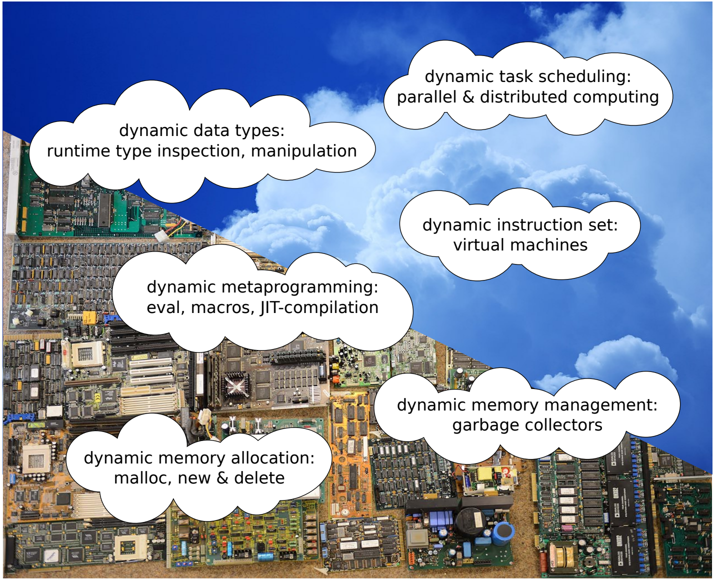
Three garbage collectors: Java, Python, and Julia

Jim Pivarski

Princeton University – IRIS-HEP

January 10, 2024

Dynamic language/programming features



Dynamic language/programming features

	alloc	reference count	GC	eval	VM	type reflect	scheduling
Fortran 77							
C	√						
C++	√	shared_ptr<T>				vtable only	std library
C++ with ROOT	√	shared_ptr<T>		√		√	√
Rust	√	Rc<T>				vtable only	√
Swift	√	√				vtable only	√
Julia	√		√	√		√	std macros
Go	√		√			vtable only	√
Java (JVM languages)	√		√		√	√	std library
Lua	√		√	√	√	√	
Python	√	√	√	√	√	√	√

This talk will focus on. . .

Java

Prototypical example of a language with garbage collection; some of our intuitions/preconceptions about garbage collectors are Java-specific.

Python

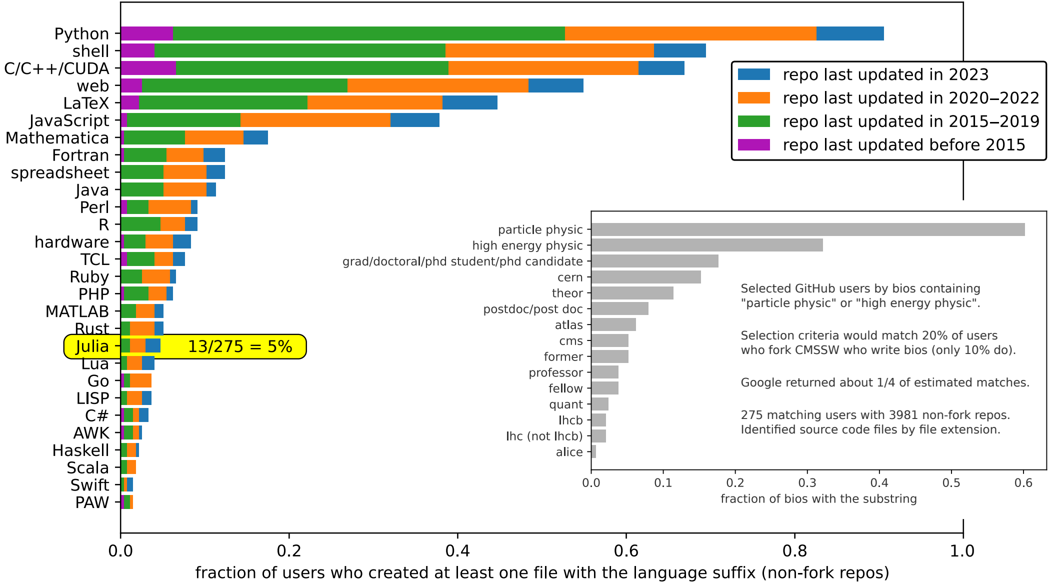
Very dynamic language, has both reference counting and mark-and-sweep garbage collection.

Julia

Up-and-coming language, potentially ideal for HEP. JIT-compiled for bare metal, but has a garbage collector.

What is the *current* status of Julia in HEP?

State of language use by particle physicists as of last November



But physicists are more interested in Julia than, say, Rust or Lua

Among "Materials" (PDFs and TXTs) in CERN's Indico search since January 2022,

- **63 refer to Julia the programming language**
- 324 refer to people named Julia
- 4 other/unclear
- **12 refer to Rust the programming language**
  (7 of those same documents also refer to Julia)
- 10 refer to oxidized metal
- 3 other/unclear
- **1 refers to Lua the programming language**
  (it's used to configure the SIMION charged particle simulator)
- 4 refer to the LHC User's Association
- 4 other/unclear

Similarly, Julia is increasingly a focus on ACAT and CHEP

ACAT 2022:

- Julia: 1 title and 1 abstract
- Python: 3 titles and 24 abstracts

CHEP 2023:

- Julia: 3 titles and 4 abstracts
- Python: 1 title and 35 abstracts

Only other programming languages mentioned: C++ (frequently) and Java (2 times).

Julia has an HSF working group, meetings, and annual workshops

HEP Software Foundation

The HEP Software Foundation facilitates cooperation and common efforts in High Energy Physics software and computing internationally.

JuliaHEP 2023, 6-9 November 2023 (more info)

Meetings

The HSF holds regular meetings in its activity areas and has bi-weekly coordination meetings as well. All of our meetings are open for everyone to join.

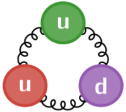
- HSF Coordination Meeting #258, 12 October 2023
- HSF Coordination Meeting #257, 28 September 2023
- HSF Coordination Meeting #256, 14 September 2023

Upcoming HSF and community events »

Full list of past meetings »

JuliaHEP Launches

After a lot of rising interest in Julia for HEP in the last few years, the HSF has started a new JuliaHEP working group.



We just published a new paper Potential of the Julia programming language for high energy physics computing and we're planning the first JuliaHEP Workshop in November. Keep an eye out for upcoming Julia events in the calendar!

Activities

We organise many activities, from our working groups, to organising events, to supporting projects as HSF projects, and helping communication within the community through our discussion forums and technical notes.

The HSF can also write letters of collaboration and cooperation to project proposals.

How to get involved »

Back to garbage collectors

Reference counting

```
>>> import sys

>>> x = object()
>>> sys.getrefcount(x)
2
```

Reference counting

```
>>> import sys

>>> x = object()
>>> sys.getrefcount(x)
2

>>> y = x
>>> sys.getrefcount(x)
3
```

Reference counting

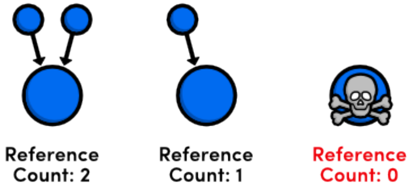
```
>>> import sys

>>> x = object()
>>> sys.getrefcount(x)
2

>>> y = x
>>> sys.getrefcount(x)
3

>>> z = [x, x, x, x, x]
>>> sys.getrefcount(x)
8
```

Reference counting

```
>>> import sys

>>> x = object()
>>> sys.getrefcount(x)
2

>>> y = x
>>> sys.getrefcount(x)
3

>>> z = [x, x, x, x, x]
>>> sys.getrefcount(x)
8

>>> del x, z
>>> sys.getrefcount(y)
2
```

Reference counting

```
>>> import sys

>>> x = object()
>>> sys.getrefcount(x)
2

>>> y = x
>>> sys.getrefcount(x)
3

>>> z = [x, x, x, x, x]
>>> sys.getrefcount(x)
8

>>> del x, z
>>> sys.getrefcount(y)
2
```



The problem with reference counting

```
>>> class HasDestructor:
...     def __del__(self):
...         print("Goodbye, world")
...

>>> x = HasDestructor()
>>> del x
Goodbye, world
```

The problem with reference counting

```
>>> class HasDestructor:
...     def __del__(self):
...         print("Goodbye, world")
...

>>> x = HasDestructor()
>>> del x
Goodbye, world

>>> y = HasDestructor()
>>> y.self = y
>>> del y
```

The problem with reference counting

```
>>> class HasDestructor:
...     def __del__(self):
...         print("Goodbye, world")
...

>>> x = HasDestructor()
>>> del x
Goodbye, world

>>> y = HasDestructor()
>>> y.self = y
>>> del y
```

All references to `y` are gone: it can't be accessed anymore. But it has not been deleted (`__del__` has not been called) because its self-reference keeps its reference count from reaching zero.

