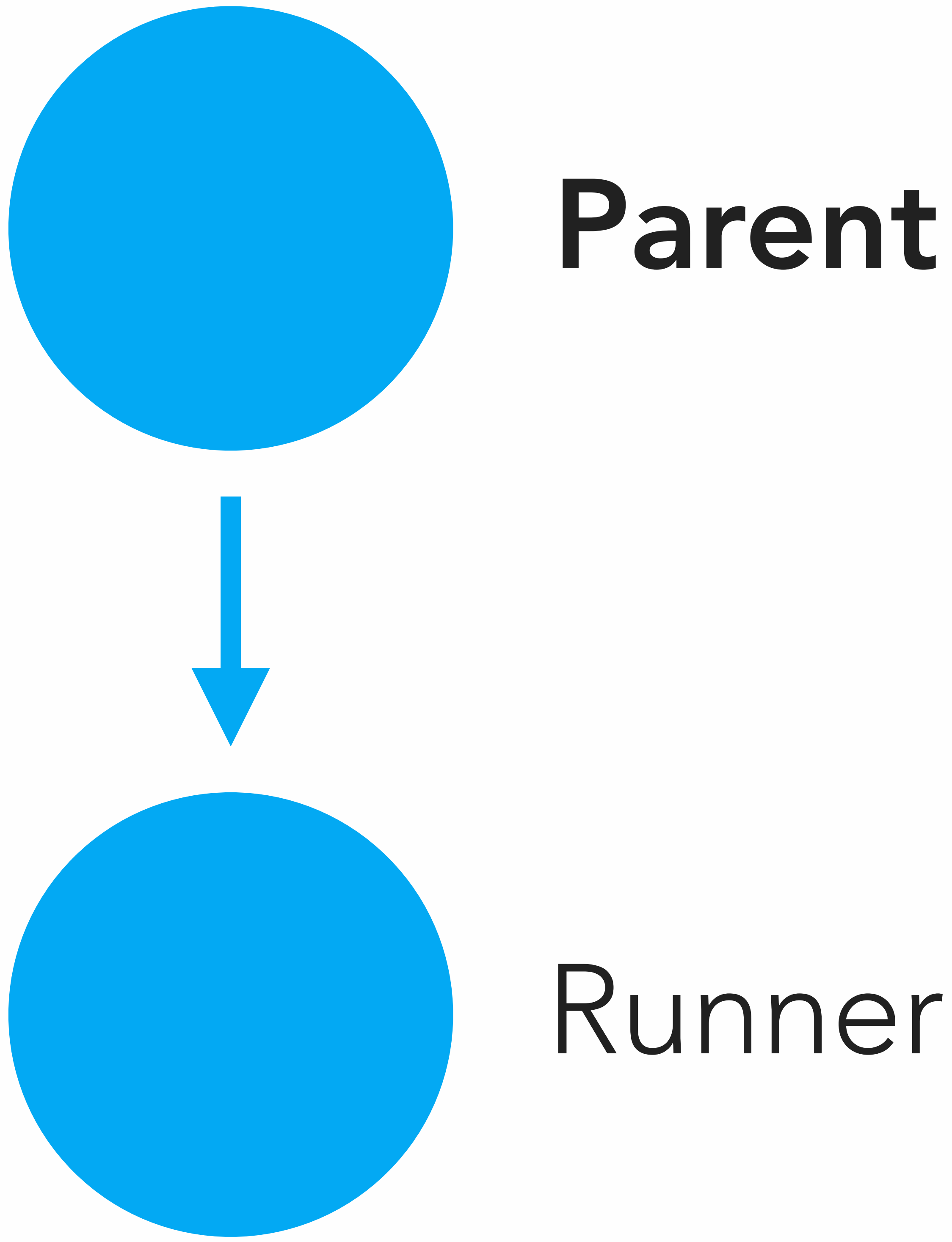
```
iex(1)> Process.spawn(Test, :test, [], min_heap_size: 83324000000000)
beam.smp(87504,0xb053f000) malloc: *** mach_vm_map(size=794757463801856) failed (error
code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
               beam.smp(87504,0xb053f000) malloc: *** mach_vm_map(size=794757463801856)
failed (error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
                              beam.smp(87504,0xb053f000) malloc: ***
mach_vm_map(size=794757462790144) failed (error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
beam.smp(87504,0xb053f000) malloc: *** mach_vm_map(size=794757462790144) failed (error
code=3)
*** error: can't allocate region
         *** set a breakpoint in malloc_error_break to debug
                                                            eheap_alloc: Cannot allocate
794757462787016 bytes of memory (of type "heap").
Crash dump is being written to: erl_crash.dump...done
```

CodeBeam Lite Berlin 2018

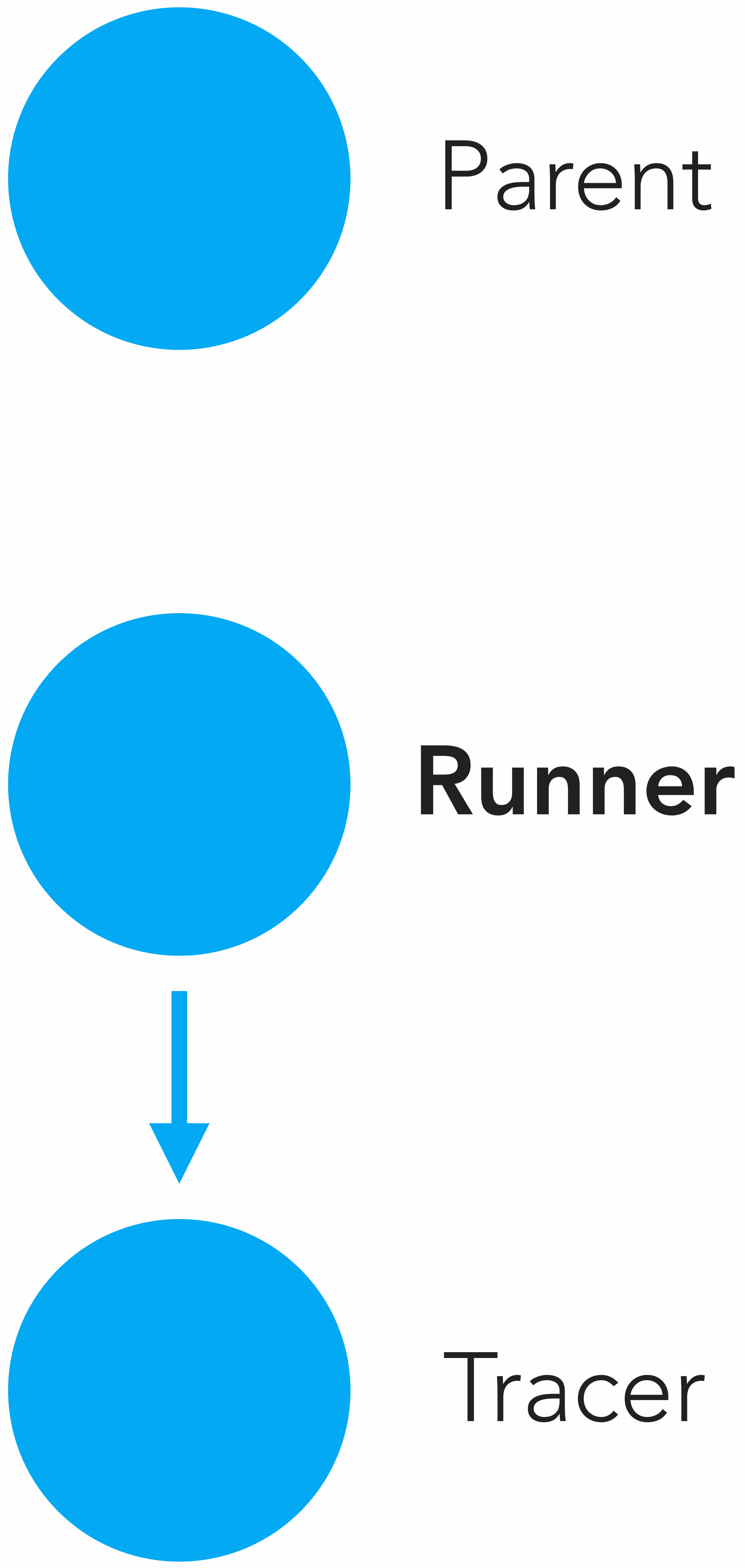
```
defmodule Benchee.MemoryMeasure do















end
```

```
defmodule Benchee.MemoryMeasure do
  def measure(function) do
  end
















end
```

```
defmodule Benchee.MemoryMeasure do
  def measure(function) do
  end

  defp start_runner(function, ref) do
  end




end
```



```
defmodule Benchee.MemoryMeasure do
  def measure(function) do
  end

  defp start_runner(function, ref) do
  end







  defp start_tracer(runner_pid) do
  end






end
```



```
defmodule Benchee.MemoryMeasure do
  def measure(function) do
  end

  defp start_runner(function, ref) do
  end









  defp start_tracer(runner_pid) do
  end

  defp tracer_loop(runner_pid, acc) do
  end

  defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  end
end
```

