The Economics of Garbage Collection

Jeremy Singer

MANCHESTER 1824

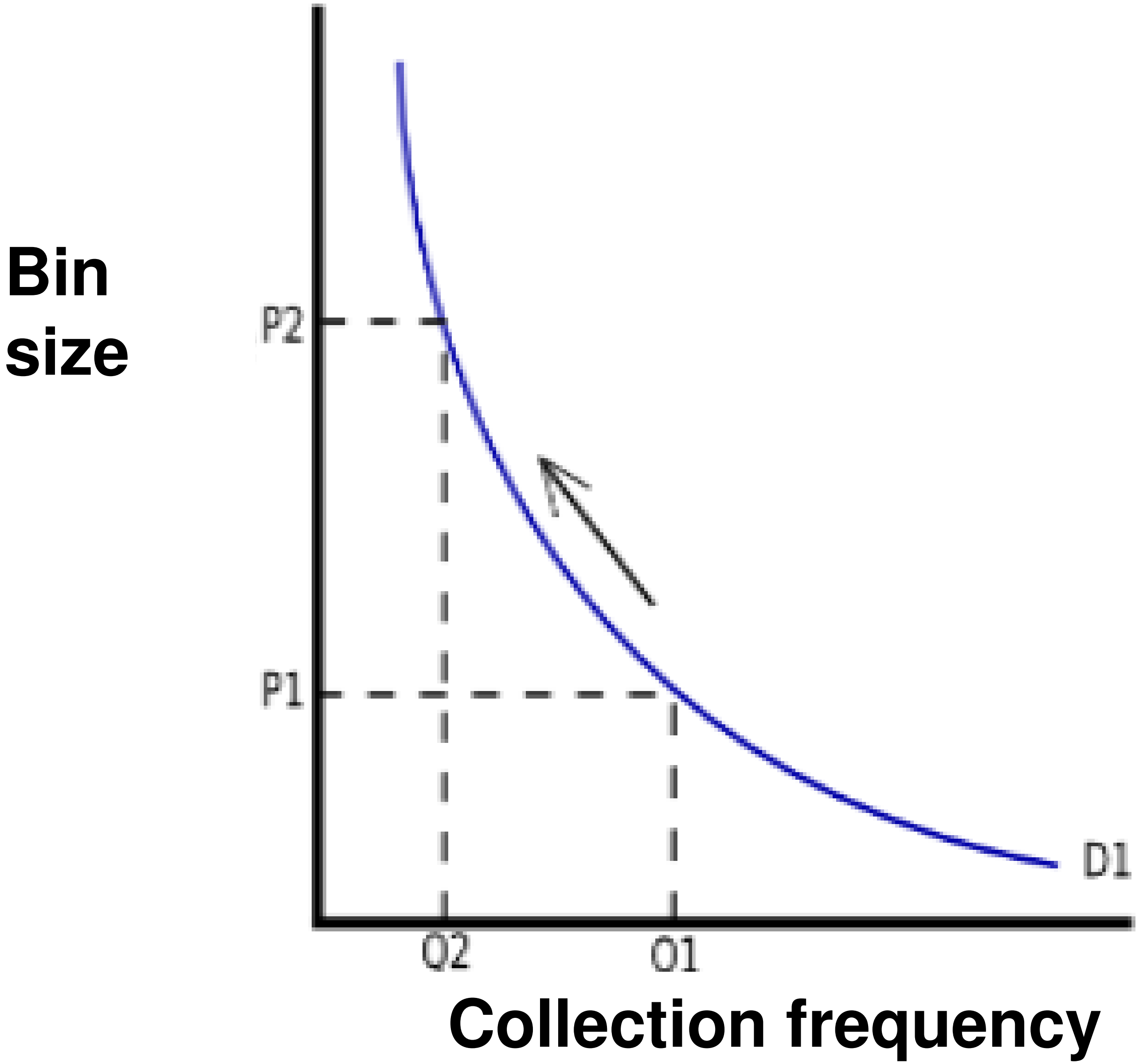
Richard Jones

University of Kent

SULO

SULO

PEAK
WASTE
TEL 01335 - 342276
PEAK
WASTE
TEL 01335 - 342276
⚠DO NOT
RUN OVER
WASHER HOSE

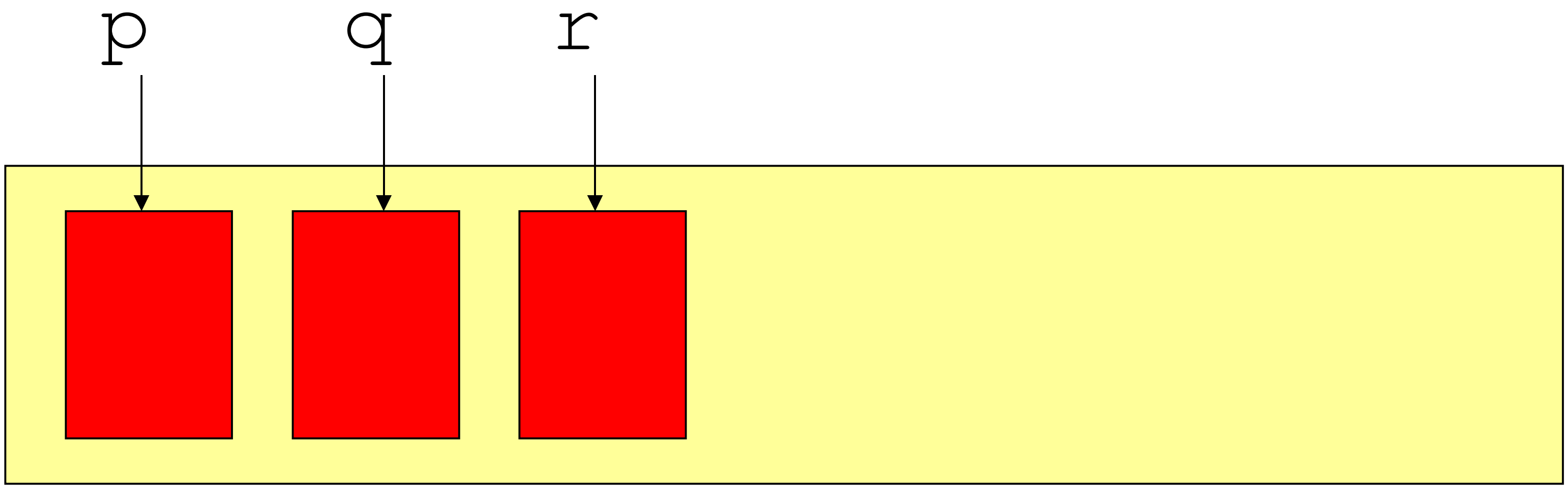

Bin size
P2
P1
D1
Q2
O1
Collection frequency

Actually …

- Garbage collection refers to automatic memory management for computer programs

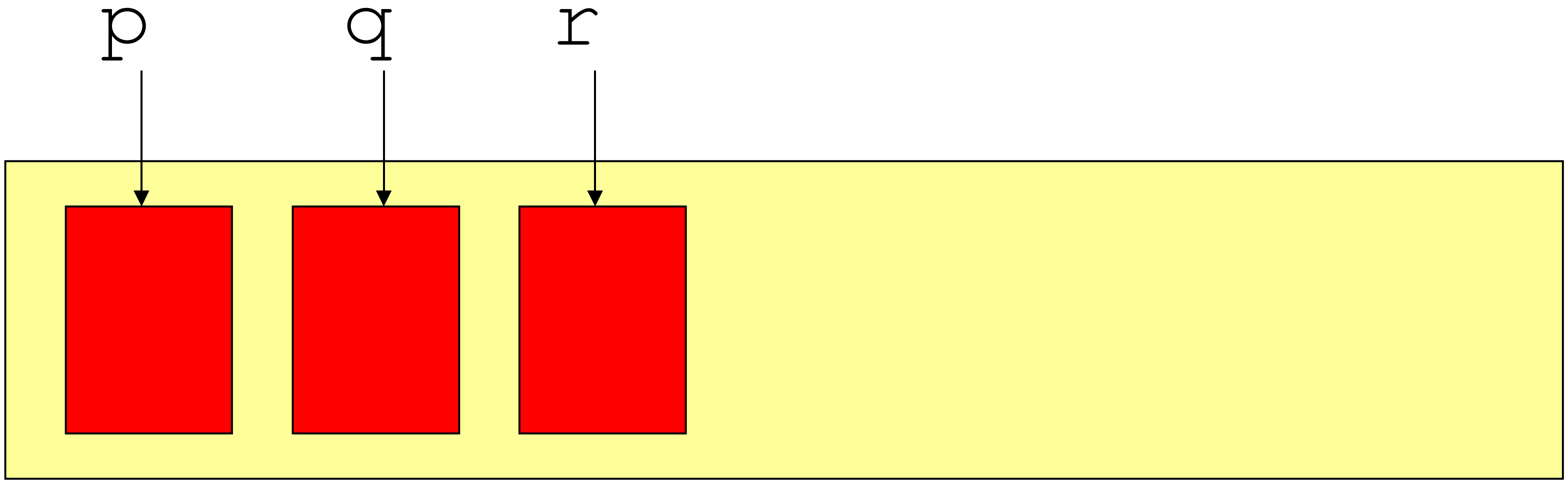
Manual memory management

```
p = malloc(…);
q = malloc(…);
r = malloc(…);
```