The problem with reference counting

```
>>> class HasDestructor:
...     def __del__(self):
...         print("Goodbye, world")
...

>>> x = HasDestructor()
>>> del x
Goodbye, world

>>> y = HasDestructor()
>>> y.self = y
>>> del y

>>> import gc
>>> gc.collect()
Goodbye, world
47
```

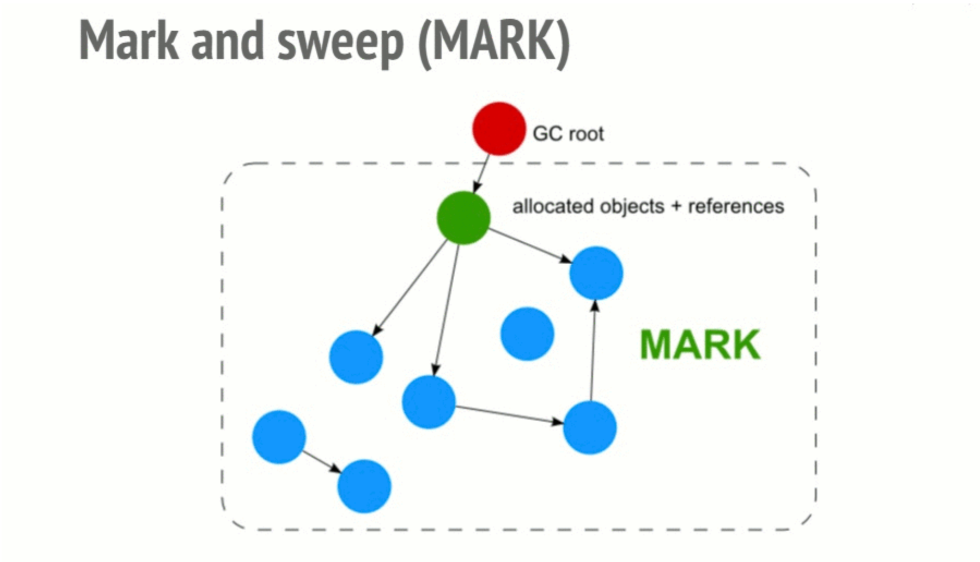
All references to `y` are gone: it can't be accessed anymore. But it has not been deleted (`__del__` has not been called) because its self-reference keeps its reference count from reaching zero.

The problem with reference counting

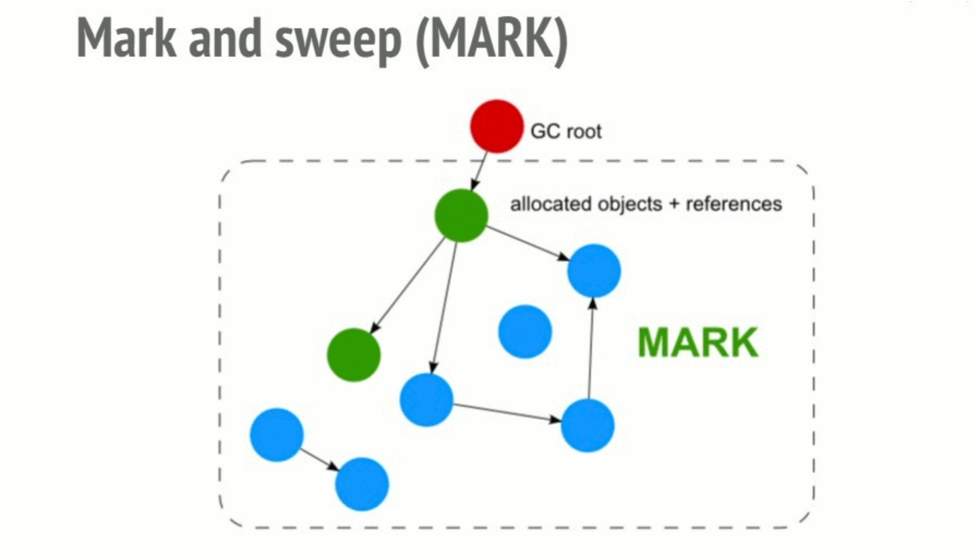
```
>>> class HasDestructor:
...     def __del__(self):
...         print("Goodbye, world")
...

>>> x = HasDestructor()
>>> del x
Goodbye, world

>>> y = HasDestructor()
>>> y.self = y
>>> del y

>>> import gc
>>> gc.collect()
Goodbye, world
47
```

All references to `y` are gone: it can't be accessed anymore. But it has not been deleted (`__del__` has not been called) because its self-reference keeps its reference count from reaching zero.

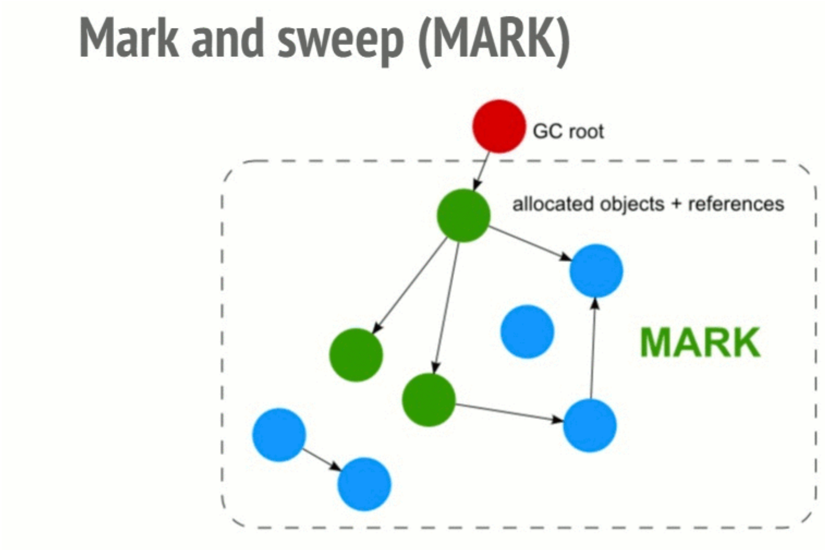
Now it's gone.