Parent

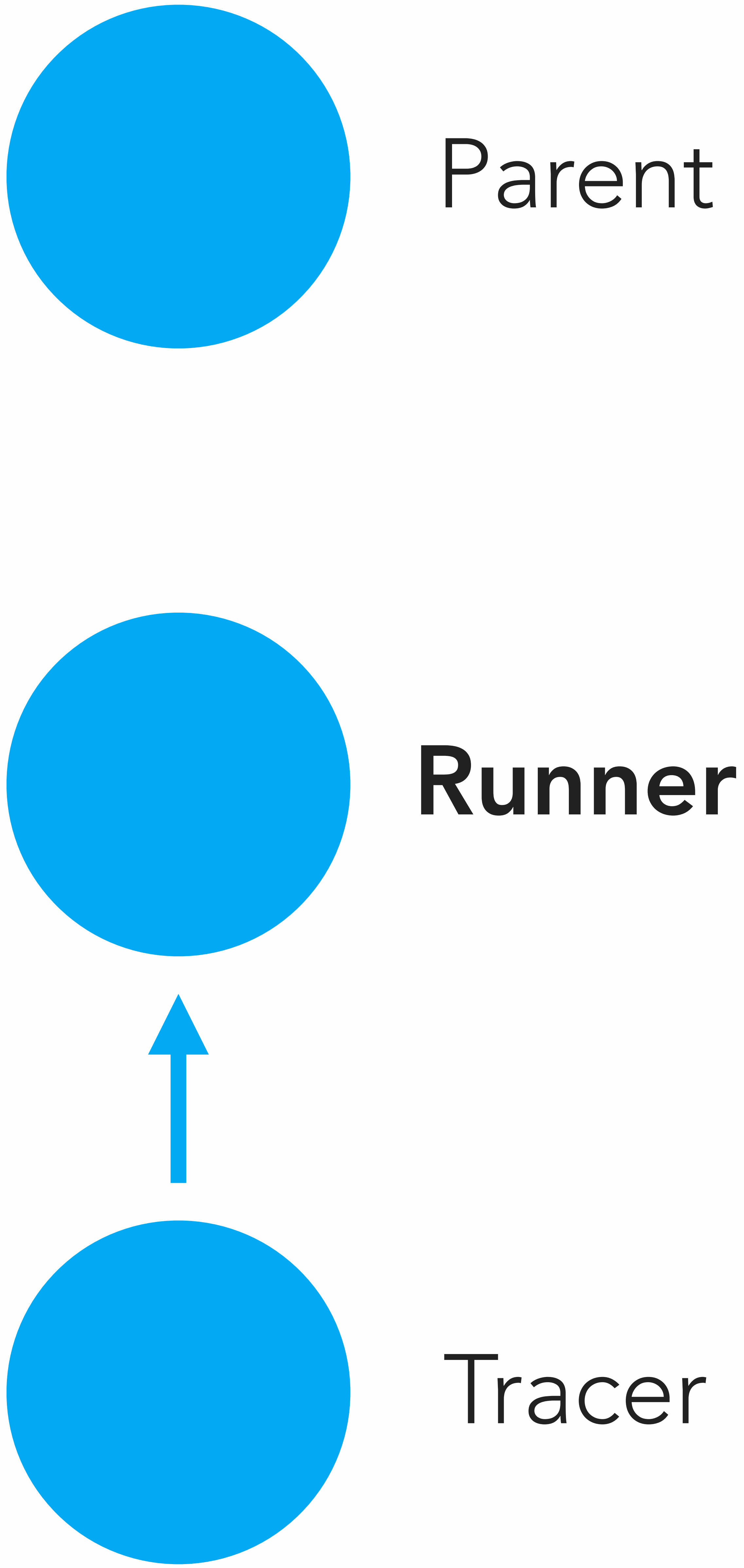
Runner

Tracer

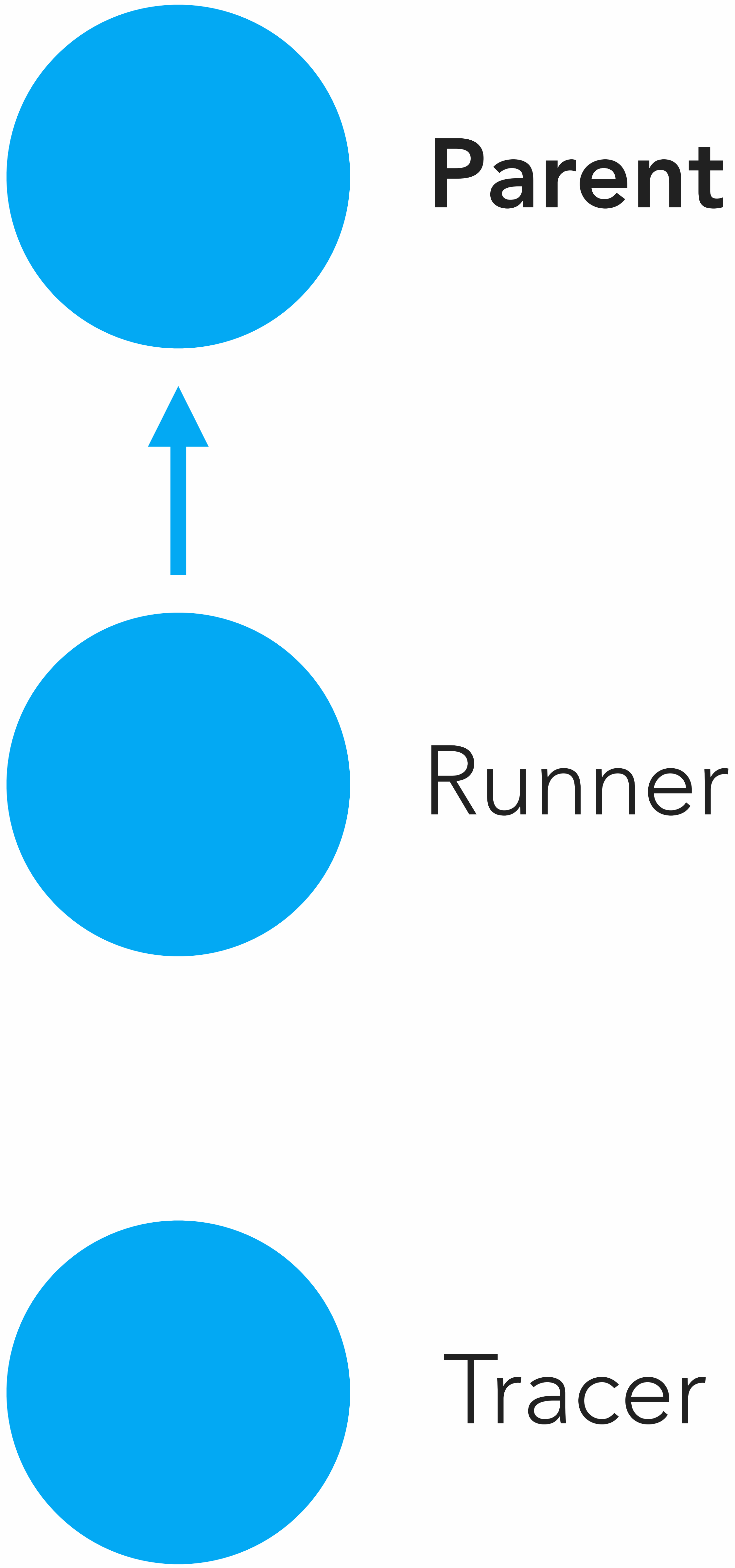
```
defmodule Benchee.MemoryMeasure do
  def measure(function) do
  end

  defp start_runner(function, ref) do
  end

  defp measure_memory(function, tracer_pid) do
  end




  defp start_tracer(runner_pid) do
  end

  defp tracer_loop(runner_pid, acc) do
  end

  defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  end
end
```

Parent
Runner
Tracer

```
defmodule Benchee.MemoryMeasure do
  def measure(function) do
  end

  defp start_runner(function, ref) do
  end

  defp measure_memory(function, tracer_pid) do
  end

  defp get_collected_memory(tracer_pid) do
  end

  defp start_tracer(runner_pid) do
  end

  defp tracer_loop(runner_pid, acc) do
  end

  defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  end
end
```



```
defmodule Benchee.MemoryMeasure do
  def measure(function) do
  end

  defp start_runner(function, ref) do
  end

  defp measure_memory(function, tracer_pid) do
  end

  defp get_collected_memory(tracer_pid) do
  end

  defp start_tracer(runner_pid) do
  end

  defp tracer_loop(runner_pid, acc) do
  end

  defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  end
end
```



```
  defp start_tracer(runner_pid) do
    spawn(fn ->


    end)
  end
```

```
end
```

```
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])

  end)
end
```

```
end
```

```
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])
    tracer_loop(runner_pid, 0)
  end)
end

defp tracer_loop(runner_pid, acc) do
  receive do



  end
end
```

```elixir
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])
    tracer_loop(runner_pid, 0)
  end)
end

defp tracer_loop(runner_pid, acc) do
  receive do



    {:trace, ^runner_pid, :gc_minor_start, info} ->


    {:trace, ^runner_pid, :gc_major_start, info} ->




  end
end
```

```
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])
    tracer_loop(runner_pid, 0)
  end)
end

defp tracer_loop(runner_pid, acc) do
  receive do



    {:trace, ^runner_pid, :gc_minor_start, info} ->
      listen_gc_end(runner_pid, :gc_minor_end, acc, Keyword.fetch!(info, :heap_size))

    {:trace, ^runner_pid, :gc_major_start, info} ->
      listen_gc_end(runner_pid, :gc_major_end, acc, Keyword.fetch!(info, :heap_size))



  end
end

defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  receive do



  end
end
```