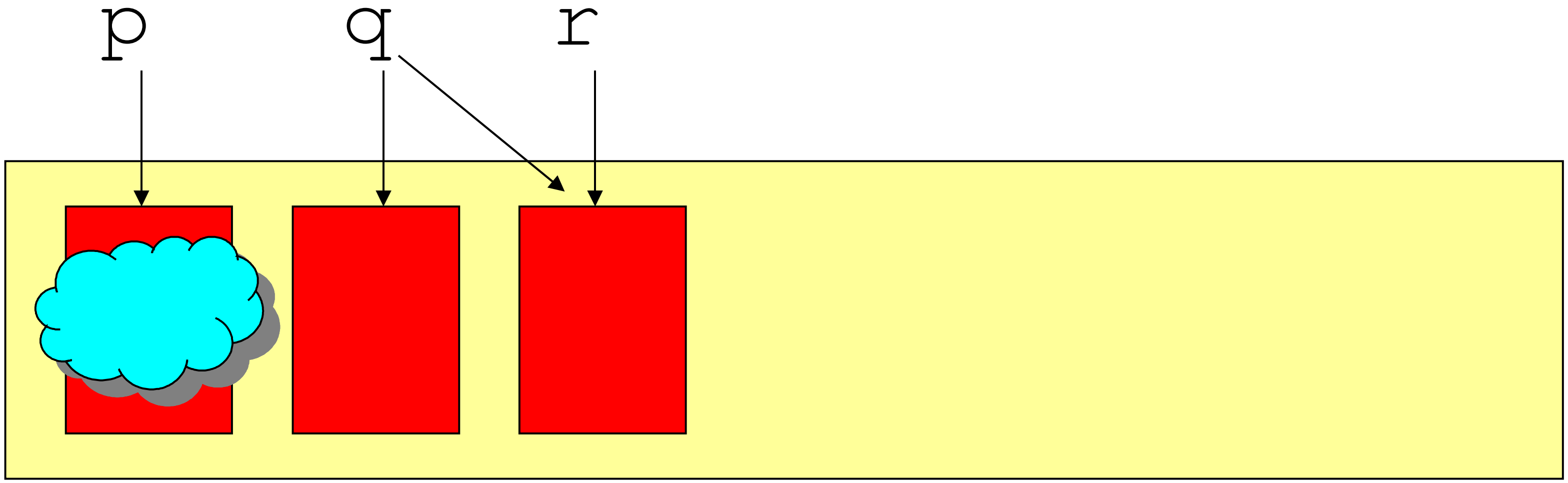
Manual memory management

```
free(p);
free(q);
free(r);
```



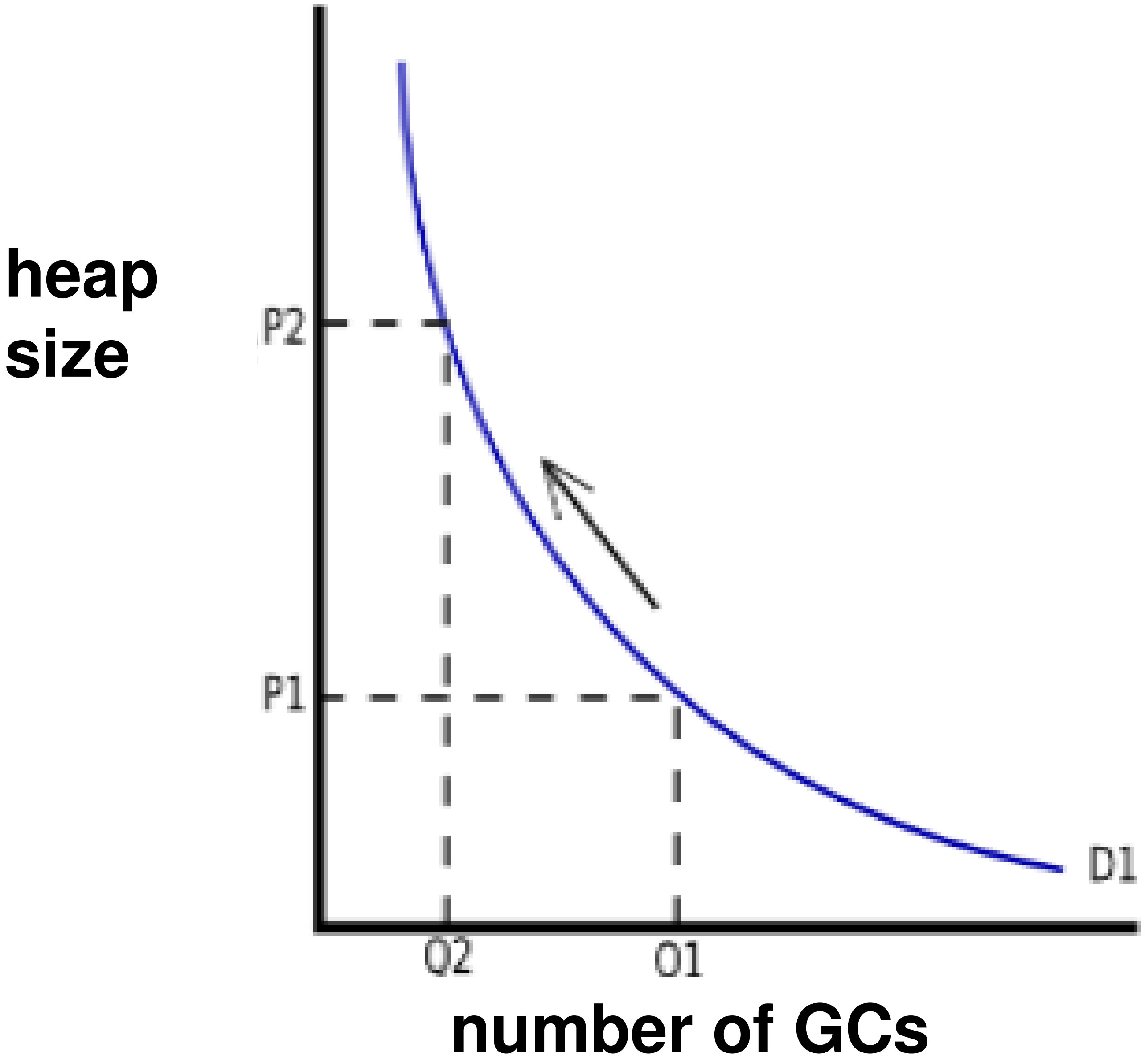
Problems

- dangling pointers
- double frees
- memory leaks (forgotten frees)



Automatic Memory Management

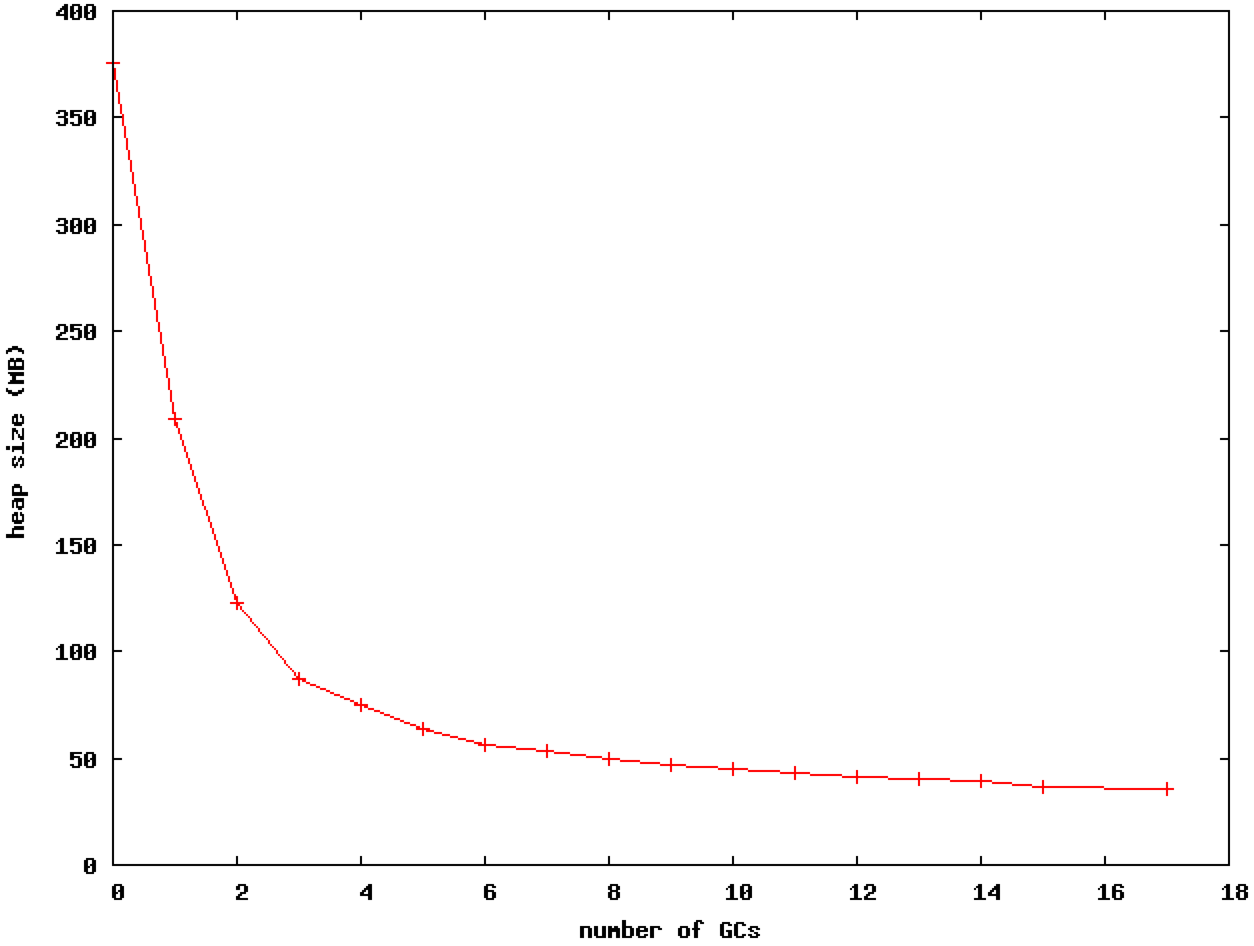
- no explicit `free()` required
- objects are collected when unreachable
- garbage collection (GC) finds unreachable objects and frees them
- GC invoked when application is running out of heap memory