Mark and sweep (MARK)
GC root
allocated objects + references
MARK

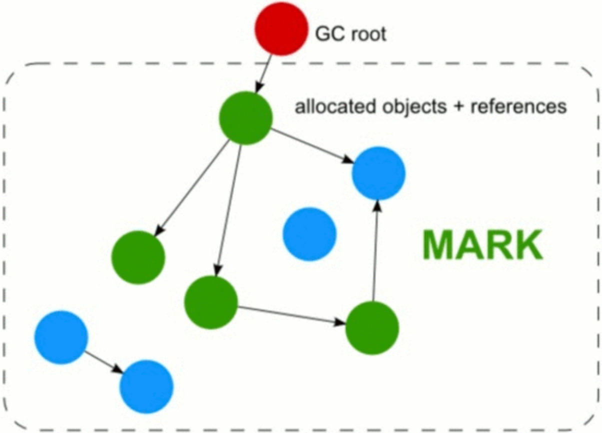
A common garbage collection algorithm



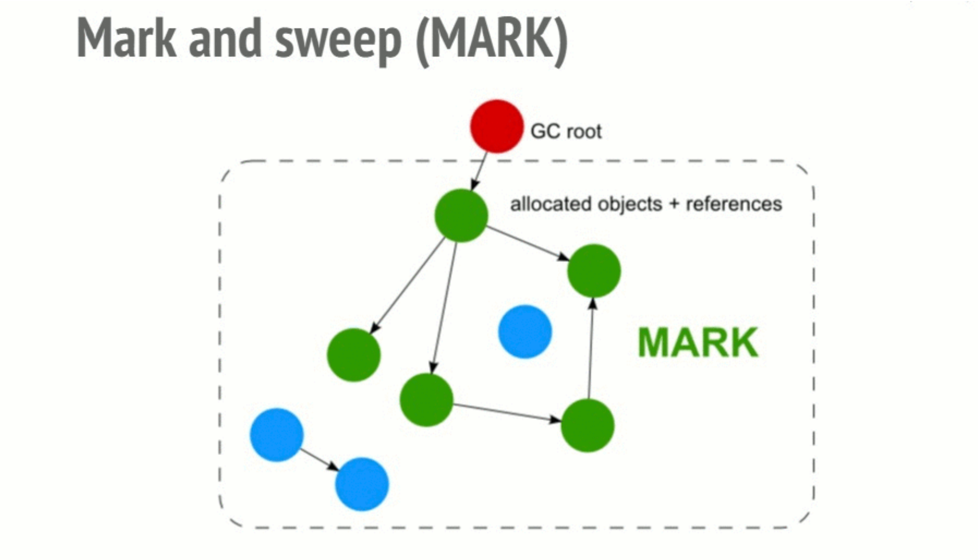
A common garbage collection algorithm



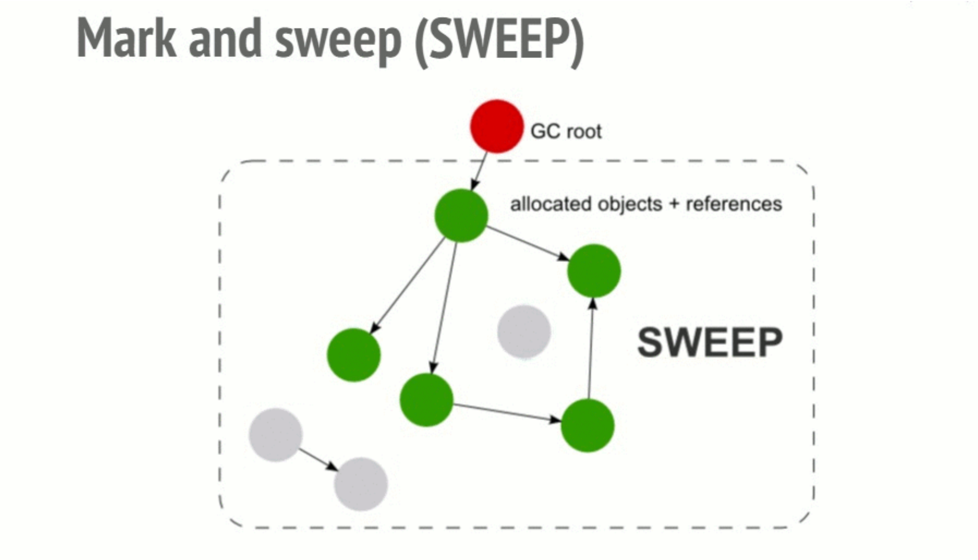
Mark and sweep (MARK)



A common garbage collection algorithm

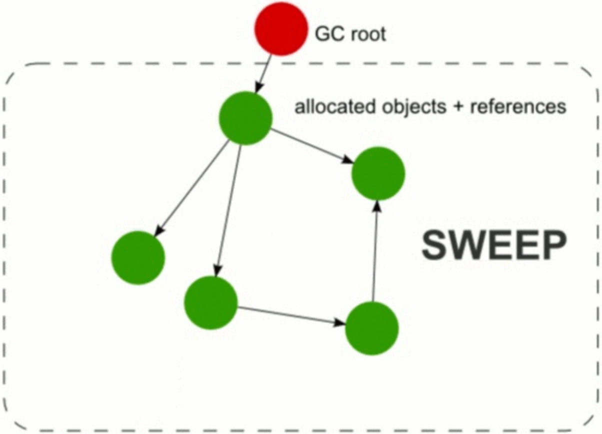


A common garbage collection algorithm

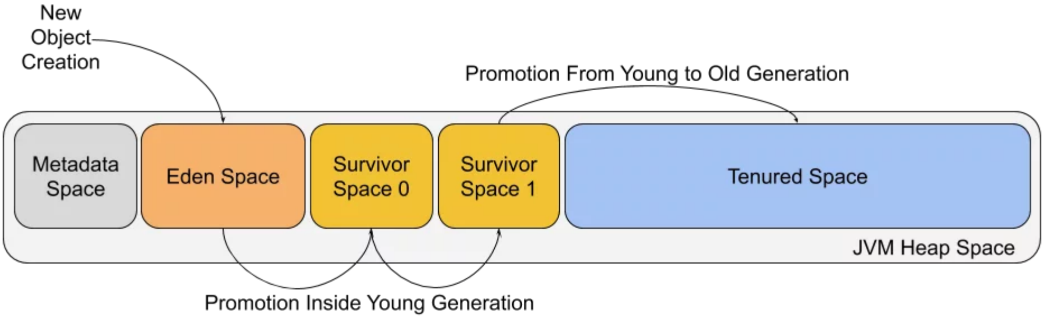


A common garbage collection algorithm

Mark and sweep (SWEEP)



Garbage collector generations



Example in Python, which has 3 generations

```
>>> import gc
>>> _ = gc.collect(); gc.disable()
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[8, 0, 8220]
```

Example in Python, which has 3 generations

```
>>> import gc
>>> _ = gc.collect(); gc.disable()
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[8, 0, 8220]

>>> import uproot
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[57034, 0, 8199]
```

Example in Python, which has 3 generations

```
>>> import gc
>>> _ = gc.collect(); gc.disable()
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[8, 0, 8220]

>>> import uproot
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[57034, 0, 8199]

>>> _ = gc.collect(); [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[3, 0, 39192]
```

Example in Python, which has 3 generations

```
>>> import gc
>>> _ = gc.collect(); gc.disable()
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[8, 0, 8220]

>>> import uproot
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[57034, 0, 8199]

>>> _ = gc.collect(); [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[3, 0, 39192]

>>> uproot.open("Zmumu.root:events").arrays()
<Array [{Type: 'GT', Run: 148031, ...}, ...] type='2304 * {Type: stri...'>
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[33573, 0, 39136]
```

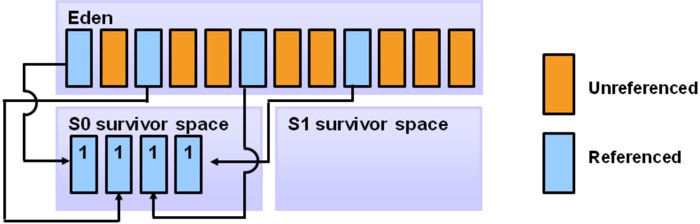
Example in Python, which has 3 generations

```
>>> import gc
>>> _ = gc.collect(); gc.disable()
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[8, 0, 8220]

>>> import uproot
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[57034, 0, 8199]

>>> _ = gc.collect(); [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[3, 0, 39192]

>>> uproot.open("Zmumu.root:events").arrays()
<Array [{Type: 'GT', Run: 148031, ...}, ...] type='2304 * {Type: stri...'>
>>> [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[33573, 0, 39136]

>>> _ = gc.collect(); [len(gc.get_objects(gen)) for gen in (0, 1, 2)]
[3, 0, 56692]
```

Java

https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
https://abiasforaction.net/category/java/gc

Rather than calling `malloc` for each new object, objects are made from preallocated memory pools. Each pool represents a different generation; those that survive mark-and-sweep are *copied* from one pool into the next.



No stable pointers, but it keeps the memory unfragmented: finding space for new objects is fast (i.e. especially good for making many short-lived objects).

Python

https://devguide.python.org/internals/garbage-collector
https://docs.python.org/3/c-api/memory.html
https://docs.python.org/3/using/cmdline.html#envvar-PYTHONMALLOC
https://rushter.com/blog/python-garbage-collector

CPython relies on reference counting for most memory management; full garbage collection is just to clean up cycles. (PyPy only has full garbage collection.)

The 3 generations are different doubly-linked lists. Mark-and-sweep marks are in the low bits of the list pointers so that garbage collection has a constant memory footprint.

```
                  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ \
                  |                   *_gc_next                   | |
                  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ | PyGC_Head
                  |                   *_gc_prev                   | |
    object -----> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ /
                  |                   ob_refcnt                   | \
                  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ | PyObject_HEAD
                  |                   *ob_type                    | |
                  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ /
                  |                      ...                      |
```

Objects have stable pointers, which are good for C/C++ extensions, but not managed by `malloc` (depends on `PYTHONMALLOC` environment variable and Python build).

Julia

https://docs.julialang.org/en/v1/devdocs/gc
https://discourse.julialang.org/t/18021/3
https://docs.julialang.org/en/v1/devdocs/object
https://github.com/JuliaLang/julia/blob/v1.10.0/src/julia.h#L106-L114

$<$ 2 kB objects are managed in pools, allocated by page; large objects use `malloc`.

Objects have headers for type reflection, and the first 2 bits are a mark-and-sweep mark and a generation (1 bit $=$ 2 generations).

```
typedef struct {
    opaque metadata;    /* sizeof(uintptr_t) header */
    jl_value_t value;   /* actual data */
} jl_taggedvalue_t;
```

Marking is depth-first and parallel; sweeping is serial. Memory is returned to the operating system on a per-page basis. Once a page has zero surviving objects, it is freed using `madvise` on a background thread.

Julia makes stack-versus-heap user-visible, to help users avoid garbage collection.