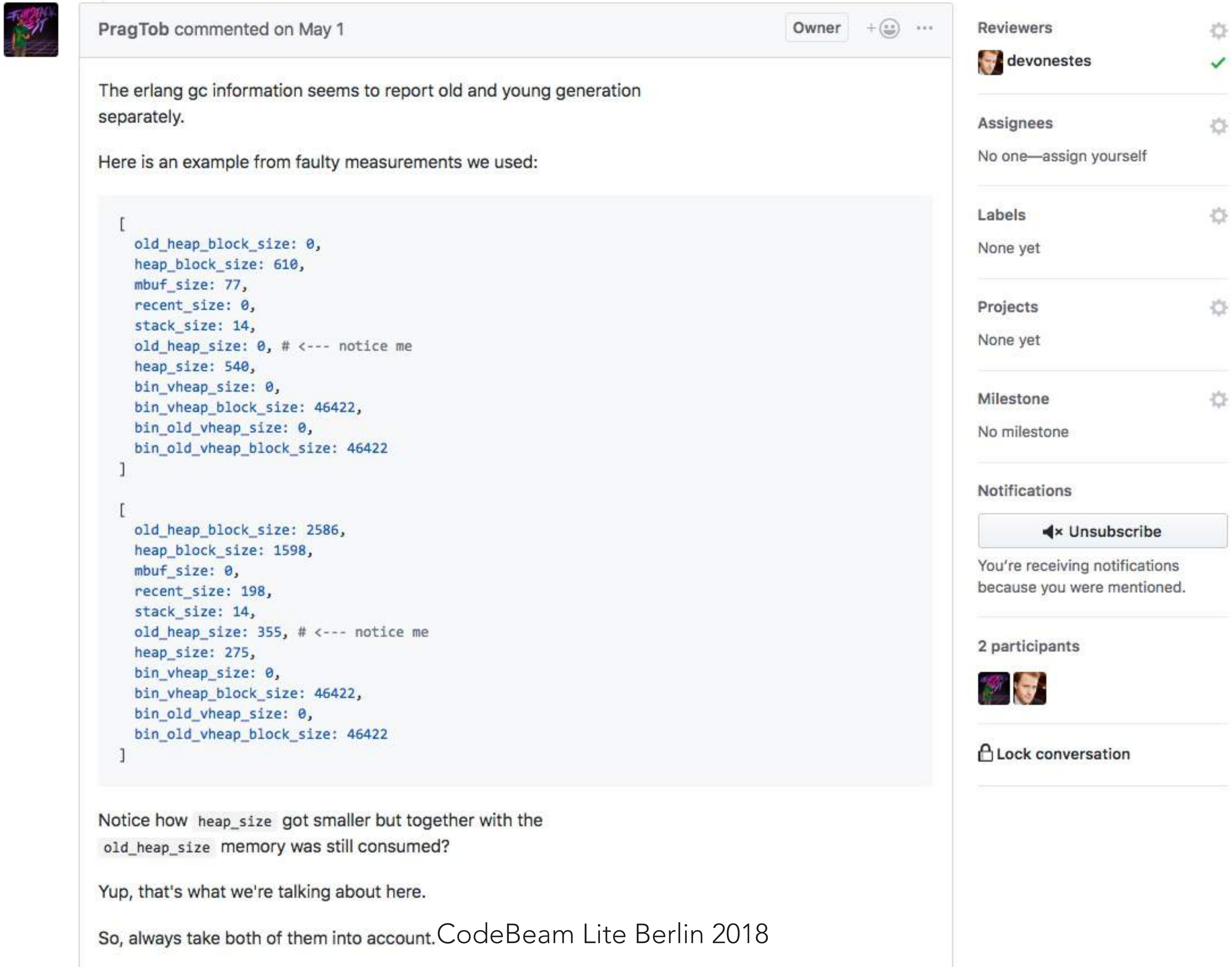
```
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])
    tracer_loop(runner_pid, 0)
  end)
end

defp tracer_loop(runner_pid, acc) do
  receive do




    {:trace, ^runner_pid, :gc_minor_start, info} ->
      listen_gc_end(runner_pid, :gc_minor_end, acc, Keyword.fetch!(info, :heap_size))

    {:trace, ^runner_pid, :gc_major_start, info} ->
      listen_gc_end(runner_pid, :gc_major_end, acc, Keyword.fetch!(info, :heap_size))




  end
end

defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  receive do
    {:trace, ^runner_pid, ^tag, info} ->


  end
end
```

```elixir
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])
    tracer_loop(runner_pid, 0)
  end)
end

defp tracer_loop(runner_pid, acc) do
  receive do



    {:trace, ^runner_pid, :gc_minor_start, info} ->
      listen_gc_end(runner_pid, :gc_minor_end, acc, Keyword.fetch!(info, :heap_size))

    {:trace, ^runner_pid, :gc_major_start, info} ->
      listen_gc_end(runner_pid, :gc_major_end, acc, Keyword.fetch!(info, :heap_size))



  end
end

defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  receive do
    {:trace, ^runner_pid, ^tag, info} ->
      mem_after = Keyword.fetch!(info, :heap_size)

  end
end
```

```
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])
    tracer_loop(runner_pid, 0)
  end)
end

defp tracer_loop(runner_pid, acc) do
  receive do



    {:trace, ^runner_pid, :gc_minor_start, info} ->
      listen_gc_end(runner_pid, :gc_minor_end, acc, Keyword.fetch!(info, :heap_size))

    {:trace, ^runner_pid, :gc_major_start, info} ->
      listen_gc_end(runner_pid, :gc_major_end, acc, Keyword.fetch!(info, :heap_size))



  end
end

defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  receive do
    {:trace, ^runner_pid, ^tag, info} ->
      mem_after = Keyword.fetch!(info, :heap_size)
      tracer_loop(runner_pid, acc + mem_before - mem_after)
  end
end
```

```
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])
    tracer_loop(runner_pid, 0)
  end)
end

defp tracer_loop(runner_pid, acc) do
  receive do
    {:get_collected_memory, reply_to, ref} ->
      send(reply_to, {ref, acc})

    {:trace, ^runner_pid, :gc_minor_start, info} ->
      listen_gc_end(runner_pid, :gc_minor_end, acc, Keyword.fetch!(info, :heap_size))

    {:trace, ^runner_pid, :gc_major_start, info} ->
      listen_gc_end(runner_pid, :gc_major_end, acc, Keyword.fetch!(info, :heap_size))



  end
end

defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  receive do
    {:trace, ^runner_pid, ^tag, info} ->
      mem_after = Keyword.fetch!(info, :heap_size)
      tracer_loop(runner_pid, acc + mem_before - mem_after)
  end
end
```

```elixir
defp start_tracer(runner_pid) do
  spawn(fn ->
    :erlang.trace(runner_pid, true, [:garbage_collection, tracer: self()])
    tracer_loop(runner_pid, 0)
  end)
end

defp tracer_loop(runner_pid, acc) do
  receive do
    {:get_collected_memory, reply_to, ref} ->
      send(reply_to, {ref, acc})

    {:trace, ^runner_pid, :gc_minor_start, info} ->
      listen_gc_end(runner_pid, :gc_minor_end, acc, Keyword.fetch!(info, :heap_size))

    {:trace, ^runner_pid, :gc_major_start, info} ->
      listen_gc_end(runner_pid, :gc_major_end, acc, Keyword.fetch!(info, :heap_size))

    :done ->
      exit(:normal)
  end
end

defp listen_gc_end(runner_pid, tag, acc, mem_before) do
  receive do
    {:trace, ^runner_pid, ^tag, info} ->
      mem_after = Keyword.fetch!(info, :heap_size)
      tracer_loop(runner_pid, acc + mem_before - mem_after)
  end
end
```



PragTob commented on May 1

The erlang gc information seems to report old and young generation separately.

Here is an example from faulty measurements we used:

```
[
  old_heap_block_size: 0,
  heap_block_size: 610,
  mbuf_size: 77,
  recent_size: 0,
  stack_size: 14,
  old_heap_size: 0, # <--- notice me
  heap_size: 540,
  bin_vheap_size: 0,
  bin_vheap_block_size: 46422,
  bin_old_vheap_size: 0,
  bin_old_vheap_block_size: 46422
]

[
  old_heap_block_size: 2586,
  heap_block_size: 1598,
  mbuf_size: 0,
  recent_size: 198,
  stack_size: 14,
  old_heap_size: 355, # <--- notice me
  heap_size: 275,
  bin_vheap_size: 0,
  bin_vheap_block_size: 46422,
  bin_old_vheap_size: 0,
  bin_old_vheap_block_size: 46422
]
```

Notice how `heap_size` got smaller but together with the `old_heap_size` memory was still consumed?

Yup, that's what we're talking about here.

So, always take both of them into account.