heap size
P2
P1
D1
Q2
Q1
number of GCs

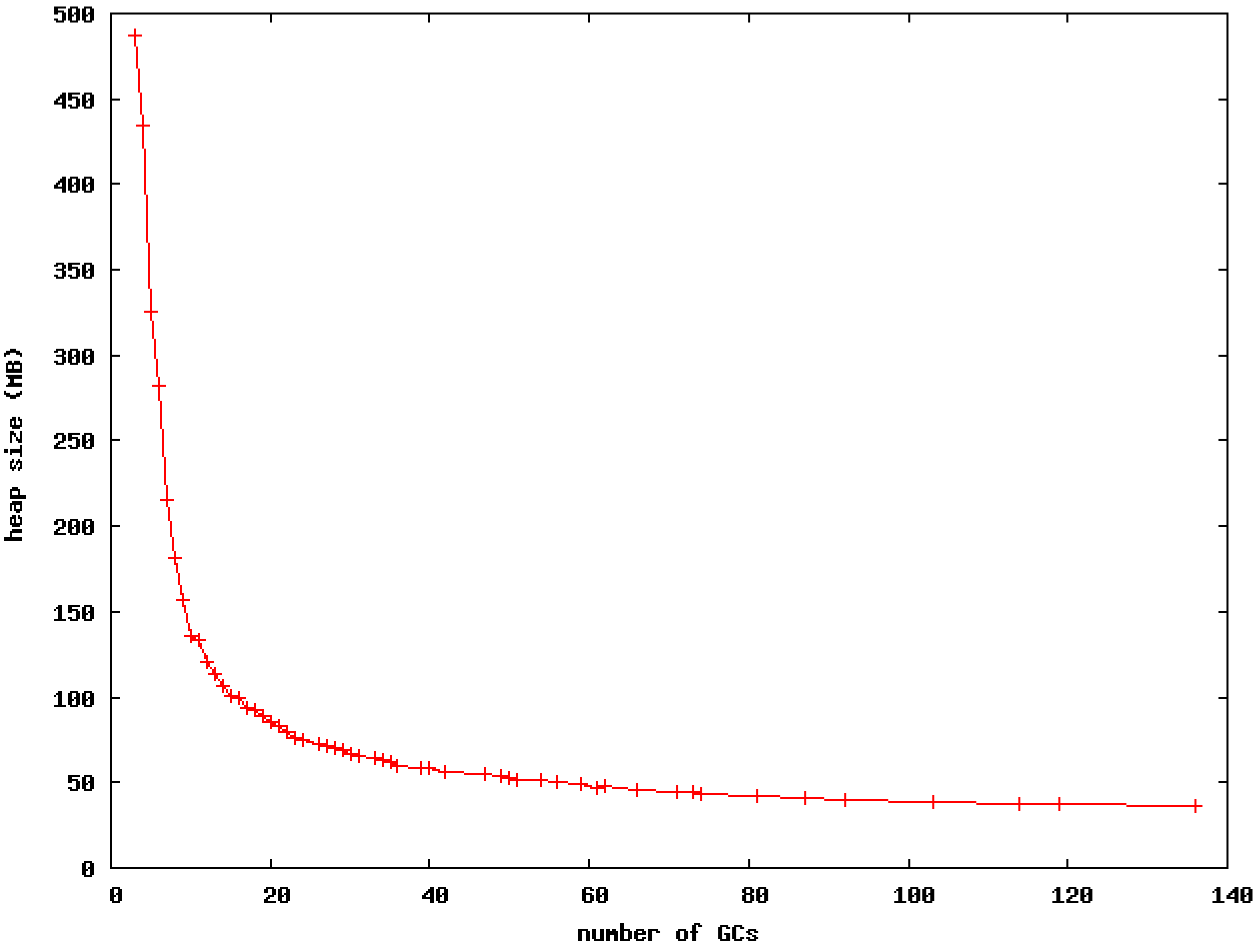
Empirical observations

- ‘real’ Java programs
  - DaCapo benchmarks v2006-10-MR2
- high-performance virtual machine
  - Jikes RVM v3.1.0
- modern architecture
  - Intel Core i7, x86_64 Linux 2.6.xxx
- run applications at a variety of fixed heap sizes

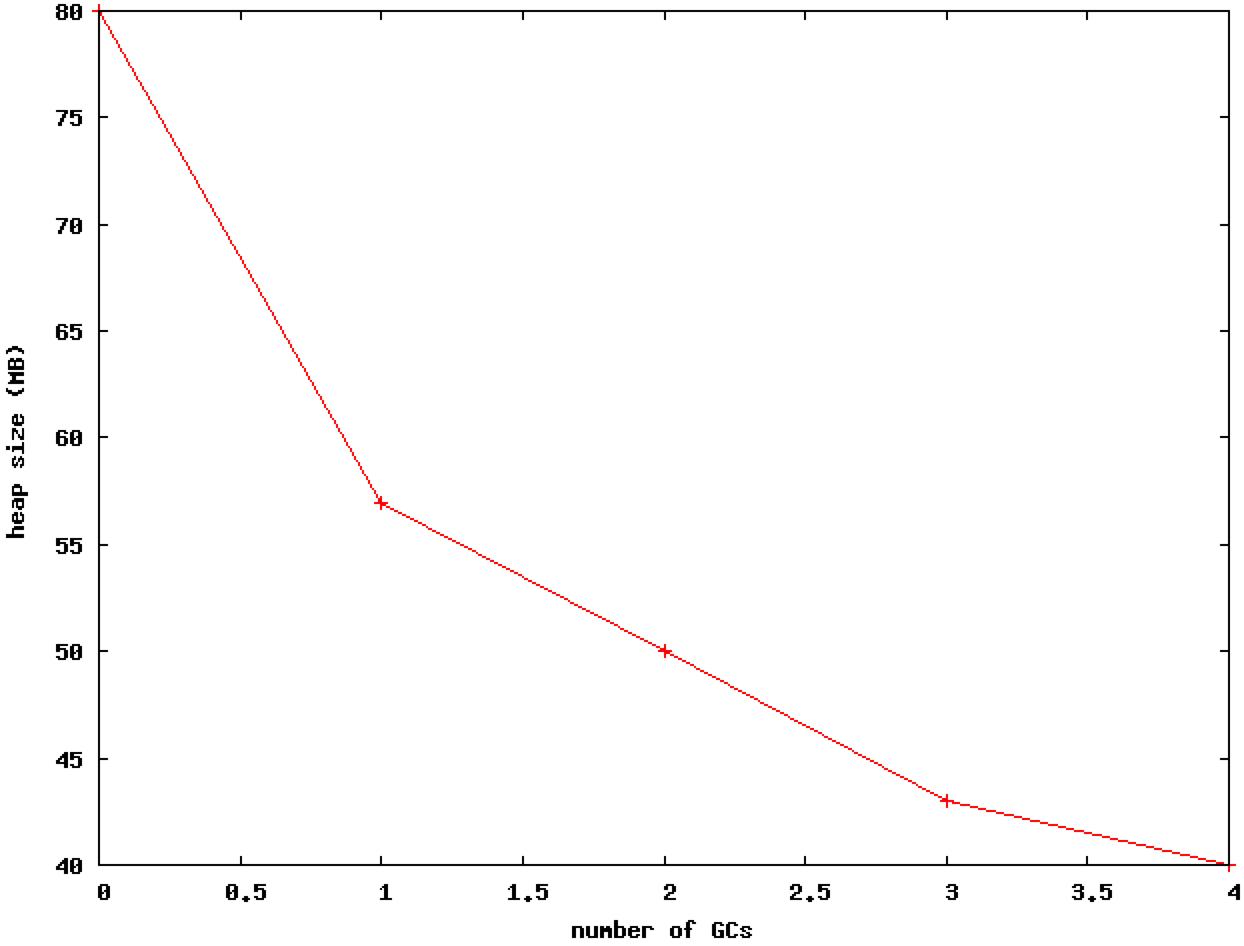
antlr


heap size (MB)
400
350
300
250
200
150
100
50
0
0
2
4
6
8
10
12
14
16
18
number of GCs

bloat

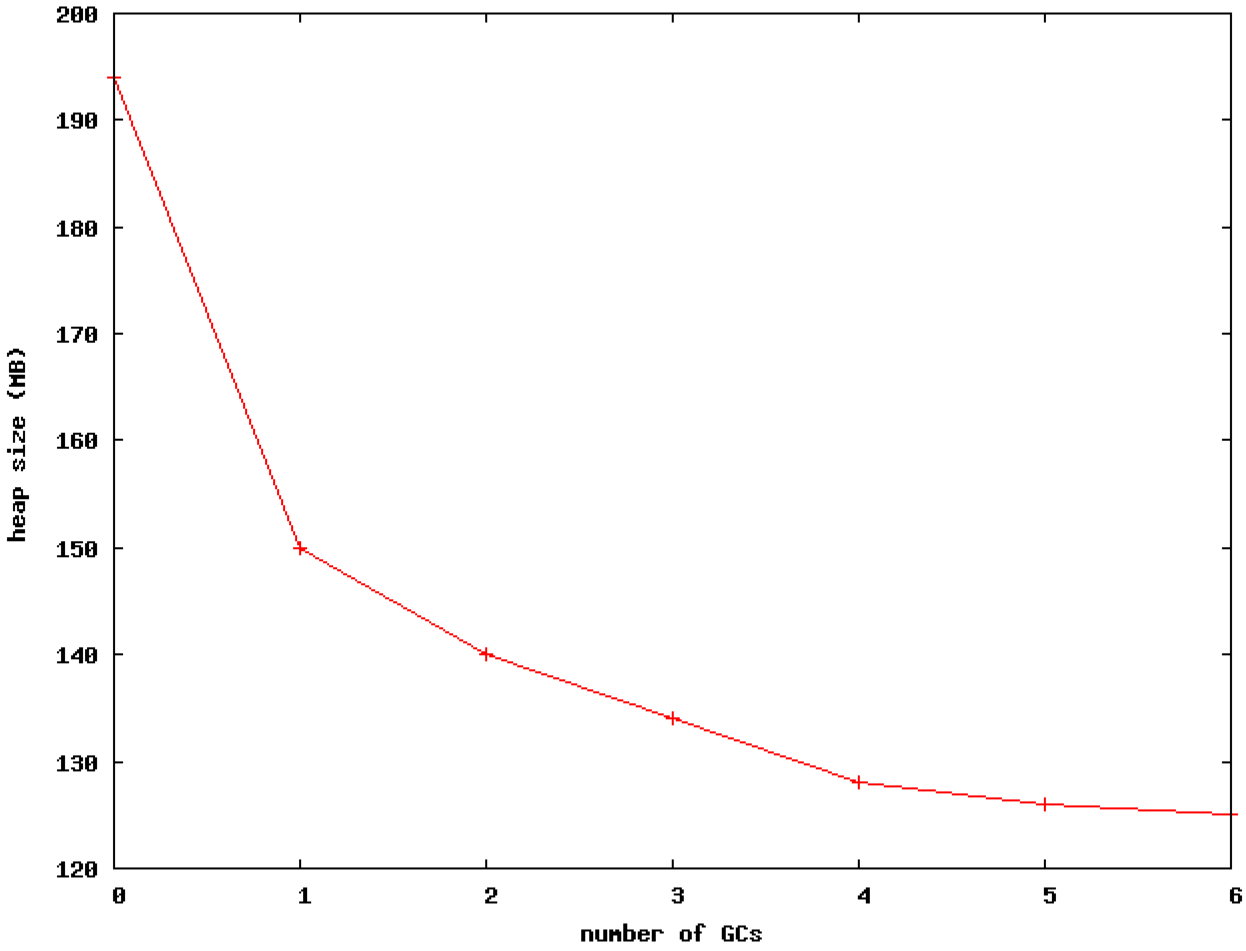

500
450
400
350
300
250
200
150
100
50
0
heap size (MB)
0
20
40
60
80
100
120
140
number of GCs