Experiments on garbage collectors

Part 1: timing experiments

Replacing objects with a lifespan of 16 ± 0 steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

```
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects with a lifespan of $16^2 = 256 \pm 0$ steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []



      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects with a lifespan of $16^3 = 4096 \pm 0$ steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []


    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects with a lifespan of $16^4 = 65536 \pm 0$ steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []

  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects with a lifespan of $16^5 = 1048576 \pm 0$ steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects with a lifespan of $16^2 = 256 \pm 97.8$ steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []



      for iE in shuffleE:
        for iD in shuffleD:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects with a lifespan of $16^3 = 4096 \pm 1660$ steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []


    for iE in shuffleE:
      for iD in shuffleD:
        for iC in shuffleC:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects with a lifespan of $16^4 = 65\,536 \pm 26\,700$ steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []

  for iE in shuffleE:
    for iD in shuffleD:
      for iC in shuffleC:
        for iB in shuffleB:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects with a lifespan of $16^5 = 1\,048\,576 \pm 428\,000$ steps

```
shuffleA = [7, 6, 4, 10, 0, 15, 9, 8, 13, 5, 12, 14, 3, 11, 2, 1]
shuffleB = [3, 8, 0, 15, 11, 2, 6, 7, 12, 9, 1, 14, 5, 13, 4, 10]
shuffleC = [2, 13, 6, 7, 4, 5, 10, 3, 12, 15, 8, 9, 14, 1, 0, 11]
shuffleD = [7, 5, 9, 15, 4, 2, 13, 12, 0, 8, 11, 6, 3, 1, 10, 14]
shuffleE = [14, 11, 10, 8, 0, 6, 5, 1, 13, 9, 7, 4, 2, 12, 3, 15]
array = np.empty(16**5, dtype=object)
for iA in shuffleA:
  for iB in shuffleB:
    for iC in shuffleC:
      for iD in shuffleD:
        for iE in shuffleE:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
for iE in shuffleE:
  for iD in shuffleD:
    for iC in shuffleC:
      for iB in shuffleB:
        for iA in shuffleA:
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE] = []
```

Replacing objects in Julia

```
shuffleA::Vector{Int64} = [ 7,  6,  4, 10,  0, 15,  9,  8,
                           13,  5, 12, 14,  3, 11,  2,  1]
...
array::Vector{Union{Vector{Int32},Nothing}} = fill(nothing, 16^5)
for iA in shuffleA
  for iB in shuffleB
    for iC in shuffleC
      for iD in shuffleD
        for iE in shuffleE
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE + 1] = []
        end
        for iE in shuffleE
          array[(((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE + 1] = []
        end
      end
    end
  end
end
```

Replacing objects on the JVM (Scala)

```scala
val shuffleA = Array[Int]( 7,   6,   4, 10,   0, 15,   9,   8,
                          13,   5, 12, 14,   3, 11,   2,   1)
...
val array = Array.fill(Math.pow(16, 5).toInt)(Option.empty[Array[Int]])
for (iA <- shuffleA) {
  for (iB <- shuffleB) {
    for (iC <- shuffleC) {
      for (iD <- shuffleD) {
        for (iE <- shuffleE)
          array((((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE)) =
            Some(Array[Int]())
        for (iE <- shuffleE)
          array((((iA*16 + iB)*16 + iC)*16 + iD)*16 + iE)) =
            Some(Array[Int]())
      }
    }
  }
}
```

Measuring time uniformly across languages

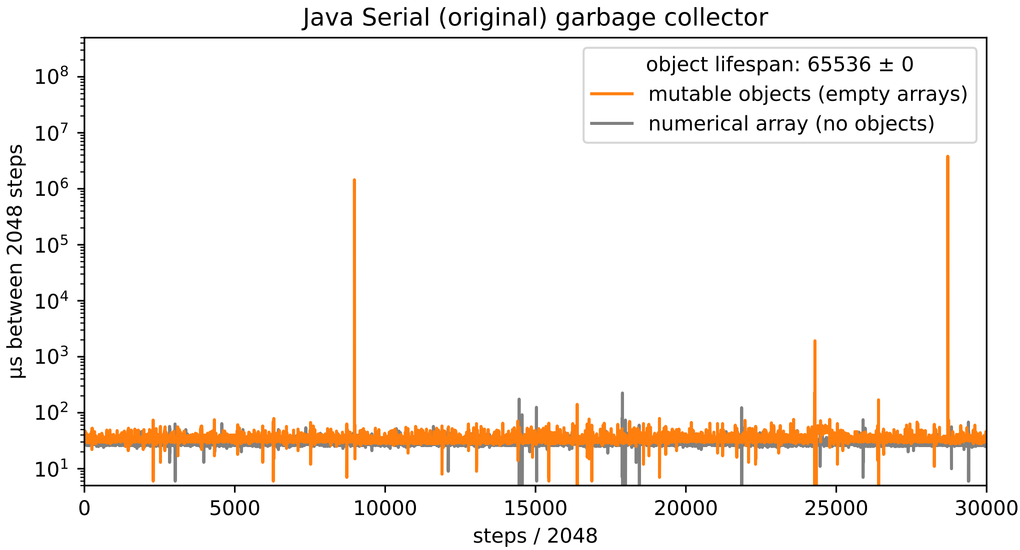
The Scala, Python, and Julia processes each send a byte (`'.'`) to a separate C++ process every 2048 (2^{11}) steps, which is listening to an unbuffered, localhost socket.

The C++ process records time differences between each received byte.

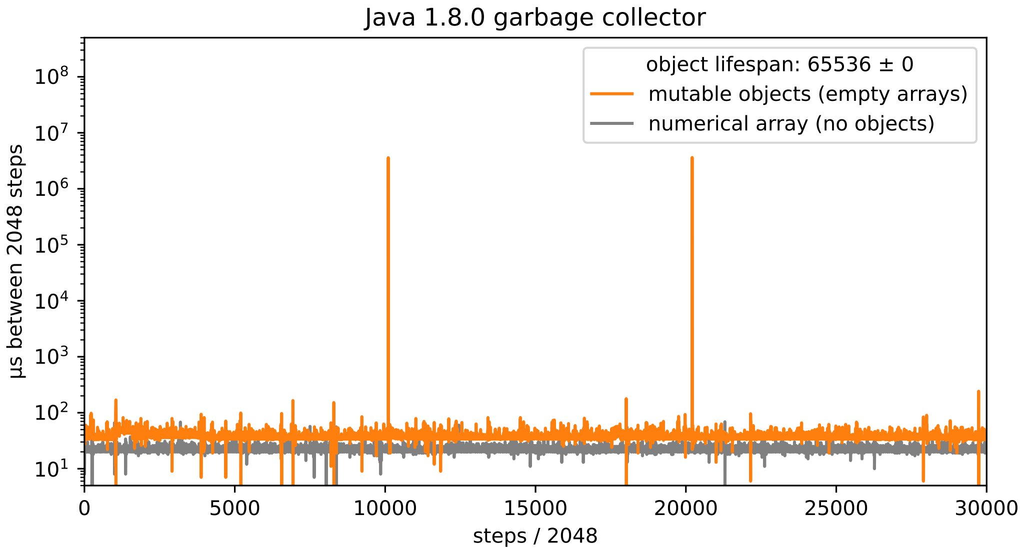
```
using namespace std::chrono;