Reviewers: devonestes ✓

Assignees: No one—assign yourself

Labels: None yet

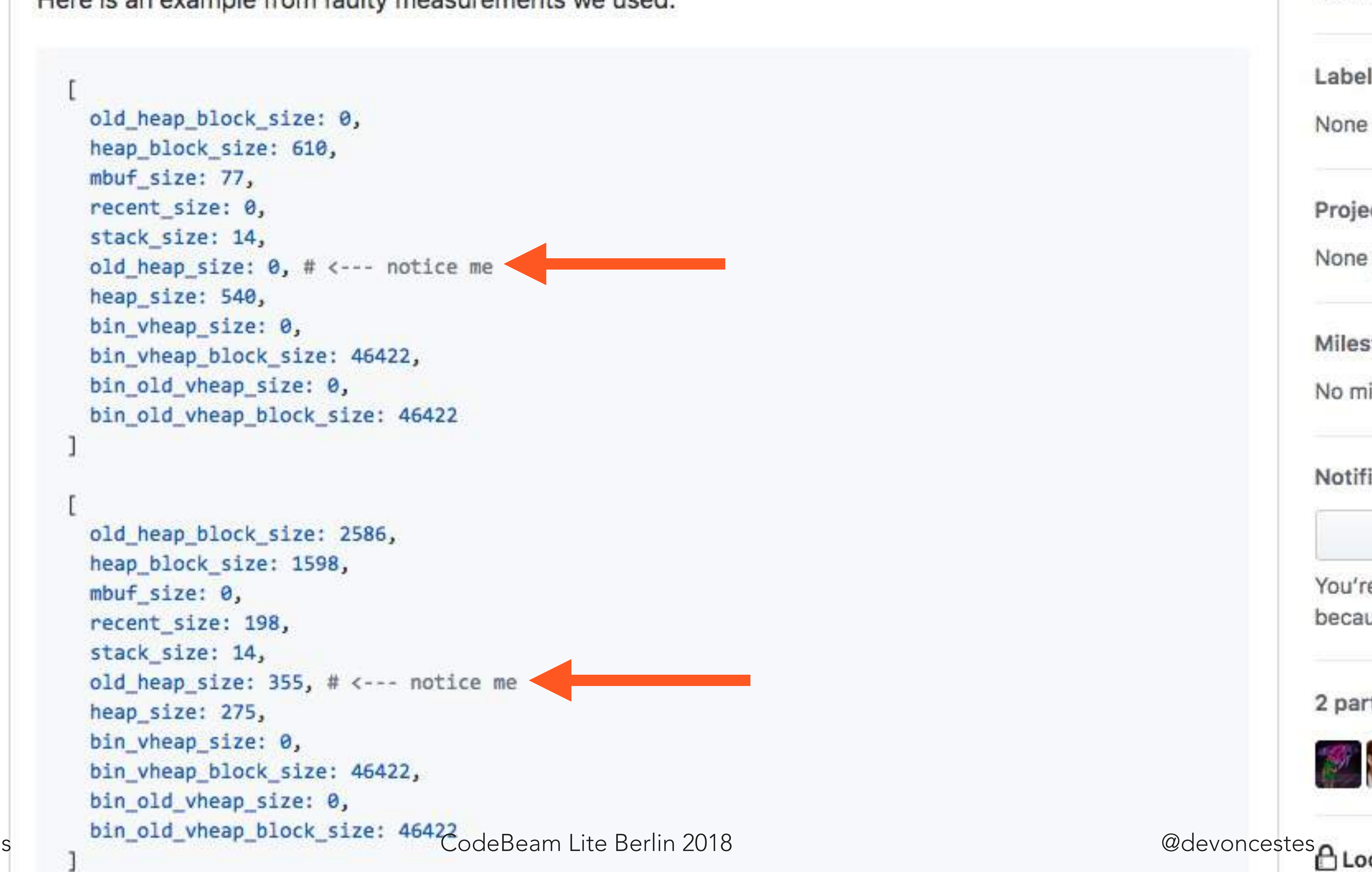
Projects: None yet

Milestone: No milestone

Notifications

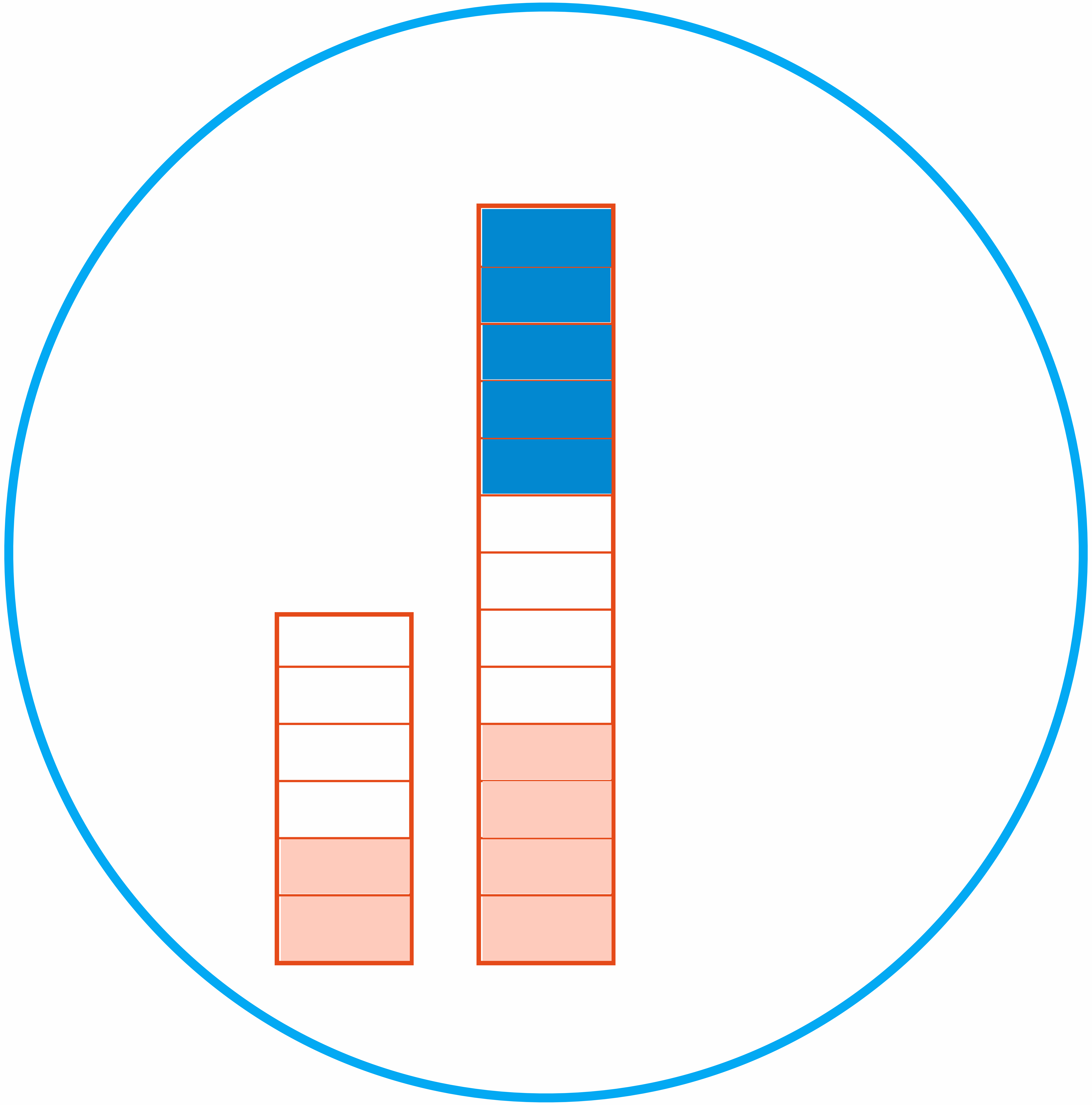
You're receiving notifications because you were mentioned.