fop

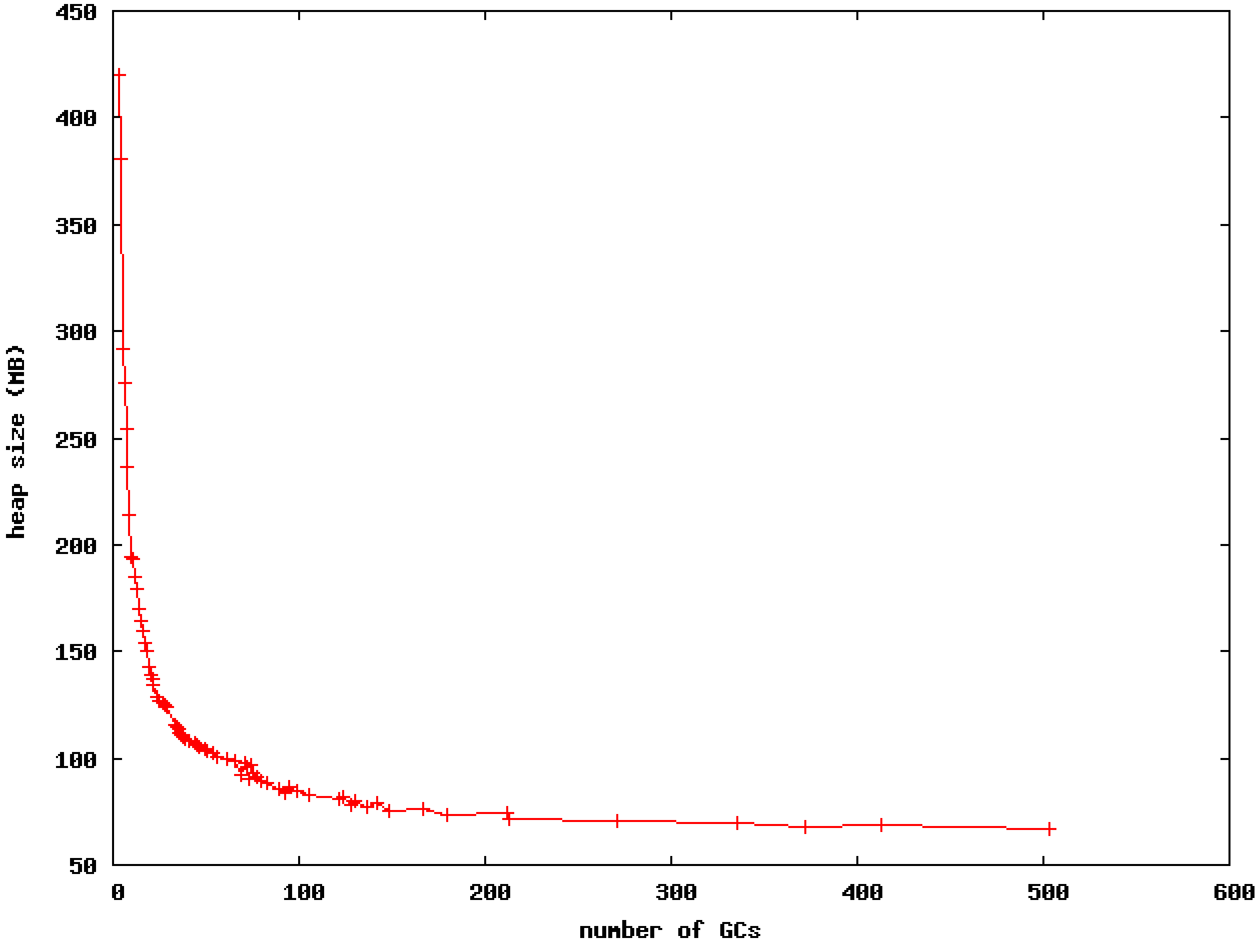


heap size (MB)
80
75
70
65
60
55
50
45
40
0
0.5
1
1.5
2
2.5
3
3.5
4
number of GCs

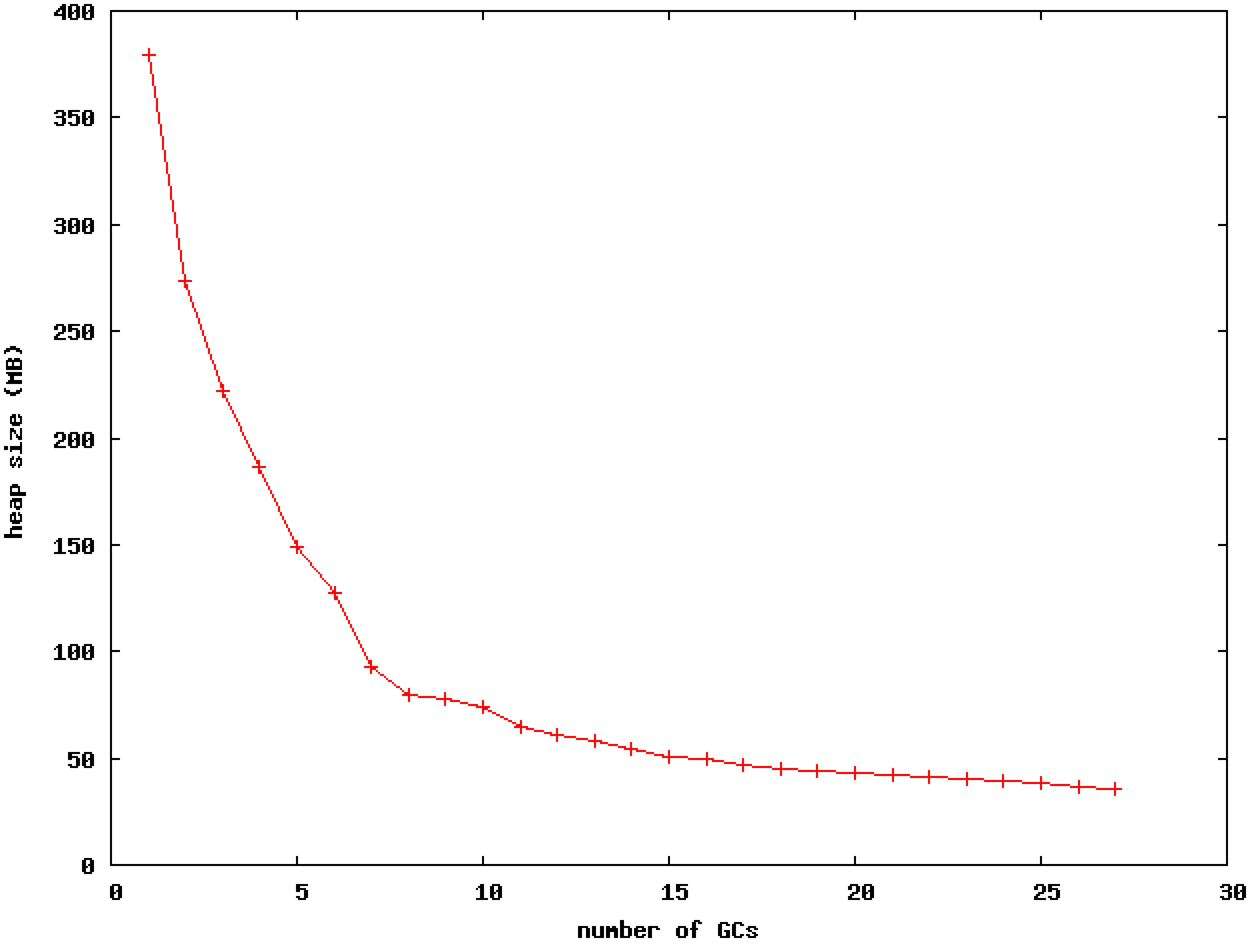
hsqldb


200
190
180
170
160
150
140
130
120
heap size (MB)
0
1
2
3
4
5
6
number of GCs

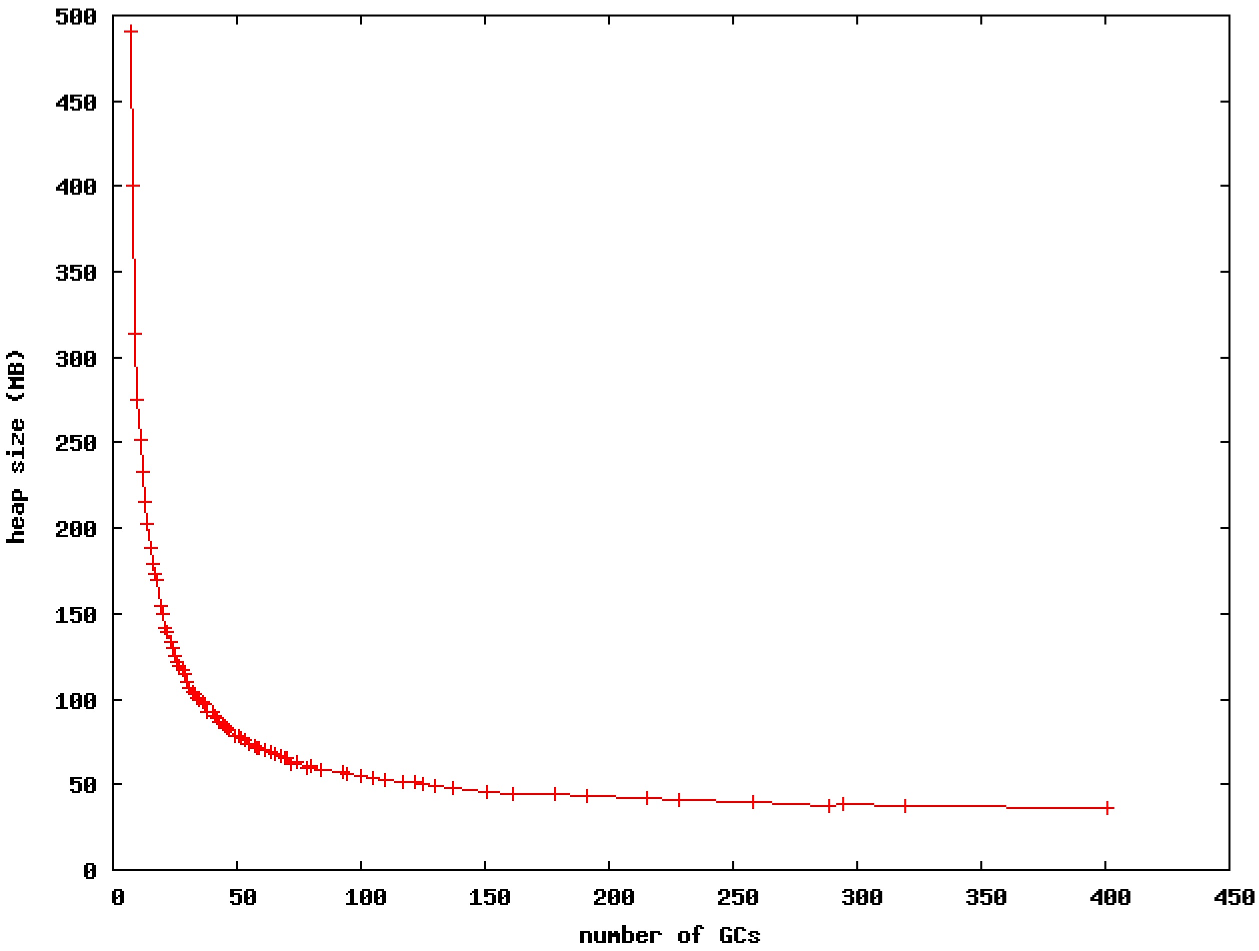
jython


heap size (MB)
450
400
350
300
250
200
150
100
50
0
100
200
300
400
500
600
number of GCs