do {
  recv(new_socket, &buffer, 1, 0);
  stop = std::chrono::steady_clock::now();
  printf("%d\n", duration_cast<microseconds>(stop - start).count());
  start = stop;
}
while (buffer == '.');
```

When Scala, Python, or Julia are stopped by a garbage collector pause, it shows up as an unusually long time between pings.

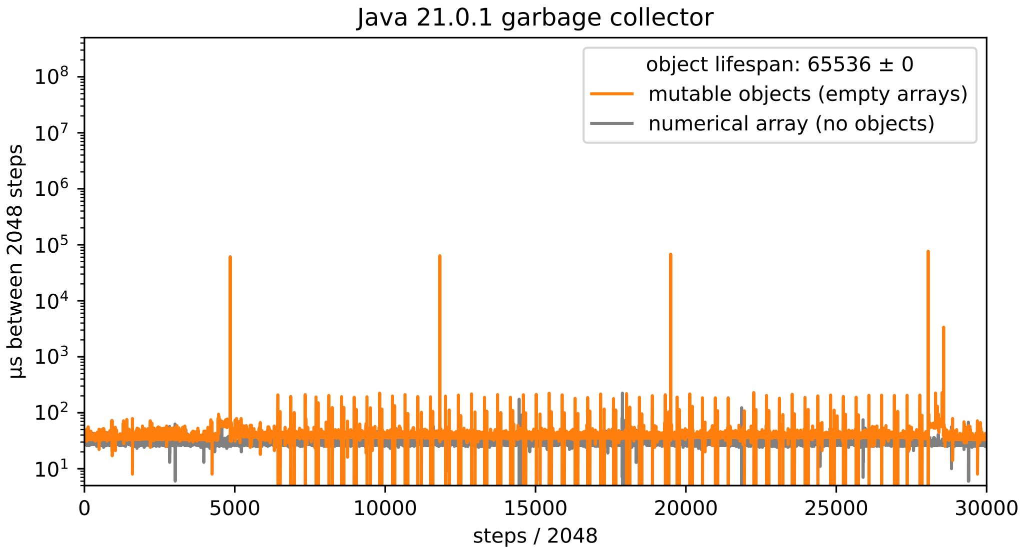
Java: actively researching garbage collectors for decades



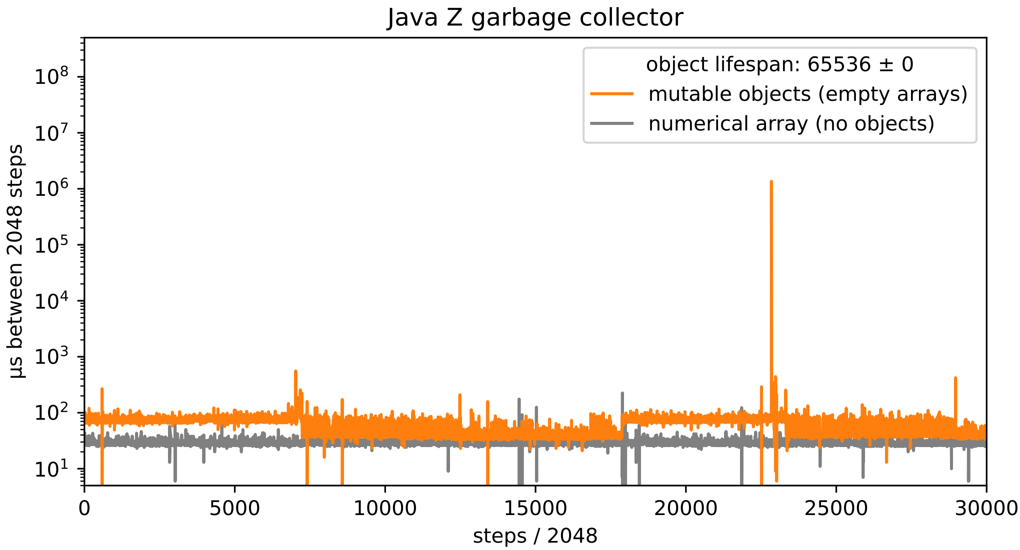
Java: actively researching garbage collectors for decades



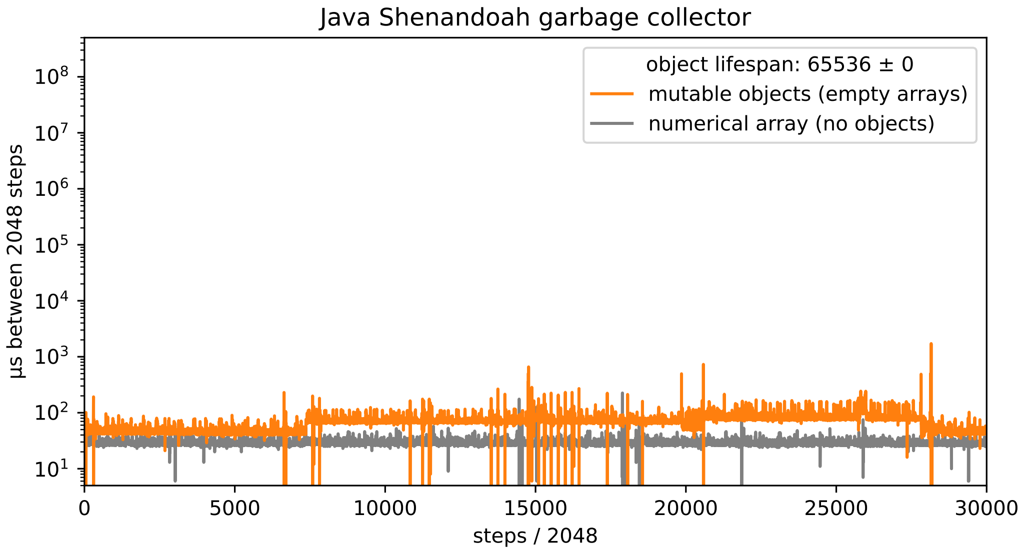
Java: actively researching garbage collectors for decades



Java: actively researching garbage collectors for decades



Java: actively researching garbage collectors for decades



Java: actively researching garbage collectors for decades

The OpenJDK and other JDK implementations come with many algorithms:

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate
G1 GC	Medium to large	Short to medium	High	Higher	Mixed workloads, Large enterprise systems	Moderate to High
Z GC	Large	Very Short	High	Very High	Latency-sensitive applications, Large-scale systems	Low to moderate
Shenandoah GC	Medium to large	Very Short	High	Very High	Low-latency applications, Large-scale systems	Low to Moderate
Epsilon GC	N/A	N/A	N/A	Very High	Performance testing, Memory allocation analysis	Very Low
Azul C4 GC	Large	Very Short	High	Very High	Enterprise applications, Cloud environments	Low to moderate
IBM Metronome GC	N/A	Very Short	Very High	Very High	Real-time applications, Predictable latency requirements	Very Low
SAP GC	Large	Short to medium	High	High	Enterprise applications, SAP environments	Moderate to High

and each has many tuning options (not explored in this talk), to address different throughput-versus-latency tradeoffs.

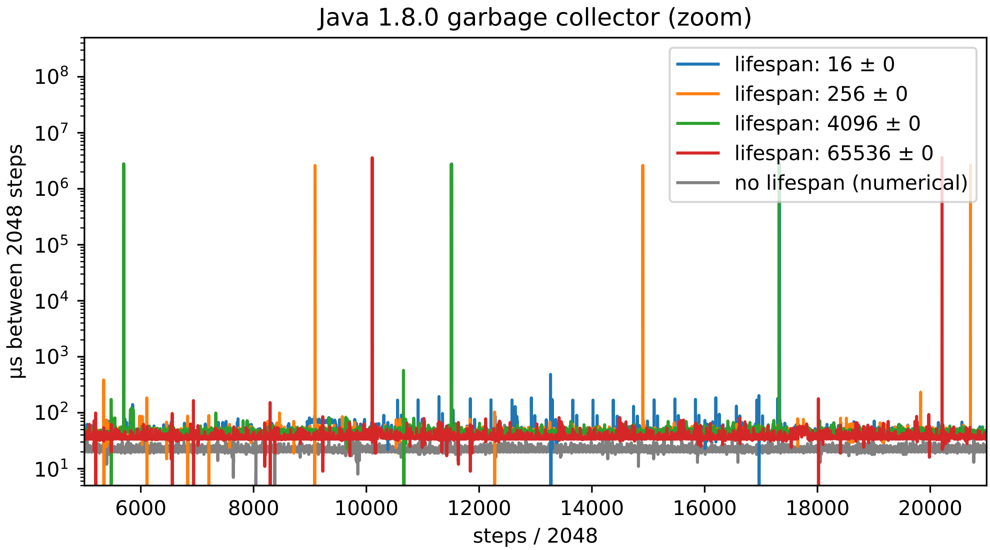
Examples of throughput-versus-latency tradeoffs

Example I've encountered: in a distributed system, one service was sending messages to another. During the garbage collector pauses, the recipient's input queue would overflow, which created yet more objects to garbage-collect, in a viscious cycle.