2 participants



Here is an example from faulty measurements we used.

```
[
  old_heap_block_size: 0,
  heap_block_size: 610,
  mbuf_size: 77,
  recent_size: 0,
  stack_size: 14,
  old_heap_size: 0, # <--- notice me
  heap_size: 540,
  bin_vheap_size: 0,
  bin_vheap_block_size: 46422,
  bin_old_vheap_size: 0,
  bin_old_vheap_block_size: 46422
]

[
  old_heap_block_size: 2586,
  heap_block_size: 1598,
  mbuf_size: 0,
  recent_size: 198,
  stack_size: 14,
  old_heap_size: 355, # <--- notice me
  heap_size: 275,
  bin_vheap_size: 0,
  bin_vheap_block_size: 46422,
  bin_old_vheap_size: 0,
  bin_old_vheap_block_size: 46422
]
```



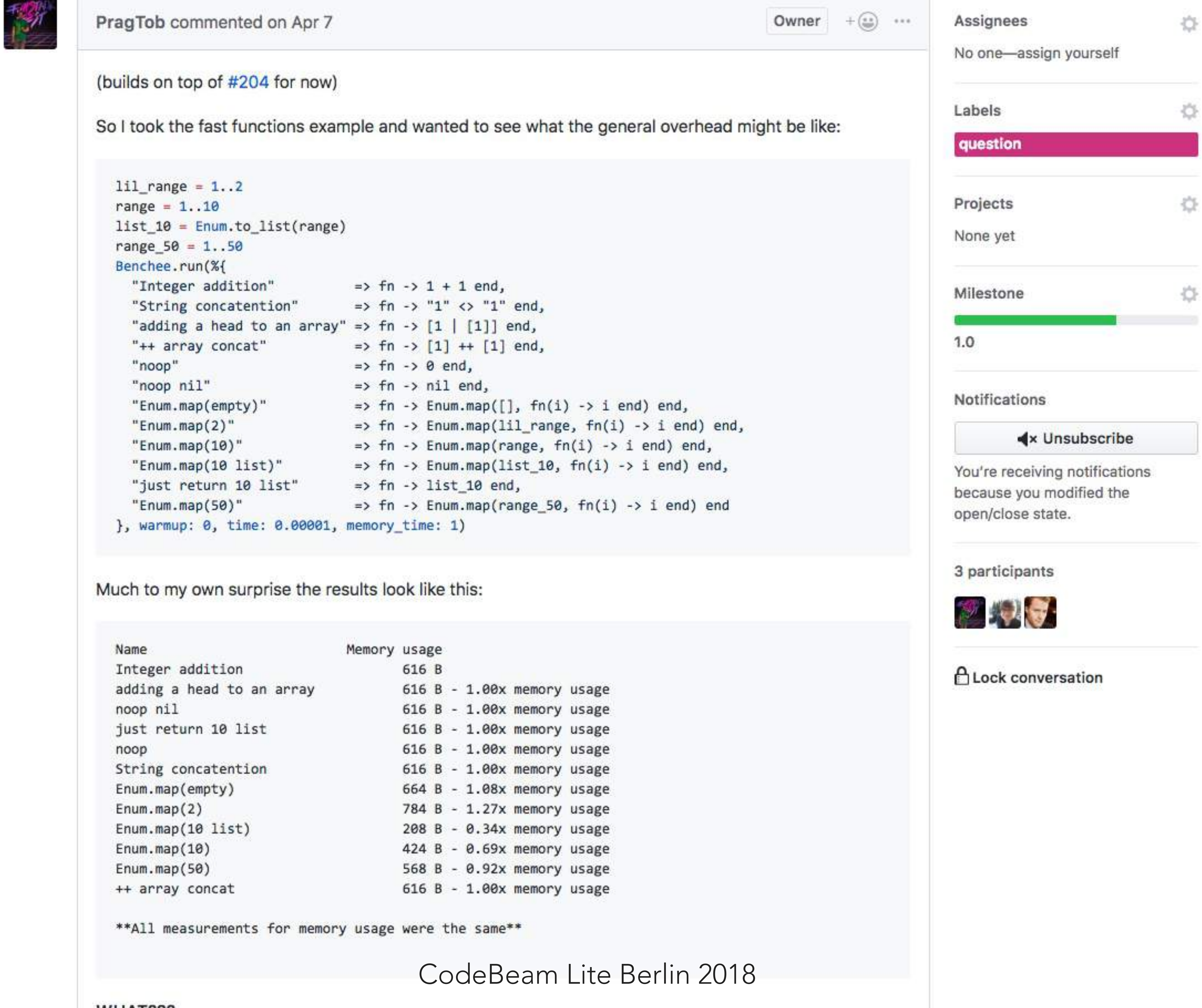
```
  end

  defp tracer_loop(pid, acc) do
    receive do
      {:get_collected_memory, reply_to, ref} ->
        send(reply_to, {ref, acc})

      {:trace, ^pid, :gc_minor_start, info} ->
-       listen_gc_end(pid, :gc_minor_end, acc, Keyword.fetch!(info, :heap_size))

      {:trace, ^pid, :gc_major_start, info} ->
-       listen_gc_end(pid, :gc_major_end, acc, Keyword.fetch!(info, :heap_size))

      :done ->
        exit(:normal)
@@ -93,8 +96,15 @@ defmodule Benchee.Benchmark.Measure.Memory do
  defp listen_gc_end(pid, tag, acc, mem_before) do
    receive do
      {:trace, ^pid, ^tag, info} ->
-       mem_after = Keyword.fetch!(info, :heap_size)
        tracer_loop(pid, acc + mem_before - mem_after)
    end
  end
end
```

```
  end

  defp tracer_loop(pid, acc) do
    receive do
      {:get_collected_memory, reply_to, ref} ->
        send(reply_to, {ref, acc})

      {:trace, ^pid, :gc_minor_start, info} ->
+       listen_gc_end(pid, :gc_minor_end, acc, total_memory(info))

      {:trace, ^pid, :gc_major_start, info} ->
+       listen_gc_end(pid, :gc_major_end, acc, total_memory(info))

      :done ->
        exit(:normal)

  defp listen_gc_end(pid, tag, acc, mem_before) do
    receive do
      {:trace, ^pid, ^tag, info} ->
+       mem_after = total_memory(info)
        tracer_loop(pid, acc + mem_before - mem_after)
    end
  end
+
+ defp total_memory(info) do
+   # `:heap_size` seems to only contain the memory size of the youngest
+   # generation `:old_heap_size` has the old generation. There is also
+   # `:recent_size` but that seems to already be accounted for.
+   Keyword.fetch!(info, :heap_size) + Keyword.fetch!(info, :old_heap_size)
+ end
end
```