luindex

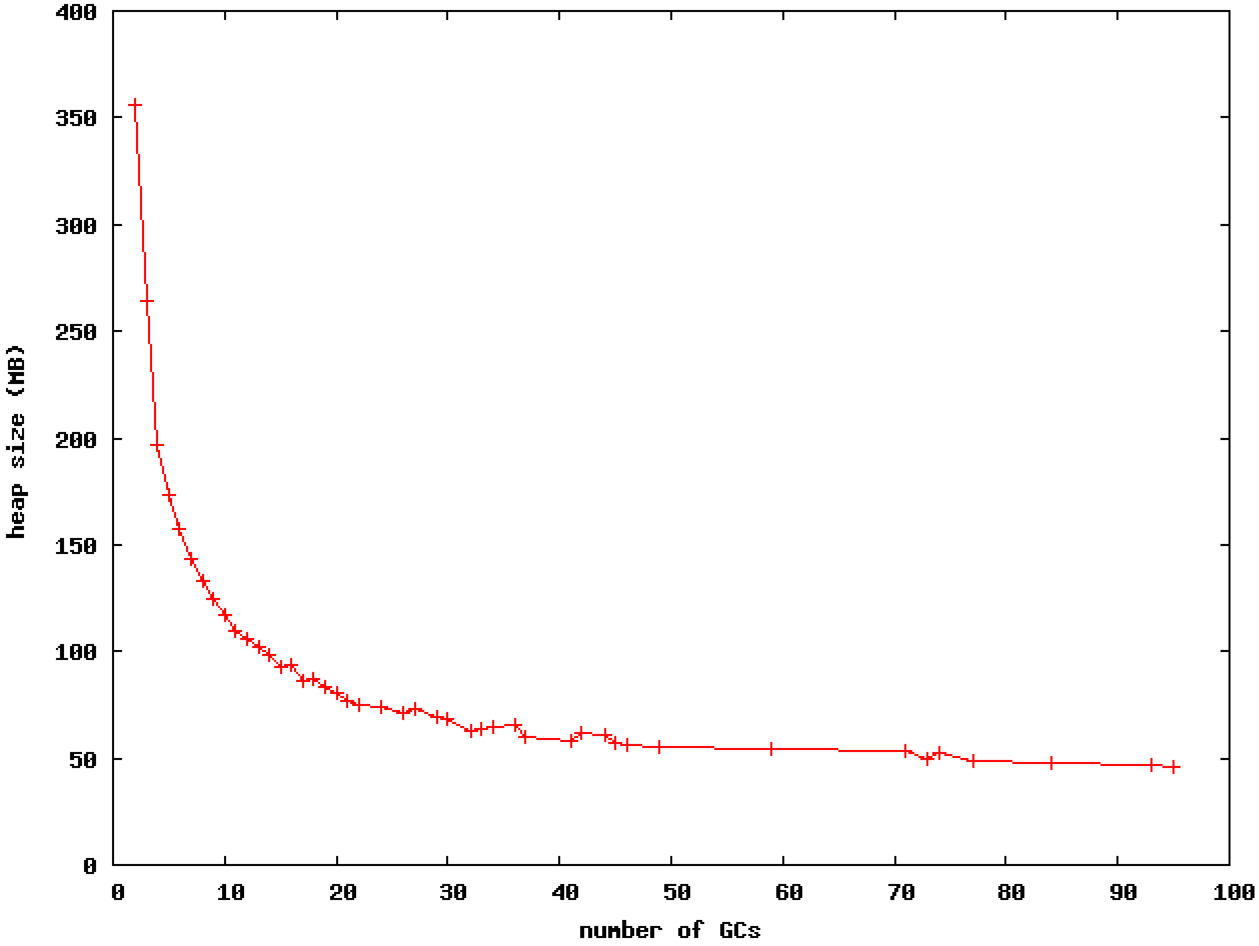

heap size (MB)
400
350
300
250
200
150
100
50
0
0
5
10
15
20
25
30
number of GCs

lusearch



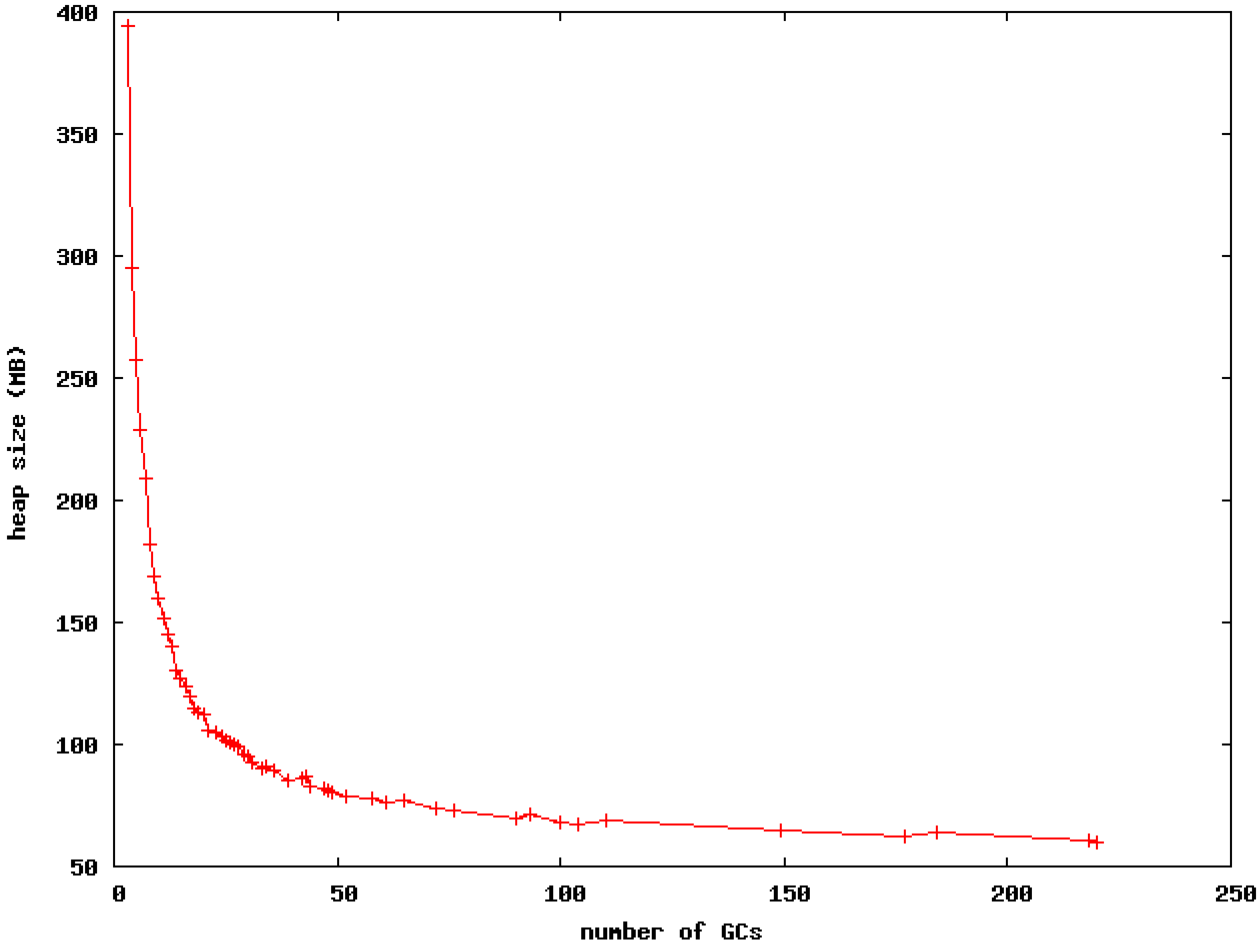
heap size (MB)
number of GCs
0
50
100
150
200
250
300
350
400
450
500
0
50
100
150
200
250
300
350
400
450

pmd



400
350
300
250
200
150
100
50
0
heap size (MB)
0
10
20
30
40
50
60
70
80
90
100
number of GCs