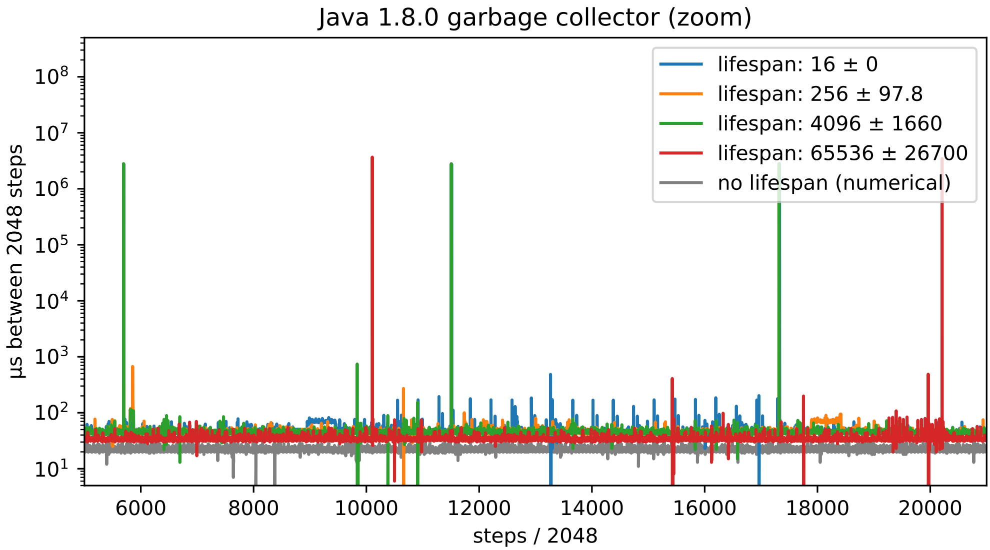
I spent days or weeks testing and tuning alternate garbage collectors.

Obvious cases for HEP: triggers, data acquisition, monitoring. Any real-time system.

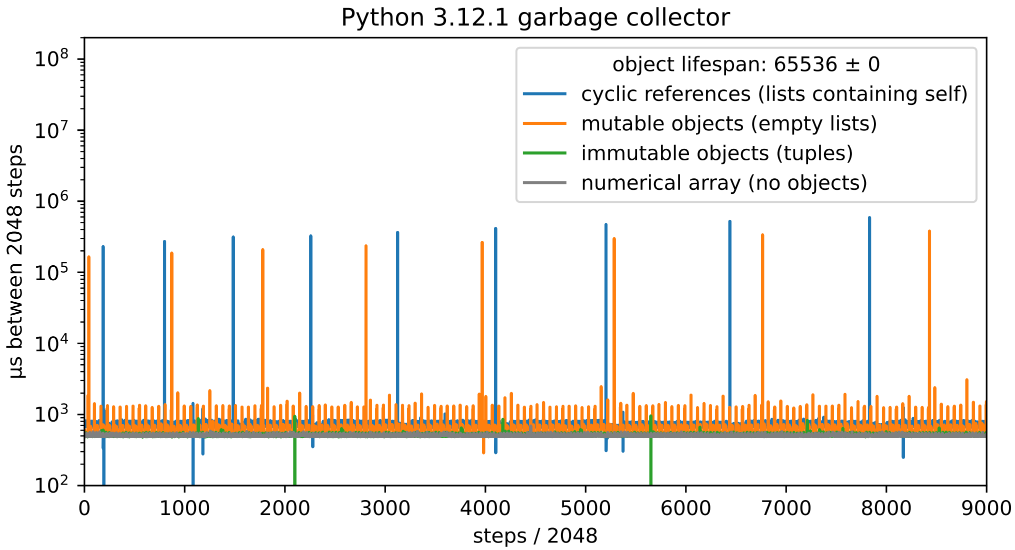
Java: scaling with object lifespan



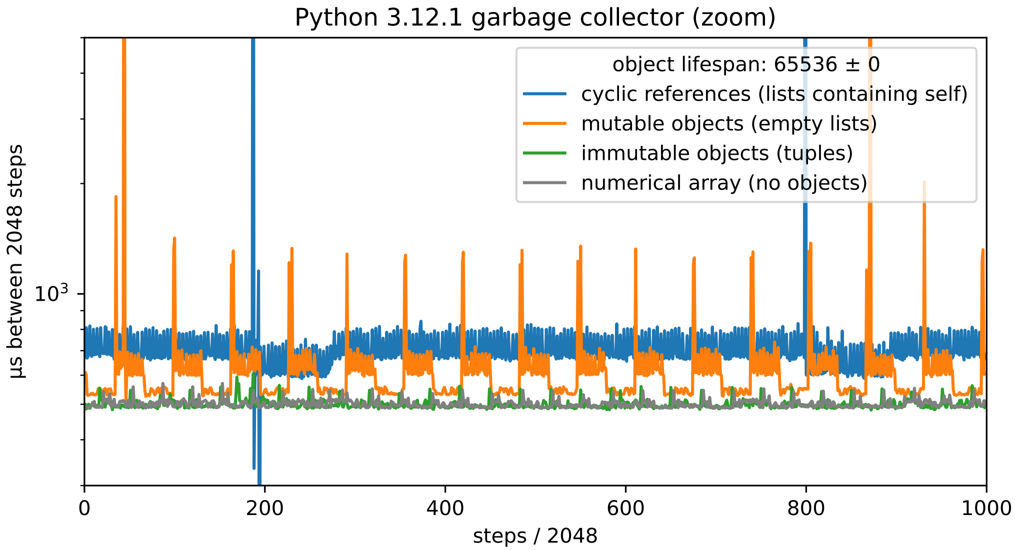
Java: scaling with object lifespan



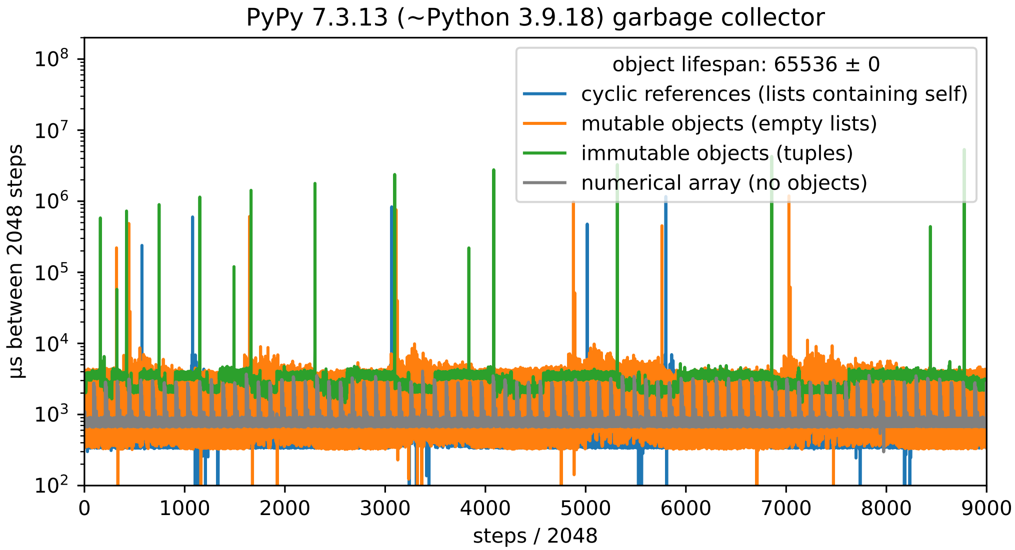
Python: only triggered for mutable objects



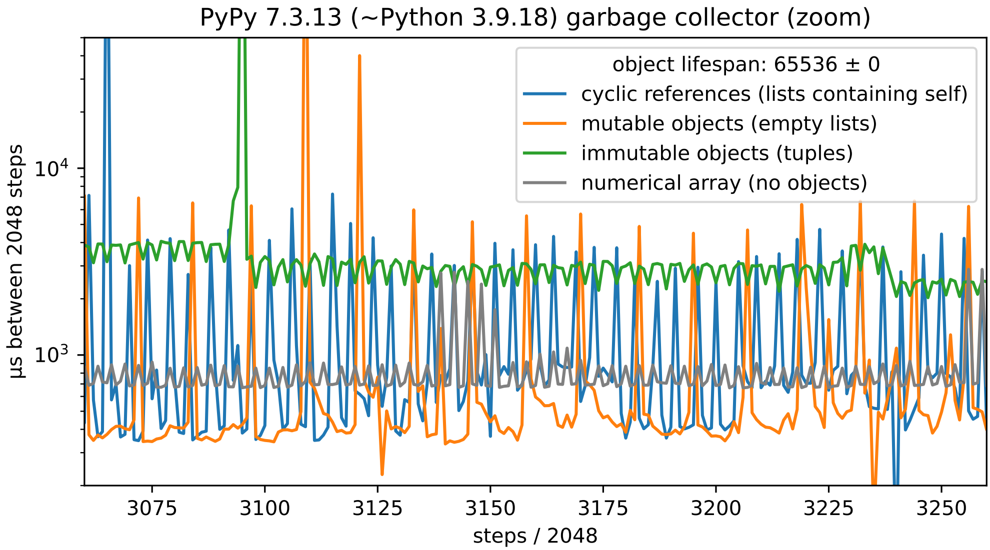
Python: only triggered for mutable objects



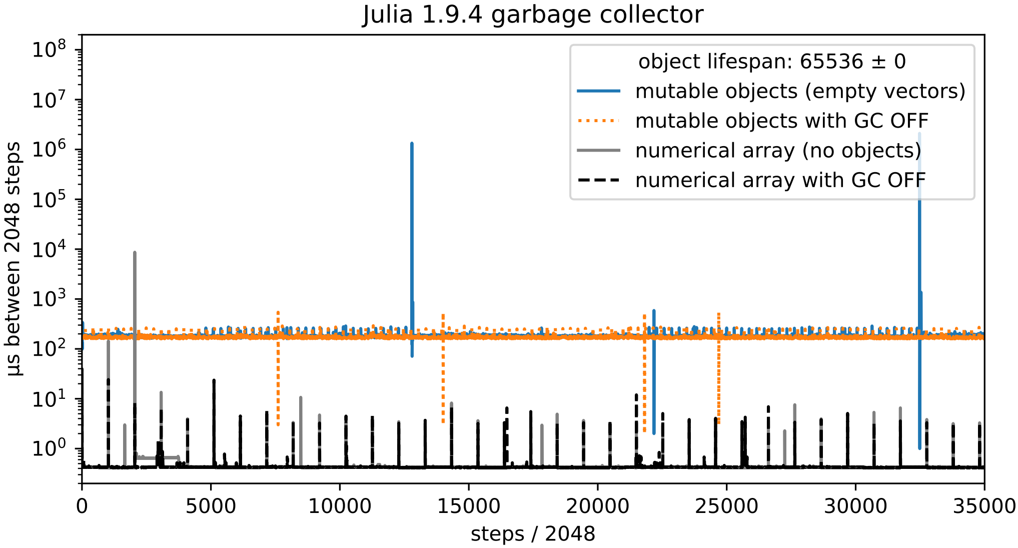
PyPy: triggered for all objects



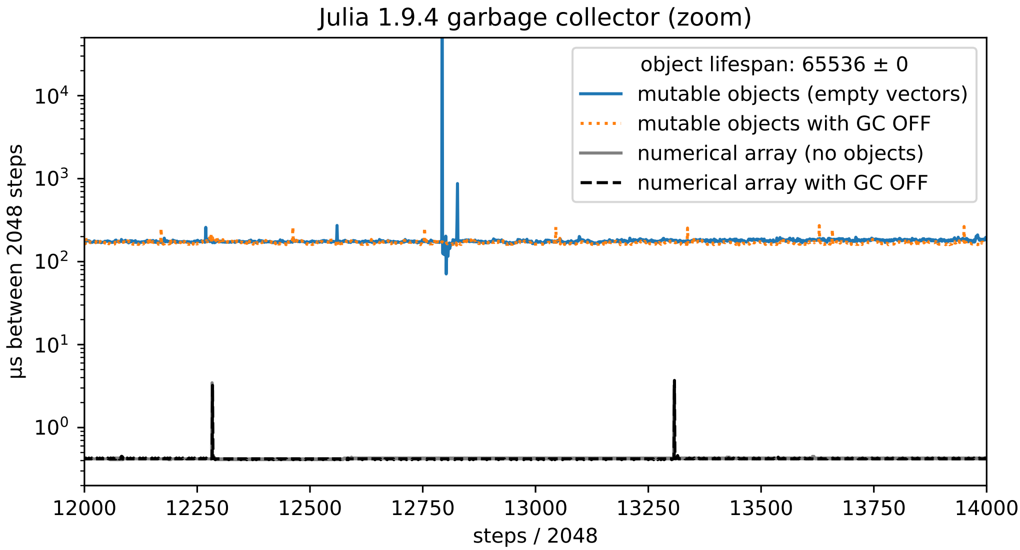
PyPy: triggered for all objects



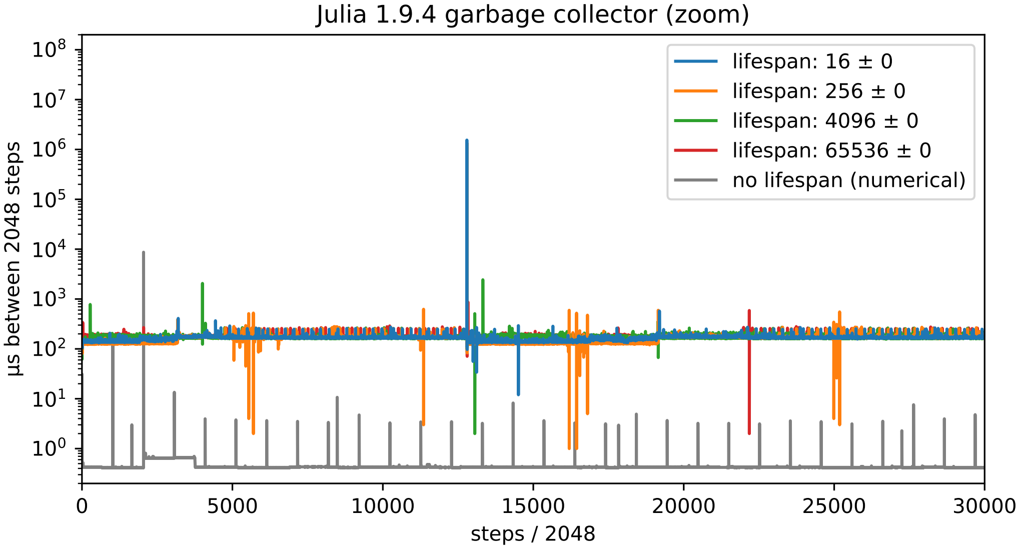
Julia: has pauses, but very regular; may be easy to reason about



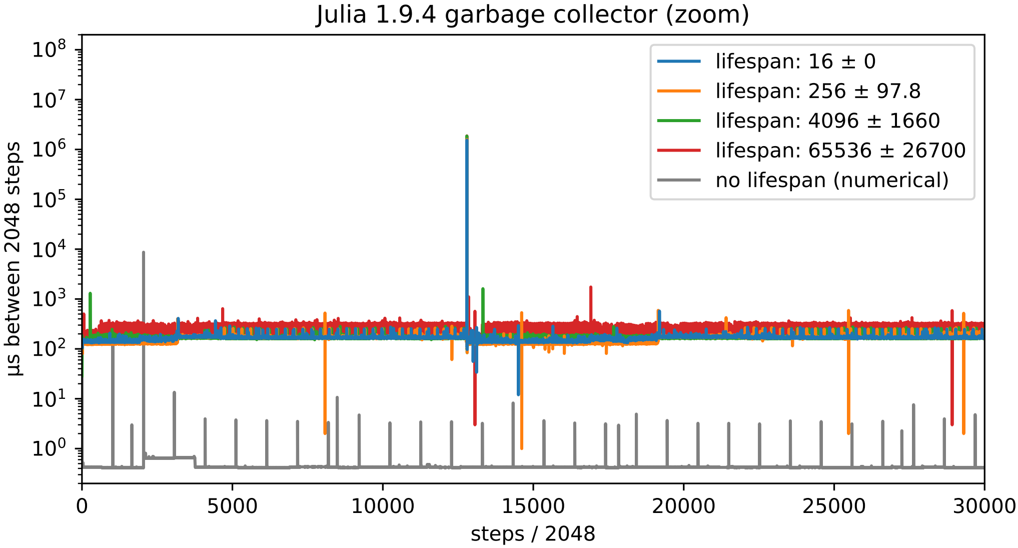
Julia: has pauses, but very regular; may be easy to reason about



Julia: short object lifespan still has second-long pauses



Julia: short object lifespan still has second-long pauses



Alternative garbage collectors in Python and Julia

Python's and Julia's garbage collectors are not as tunable as Java's, but you can...

- Turn them off completely, at least for debugging: `gc.disable()` (Python) and `GC.enable(false)` (Julia).
- Invoke them manually: `gc.collect()` (Python) and `GC.gc()` (Julia).
- Python's garbage collector is triggered by the number of objects since last collection, which can be tuned: `gc.set_threshold(700, 10, 10)`.
- Both can be logged, and in Python, you can also set callbacks (code) on various garbage collector phases.
- Python has an alternate implementation, PyPy, which only does tracing garbage collection (no reference counting) and is tuned differently: https://doc.pypy.org/en/latest/gc_info.html.

Julia community helps its members avoid heap-allocation

Heap-allocation and garbage collection exist for prototyping and early iterations of development, but the Julia community actively helps users eliminate it from hot loops.