ProTip! Use n and p to navigate between commits in a pull request.

```
        {:trace, ^pid, ^tag, info} ->
+         mem_after = total_memory(info)
          tracer_loop(pid, acc + mem_before - mem_after)
      end
    end
+
+   defp total_memory(info) do
+     # `:heap_size` seems to only contain the memory size of the youngest
+     # generation `:old_heap_size` has the old generation. There is also
+     # `:recent_size` but that seems to already be accounted for.
+     Keyword.fetch!(info, :heap_size) + Keyword.fetch!(info, :old_heap_size)
+   end
  end
```

avigate between commits in a pull request.



PragTob commented on Apr 7

(builds on top of #204 for now)

So I took the fast functions example and wanted to see what the general overhead might be like:

```
lil_range = 1..2
range = 1..10
list_10 = Enum.to_list(range)
range_50 = 1..50
Benchee.run(%{
  "Integer addition"          => fn -> 1 + 1 end,
  "String concatention"       => fn -> "1" <> "1" end,
  "adding a head to an array" => fn -> [1 | [1]] end,
  "++ array concat"           => fn -> [1] ++ [1] end,
  "noop"                      => fn -> 0 end,
  "noop nil"                  => fn -> nil end,
  "Enum.map(empty)"           => fn -> Enum.map([], fn(i) -> i end) end,
  "Enum.map(2)"               => fn -> Enum.map(lil_range, fn(i) -> i end) end,
  "Enum.map(10)"              => fn -> Enum.map(range, fn(i) -> i end) end,
  "Enum.map(10 list)"         => fn -> Enum.map(list_10, fn(i) -> i end) end,
  "just return 10 list"       => fn -> list_10 end,
  "Enum.map(50)"              => fn -> Enum.map(range_50, fn(i) -> i end) end
}, warmup: 0, time: 0.00001, memory_time: 1)
```

Much to my own surprise the results look like this:

```
Name                             Memory usage
Integer addition                        616 B
adding a head to an array               616 B - 1.00x memory usage
noop nil                                616 B - 1.00x memory usage
just return 10 list                     616 B - 1.00x memory usage
noop                                    616 B - 1.00x memory usage
String concatention                     616 B - 1.00x memory usage
Enum.map(empty)                         664 B - 1.08x memory usage
Enum.map(2)                             784 B - 1.27x memory usage
Enum.map(10 list)                       208 B - 0.34x memory usage
Enum.map(10)                            424 B - 0.69x memory usage
Enum.map(50)                            568 B - 0.92x memory usage
++ array concat                         616 B - 1.00x memory usage

**All measurements for memory usage were the same**
```

```
lil_range = 1..2
range = 1..10
list_10 = Enum.to_list(range)
range_50 = 1..50
Benchee.run(%{
  "Integer addition"          => fn -> 1 + 1 end,
  "String concatention"       => fn -> "1" <> "1" end,
  "adding a head to an array" => fn -> [1 | [1]] end,
  "++ array concat"           => fn -> [1] ++ [1] end,
  "noop"                      => fn -> 0 end,
  "noop nil"                  => fn -> nil end,
  "Enum.map(empty)"           => fn -> Enum.map([], fn(i) -> i end) end,
  "Enum.map(2)"               => fn -> Enum.map(lil_range, fn(i) -> i end) end,
  "Enum.map(10)"              => fn -> Enum.map(range, fn(i) -> i end) end,
  "Enum.map(10 list)"         => fn -> Enum.map(list_10, fn(i) -> i end) end,
  "just return 10 list"       => fn -> list_10 end,
  "Enum.map(50)"              => fn -> Enum.map(range_50, fn(i) -> i end) end
}, warmup: 0, time: 0.00001, memory_time: 1)
```

Much to my own surprise the results look like this:

```
Name                          Memory usage
Integer addition                     616 B
adding a head to an array            616 B - 1.00x memory usage
noop nil                             616 B - 1.00x memory usage
just return 10 list                  616 B - 1.00x memory usage
noop                                 616 B - 1.00x memory usage
String concatention                  616 B - 1.00x memory usage
Enum.map(empty)                      664 B - 1.08x memory usage
Enum.map(2)                          784 B - 1.27x memory usage
Enum.map(10 list)                    208 B - 0.34x memory usage
Enum.map(10)                         424 B - 0.69x memory usage
Enum.map(50)                         568 B - 0.92x memory usage
++ array concat                      616 B - 1.00x memory usage

**All measurements for memory usage were the same**
```

Erlang › Erlang Questions
Login Re
Previous Topic
Next T
Classic
List
Threaded
10 messages
Optic

Memory usage in OTP 21

Devon Estes Jul 19, 2018; 12:36am **Memory usage in OTP 21**

Reply | Threaded | Mo
5 posts

Hey everyone,

First off, I would like to apologize for the following code example in Elixir - it's the language I know best, and where we're having the problem.

Anyway, I'm one of the maintainers of an Elixir benchmarking tool called Benchee. A few months ago we added memory measurement as a feature in our benchmarking tool. However, with the release of OTP we're seeing some measurements that seem very strange. For example, according to our measurements, the following function uses 0 bytes of memory: `Enum.to_list(1..10)`. On OTP 20, this always uses between 350-380 byes depending on the platform. There are a few other instances of these kinds of functions which we believe should be using memory somewhere, but the results that we get back from our measurements say they are not using any memory, and all of them are around functions that in OTP 20 used very little memory. We are also seeing somewhat frequently what appears to be _negative_ net memory usage, which again seems really strange.

So, is this some sort of optimization that we're missing, or is there somewhere else (maybe in a heap fragment?) that these structures might be stored? And if they are somewhere else, is it possible to measu this memory usage?

At the moment we're measuring memory usage by using `erlang:trace/3` to listen to the garbage collection events, and calculate the used memory from there, and adding any remaining used memory on the heap at the end of the function (after the last GC run).

Thanks again for any help y'all might be able to offer!

____
erlang-questions mailing list
[hidden email]
http://erlang.org/mailman/listinfo/erlang-questions

Jesper Louis Ande Jul 19, 2018; 5:28pm **Re: Memory usage in OTP 21**

I'm pretty sure that's not it, but a function such as

Enum.to_list(1..10)

contains an enumeration which is a constant and to_list can be unfolded to produce [1, ..., 10].