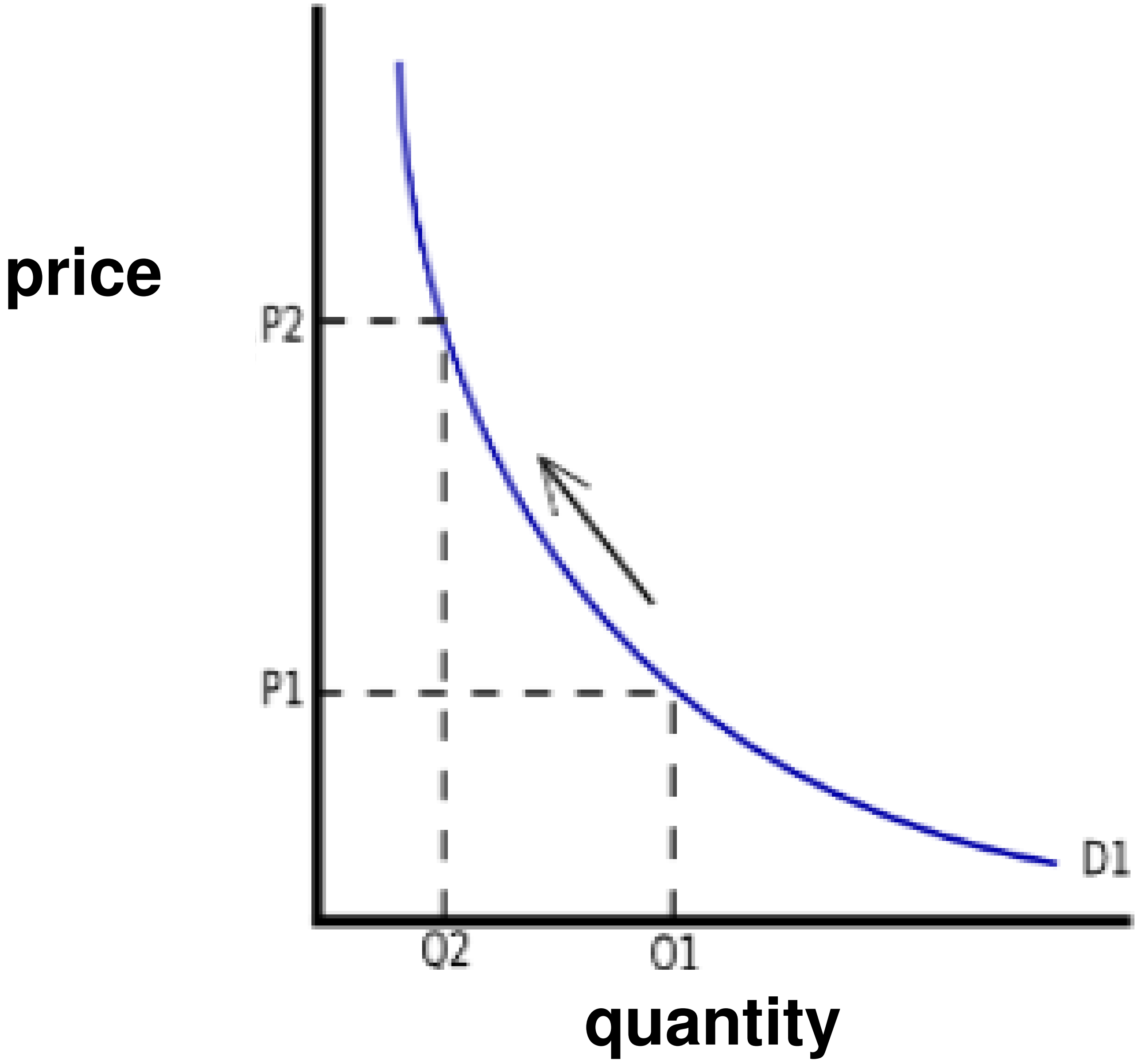
xalan



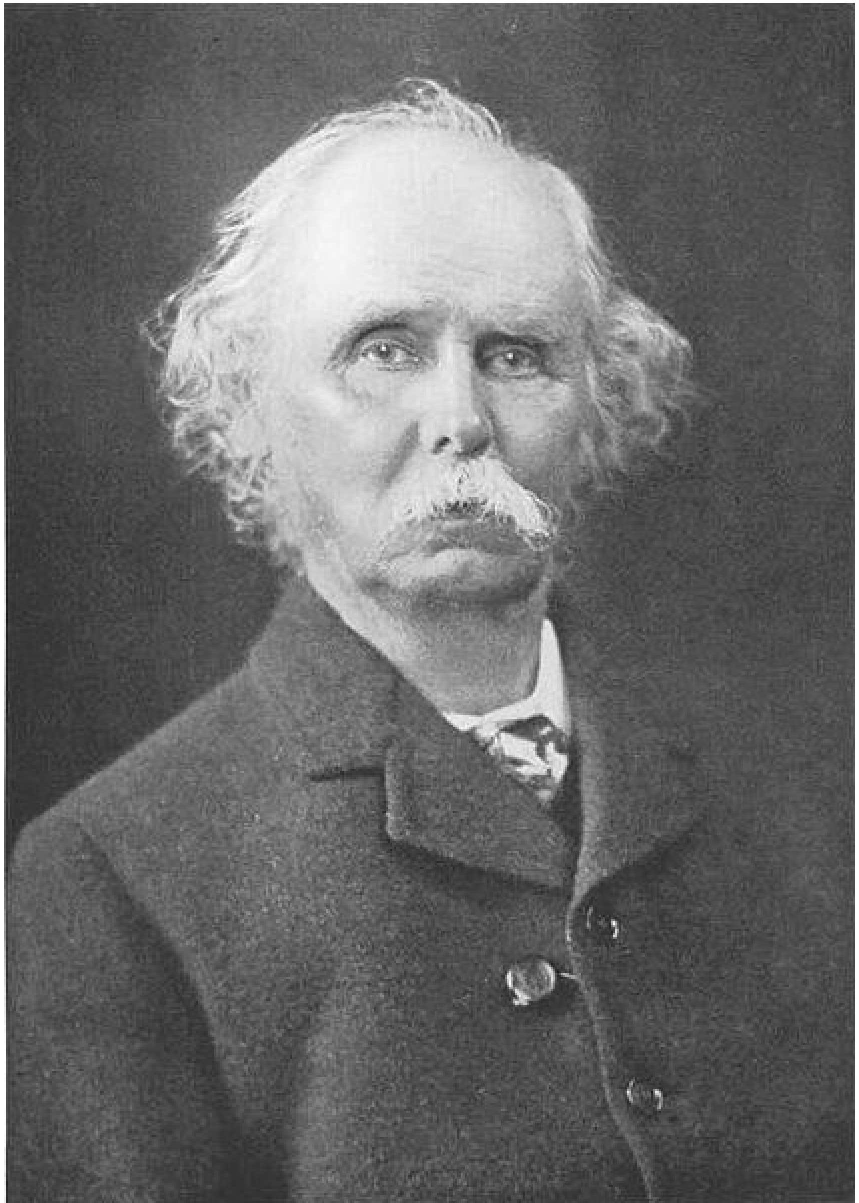
heap size (MB)
400
350
300
250
200
150
100
50
0
50
100
150
200
250
number of GCs

Now for the *serious* analogy

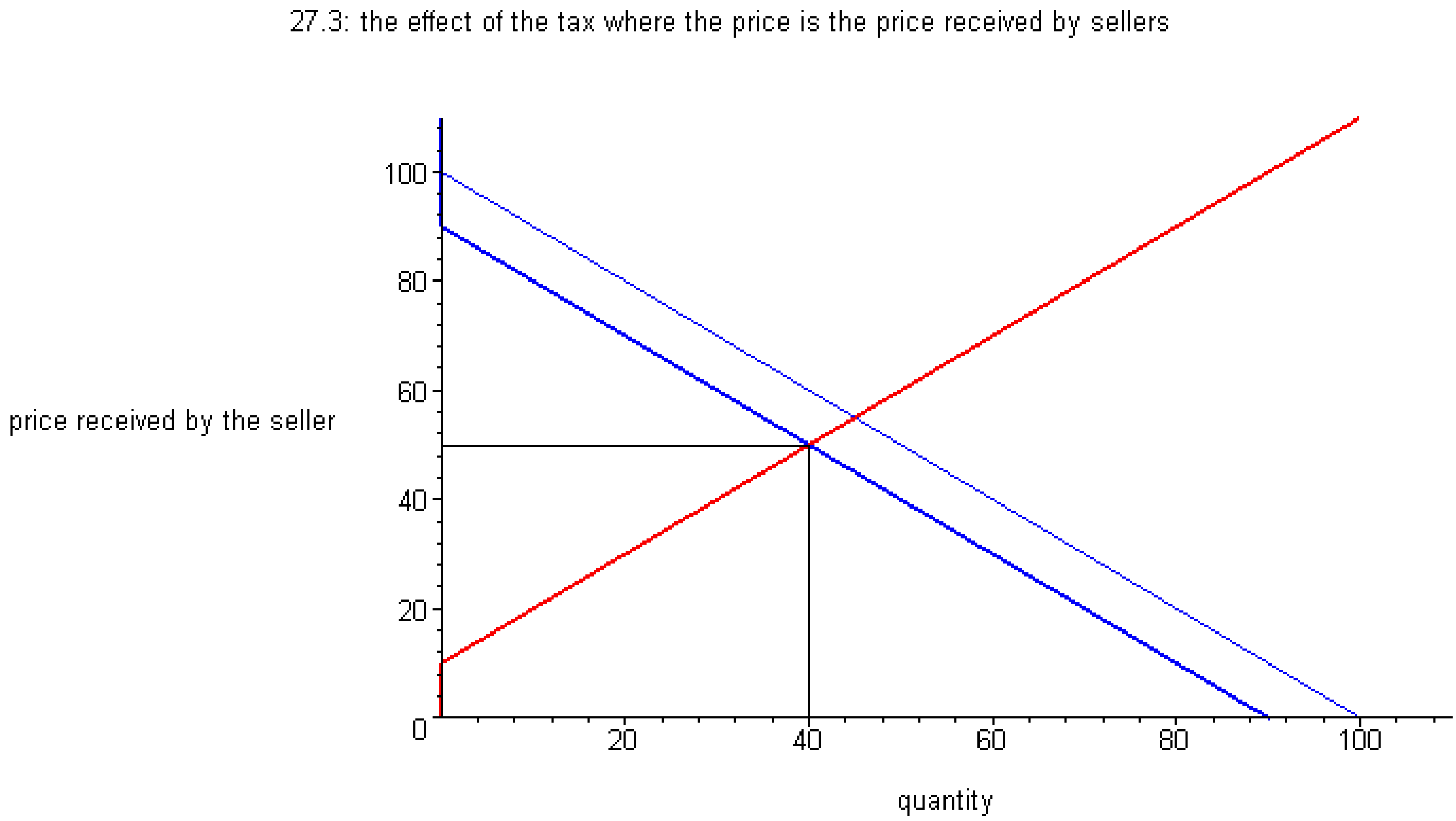
- micro-economics
  - – interactions in a single market
  - – (supply and demand of a single commodity)
  - – A demand curve shows the relationship between price of an item, and the quantity that consumers will purchase at a given price
  - – Law of demand:
    - more expensive => less required


price
P2
P1
D1
Q2
O1
quantity

Alfred Marshall: 1842-1924

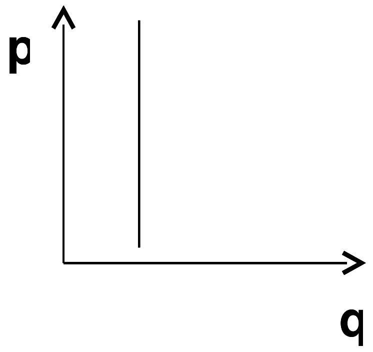


Effect of VAT (sales tax)