- Same philosophy as with other dynamic features, such as type reflection.
- https://docs.julialang.org/en/v1/manual/performance-tips/#Measure-performance-with-[@time](@ref)-and-pay-attention-to-memory-allocation
- https://stackoverflow.com/questions/tagged/julia+allocation
- Static allocation-checker: https://github.com/JuliaLang/AllocCheck.jl.

Experiments on garbage collectors

Part 2: memory footprint

Does garbage collection run more often if there's less memory?

Does garbage collection run more often if there's less memory?

Java: YES. That's what the `-Xms` (min) and `-Xmx` (max) arguments are for.

Does garbage collection run more often if there's less memory?

Java: YES. That's what the `-Xms` (min) and `-Xmx` (max) arguments are for.

Python and Julia: ? I had always assumed so, but let's test it.

Does garbage collection run more often if there's less memory?

Java: YES. That's what the `-Xms` (min) and `-Xmx` (max) arguments are for.

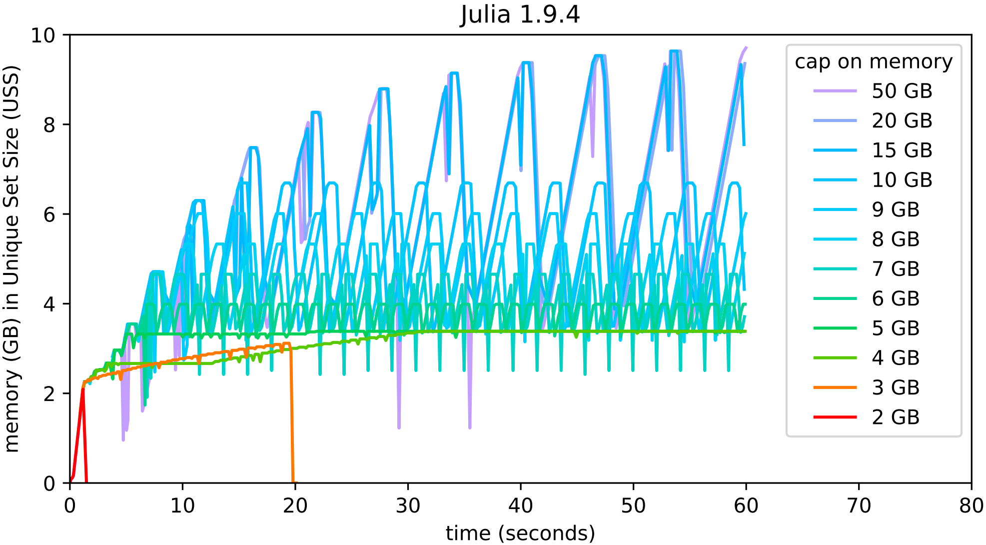
Python and Julia: ? I had always assumed so, but let's test it.

The following runs command `X` with at most `1000M` of memory:

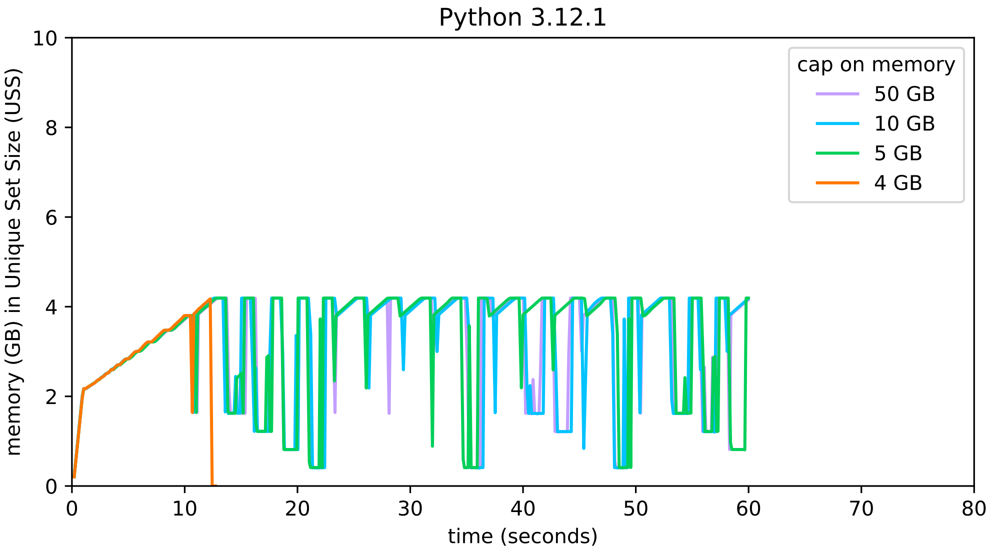
```
systemd-run --user --scope -p MemoryMax=1000M -p MemorySwapMax=0M X
```

Julia: YES

https://discourse.julialang.org/t/64592

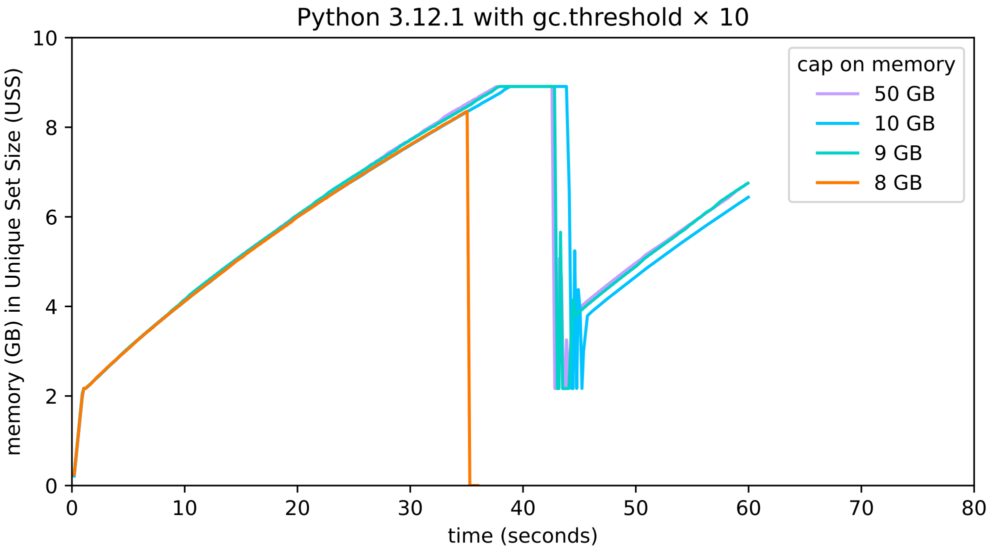


Python: NO



Python 3.12.1
memory (GB) in Unique Set Size (USS)
time (seconds)
cap on memory
50 GB
10 GB
5 GB
4 GB
0
2
4
6
8
10
0
10
20
30
40
50
60
70
80

Python: NO



Python 3.12.1 with gc.threshold × 10
memory (GB) in Unique Set Size (USS)
time (seconds)
cap on memory
50 GB
10 GB
9 GB
8 GB
0
2
4
6
8
10
0
10
20
30
40
50
60
70
80

Conclusions

- Garbage collection is just one of many dynamic programming language features, some of which we take for granted.

Conclusions

- Garbage collection is just one of many dynamic programming language features, some of which we take for granted.
- Julia is an up-and-coming language, which could be ideal for HEP.

Conclusions