Since that is a constant it ends up as a literal in the beam bytecode and thus it never ever generates any garbage when called. I'm pretty sure that Erlang compiler is not smart enough to make this unfolding, but it doesn't take a lot of work make a compiler constant fold such a case. Especially if it is common in code since compiler developers tend to target that. Another common trick is if escape analysis shows the result doesn't outlive its scope in which case data can be sta allocated, making it far more unlikely to produce heap bump allocation and thus trigger the GC.

This assumption can be verifed by disassembly of the beam bytecode and looking for what the system is doing.

The negative allocation sounds strange to me though. That warrants investigation in what the trace calls are returning IMO to verify it happens at that level or lower.

On Thu, Jul 19, 2018 at 7:42 AM Devon Estes <[hidden email]> wrote:

Changes from 1 commit ▾
Jump to... ▾
+7 −8 🟩🟩🟥🟥▨
Diff settings ▾
Review changes ▾
View
View

devonestes committed on Aug 1

commit aef12117d0b60ab24c00ddfcc1b4cb57523b725b

7 🟩🟩🟥🟥▨ lib/benchee/benchmark/measure/memory.ex

```
@@ -68,10 +68,9 @@ defmodule Benchee.Benchmark.Measure.Memory do
```

```
    end

    defp start_tracer(pid) do
-     spawn(fn ->
-       :erlang.trace(pid, true, [:garbage_collection, tracer: self()])
-       tracer_loop(pid, 0)
-     end)
    end

    defp tracer_loop(pid, acc) do
```

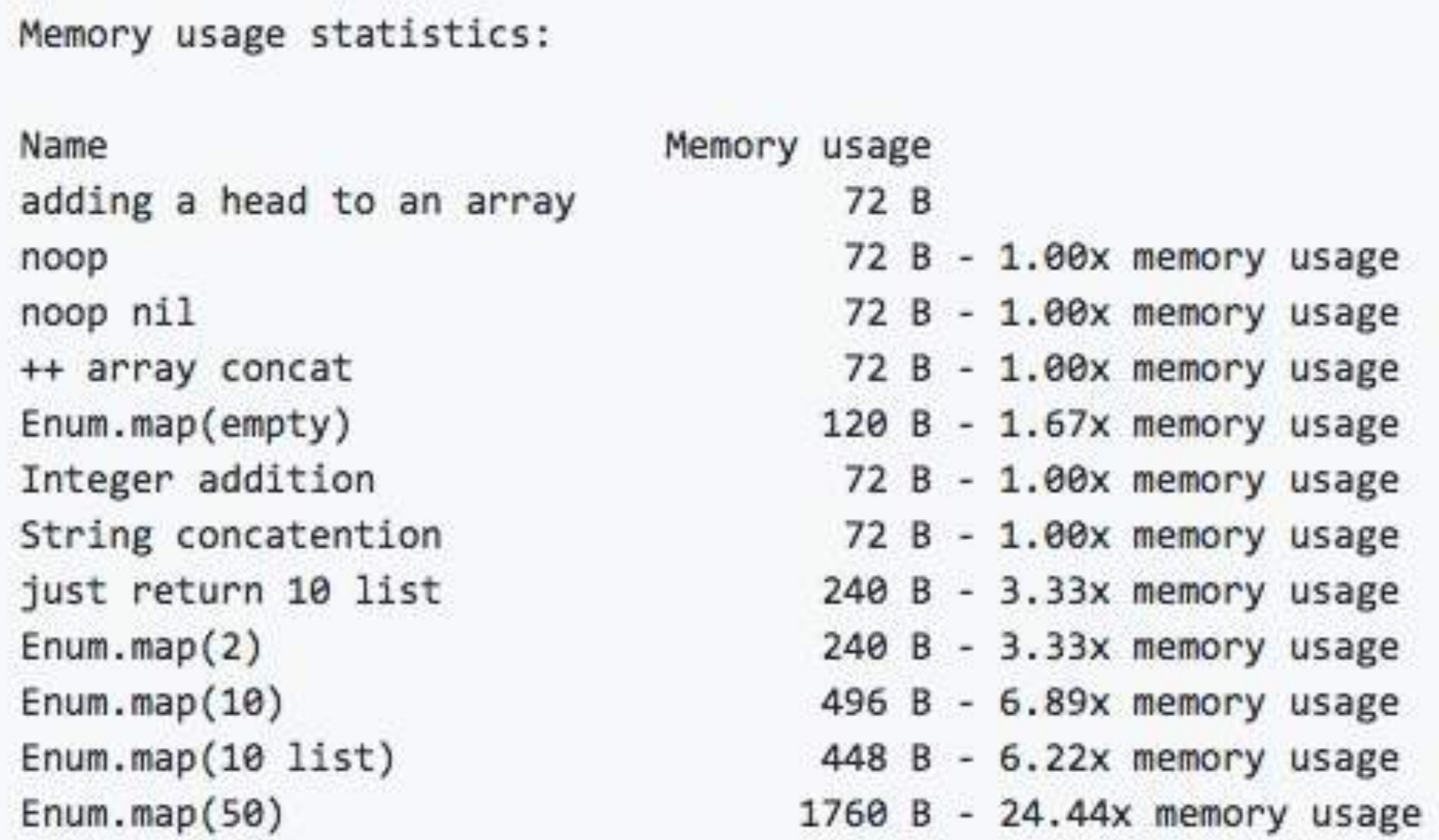
```
    end

    defp start_tracer(pid) do
+     tracer = spawn(fn -> tracer_loop(pid, 0) end)
+     :erlang.trace(pid, true, [:garbage_collection, tracer: tracer])
+     tracer
    end

    defp tracer_loop(pid, acc) do
```

8 🟩🟩🟥🟥▨ test/benchee/benchmark/measure/memory_test.exs

```
@@ -16,16 +16,16 @@ defmodule Benchee.MemoryMeasureTest do
```

```
        # We need to have some wiggle room here because memory used varies from
        # system to system. It's consistent in an environment, but changes
```

```
        # We need to have some wiggle room here because memory used varies from
        # system to system. It's consistent in an environment, but changes
```



```
Memory usage statistics:

Name                           Memory usage
adding a head to an array              72 B
noop                                   72 B - 1.00x memory usage
noop nil                               72 B - 1.00x memory usage
++ array concat                        72 B - 1.00x memory usage
Enum.map(empty)                       120 B - 1.67x memory usage
Integer addition                       72 B - 1.00x memory usage
String concatention                    72 B - 1.00x memory usage
just return 10 list                   240 B - 3.33x memory usage
Enum.map(2)                           240 B - 3.33x memory usage
Enum.map(10)                          496 B - 6.89x memory usage
Enum.map(10 list)                     448 B - 6.22x memory usage
Enum.map(50)                         1760 B - 24.44x memory usage
```

Q
&
A

DEVON ESTES
SENIOR SOFTWARE ENGINEER - ORCHARD SYSTEMS

DEVON.C.ESTES@GMAIL.COM
@DEVONCESTES