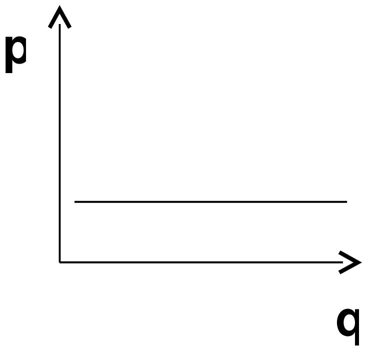


Elasticity

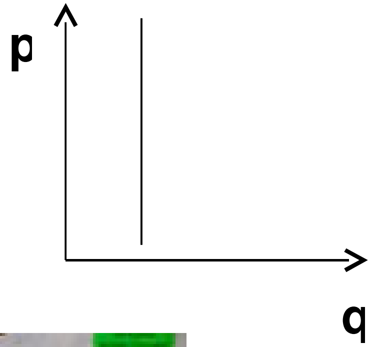
- Inelastic goods (E= 0)
  - demand is independent of price



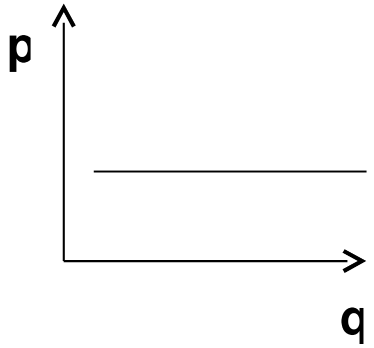
- Elastic goods (E=∞)
  - same price for all demand



Inelastic good



Elastic good



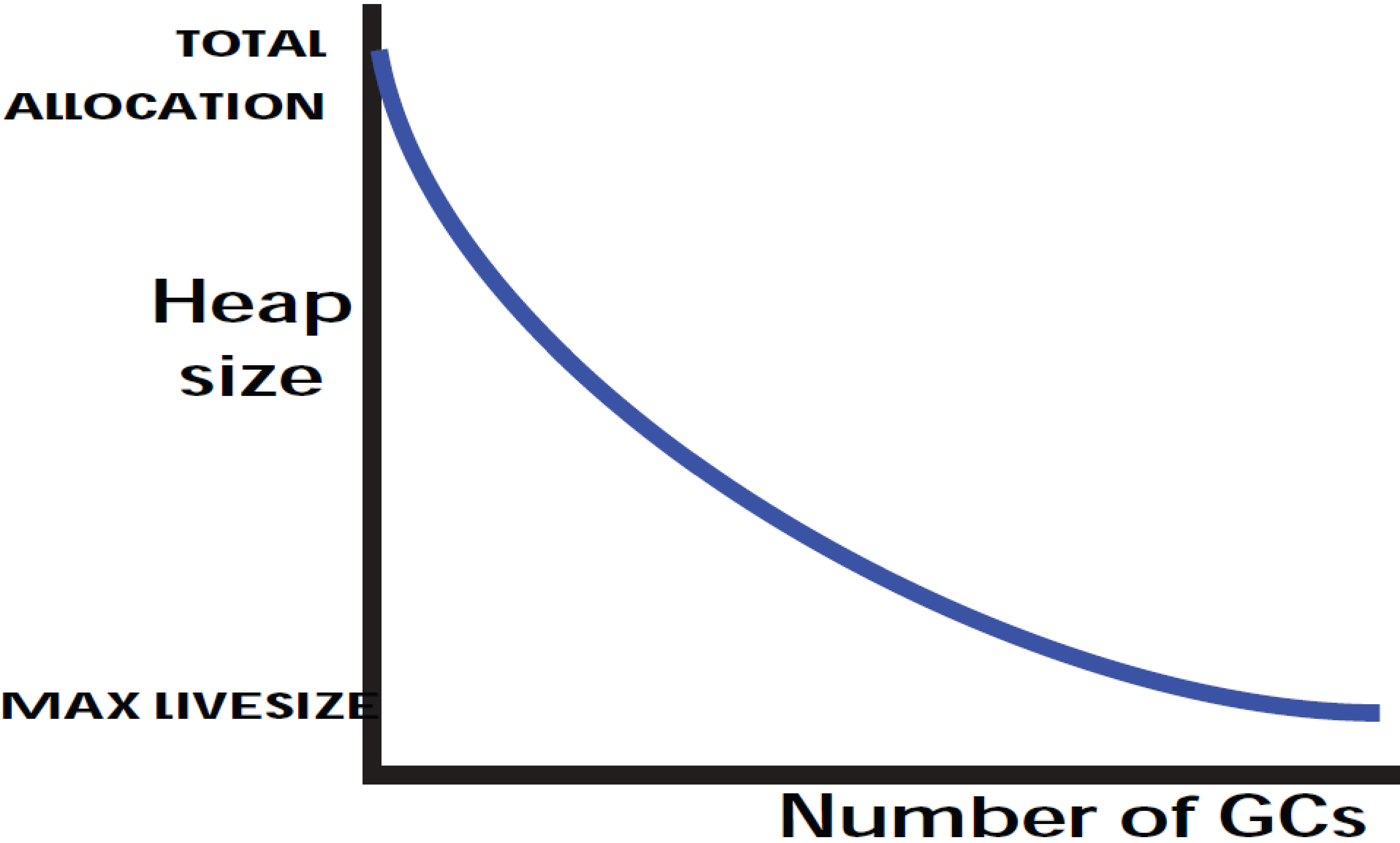
Calculating Elasticity

E = % change in quantity / % change in price

$$E = (dQ / dP)*(P/Q)$$

Apply this theory to GC

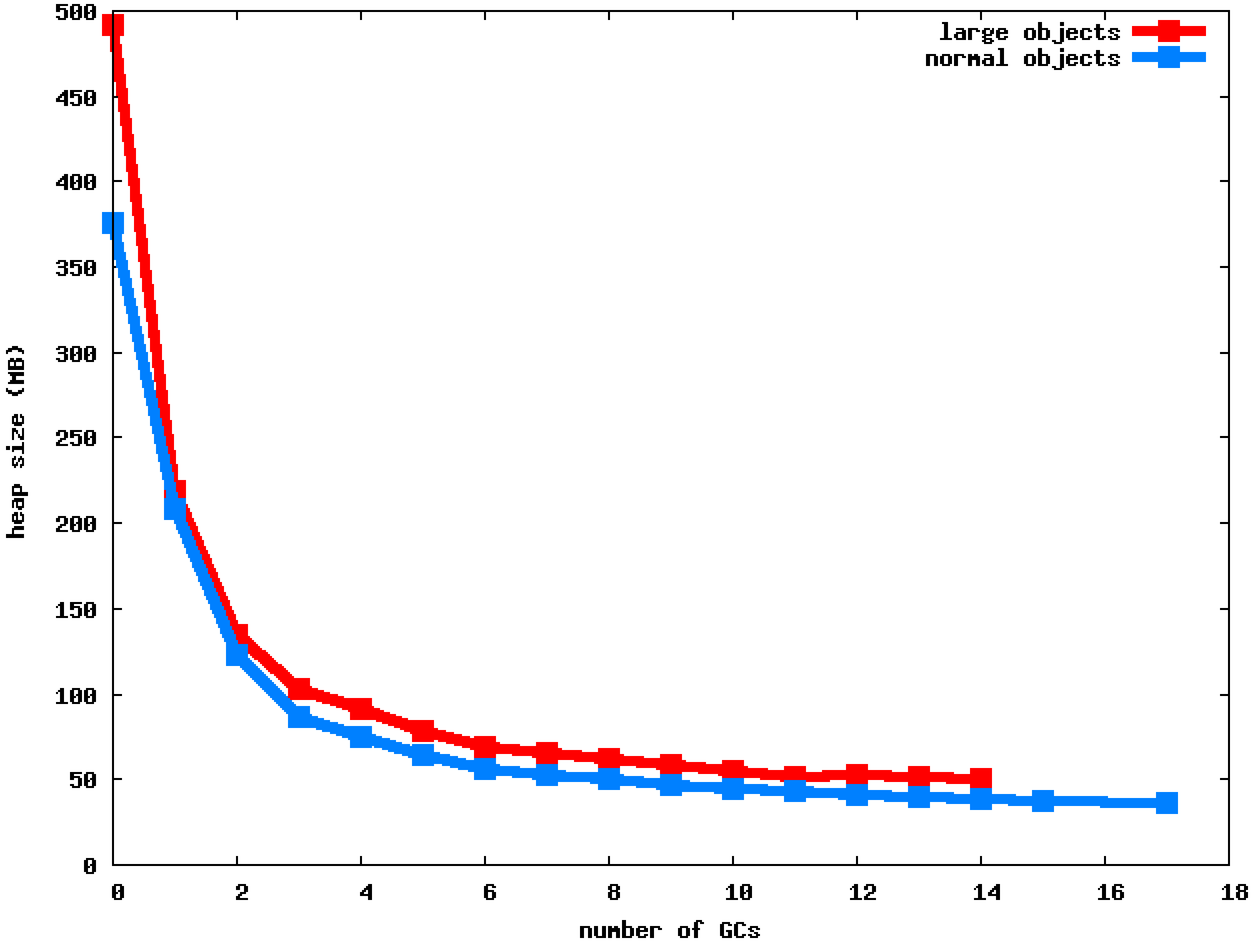
- demand curve == allocation curve
- price == heap size
- quantity == number of GCs