- Garbage collection is just one of many dynamic programming language features, some of which we take for granted.
- Julia is an up-and-coming language, which could be ideal for HEP.
- Java's garbage collector is generational, compacting, scales with available memory, and has many, many options.

Conclusions

- Garbage collection is just one of many dynamic programming language features, some of which we take for granted.
- Julia is an up-and-coming language, which could be ideal for HEP.
- Java's garbage collector is generational, compacting, scales with available memory, and has many, many options.
- Python's garbage collector is generational (3), not compacting, does not scale with available memory, and is only needed to clean up objects with cyclical references.

Conclusions

- Garbage collection is just one of many dynamic programming language features, some of which we take for granted.
- Julia is an up-and-coming language, which could be ideal for HEP.
- Java's garbage collector is generational, compacting, scales with available memory, and has many, many options.
- Python's garbage collector is generational (3), not compacting, does not scale with available memory, and is only needed to clean up objects with cyclical references.
- Julia's garbage collector is generational (2), not compacting, and scales with available memory. Also, I'd characterize it as "well-behaved."

Conclusions

- Garbage collection is just one of many dynamic programming language features, some of which we take for granted.
- Julia is an up-and-coming language, which could be ideal for HEP.
- Java's garbage collector is generational, compacting, scales with available memory, and has many, many options.
- Python's garbage collector is generational (3), not compacting, does not scale with available memory, and is only needed to clean up objects with cyclical references.
- Julia's garbage collector is generational (2), not compacting, and scales with available memory. Also, I'd characterize it as "well-behaved."
- The Julia community considers avoiding heap-allocation a priority and helps users to achieve it.