TOTAL
ALLOCATION
Heap
size
MAX LIVESIZE
Number of GCs

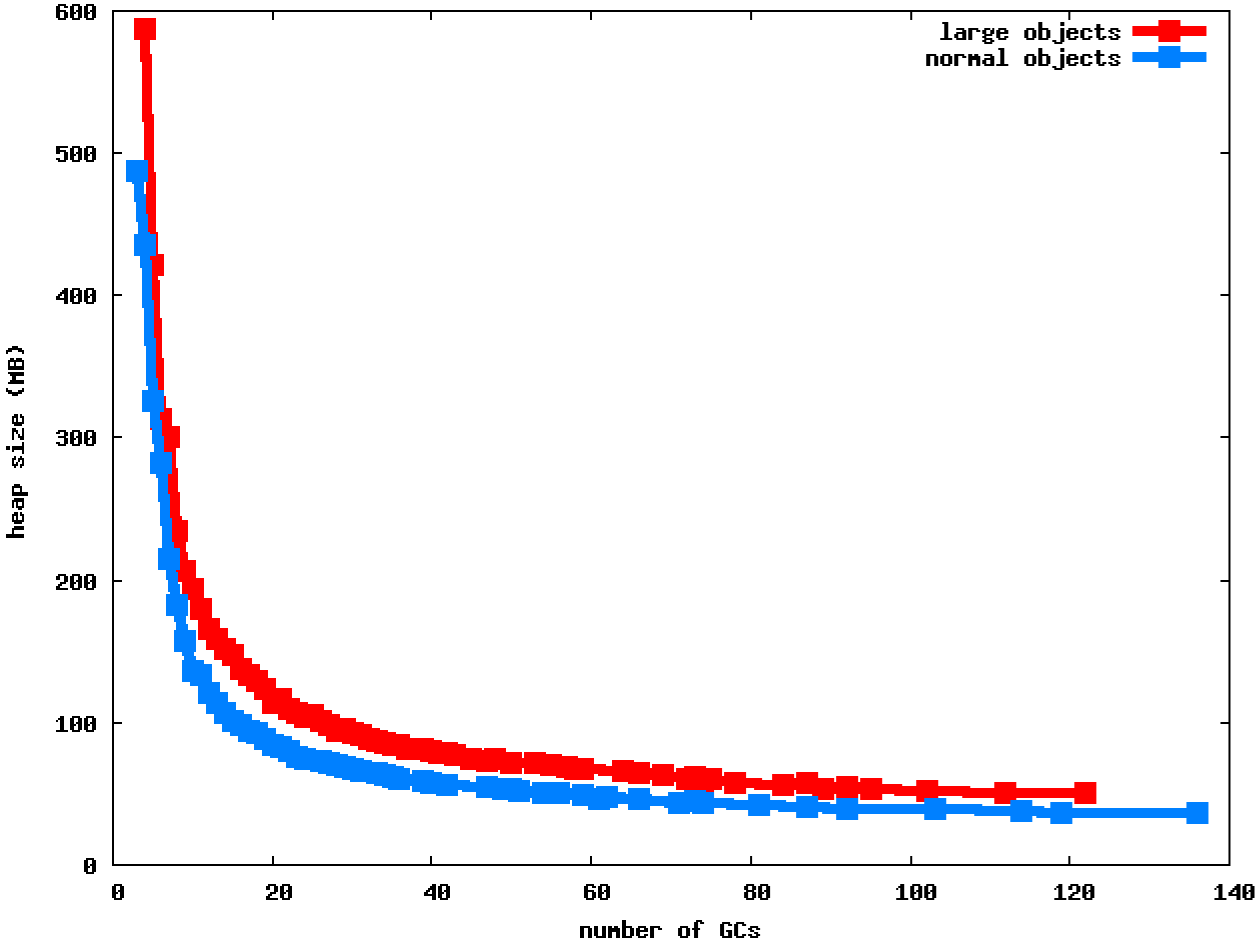
tax == book-keeping info

- increased “cost” per object
- simulate with increased object header size

antlr

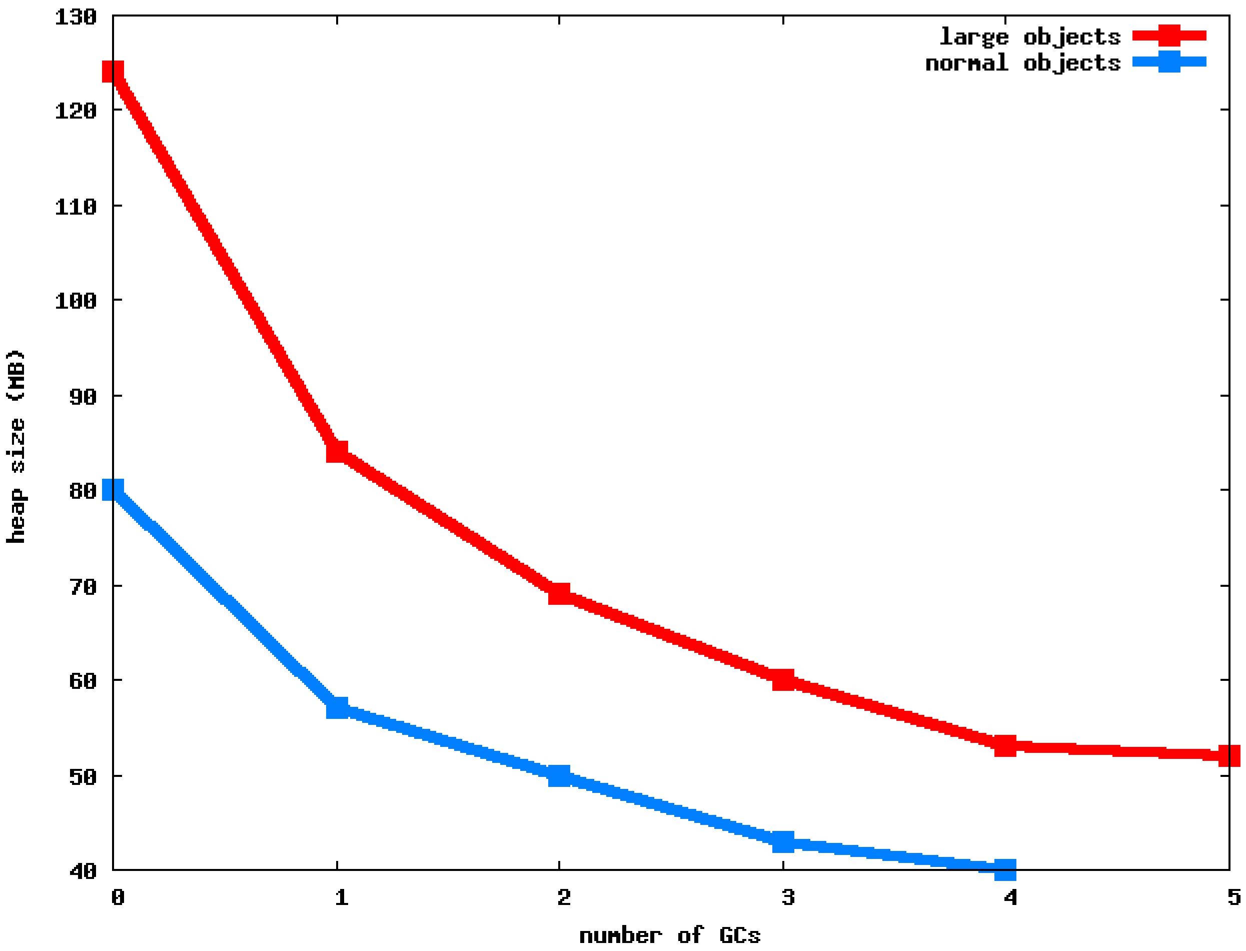

large objects
normal objects
heap size (MB)
500
450
400
350
300
250
200
150
100
50
0
0
2
4
6
8
10
12
14
16
18
number of GCs

bloat

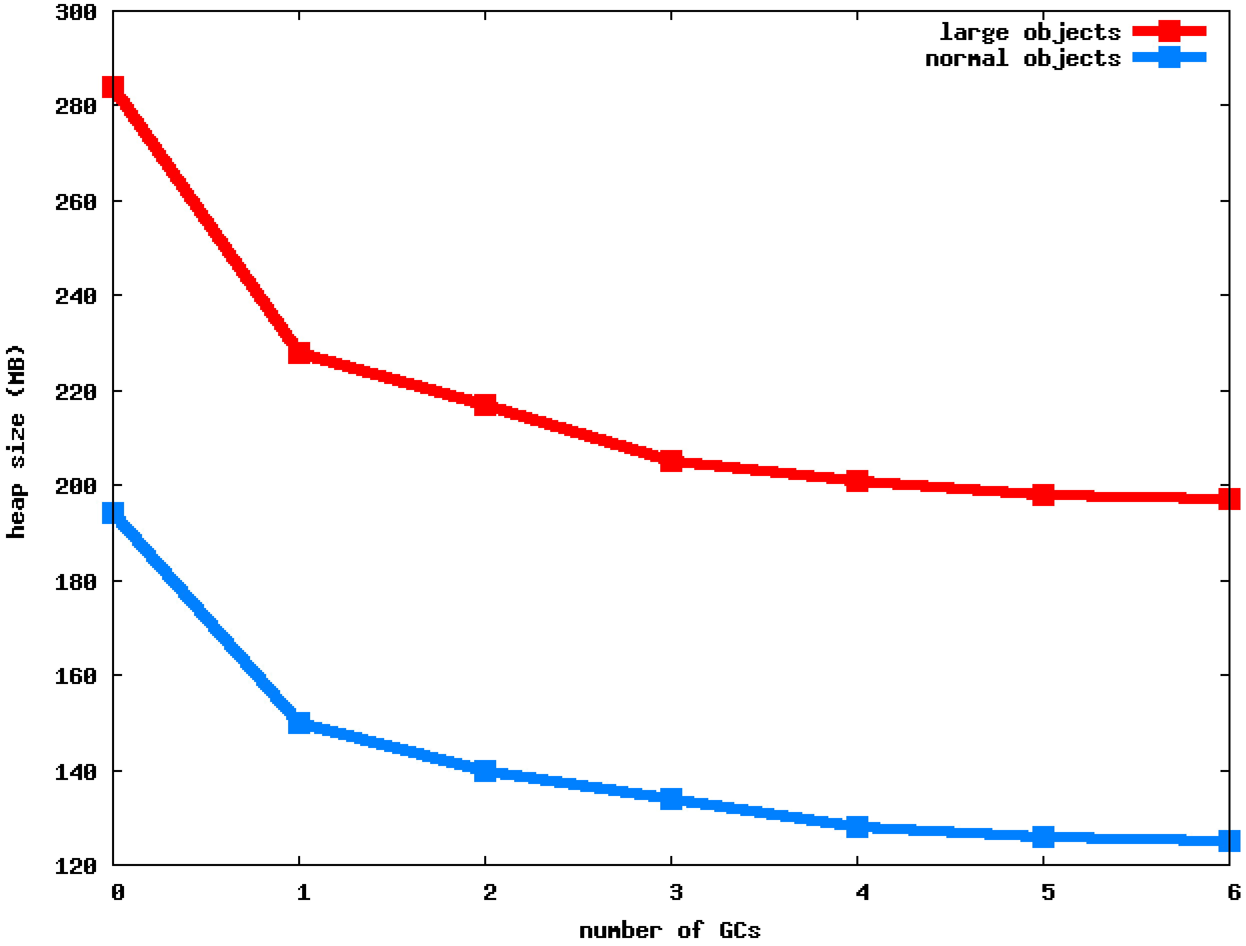


large objects
normal objects
heap size (MB)
600
500
400
300
200
100
0
0
20
40
60
80
100
120
140
number of GCs

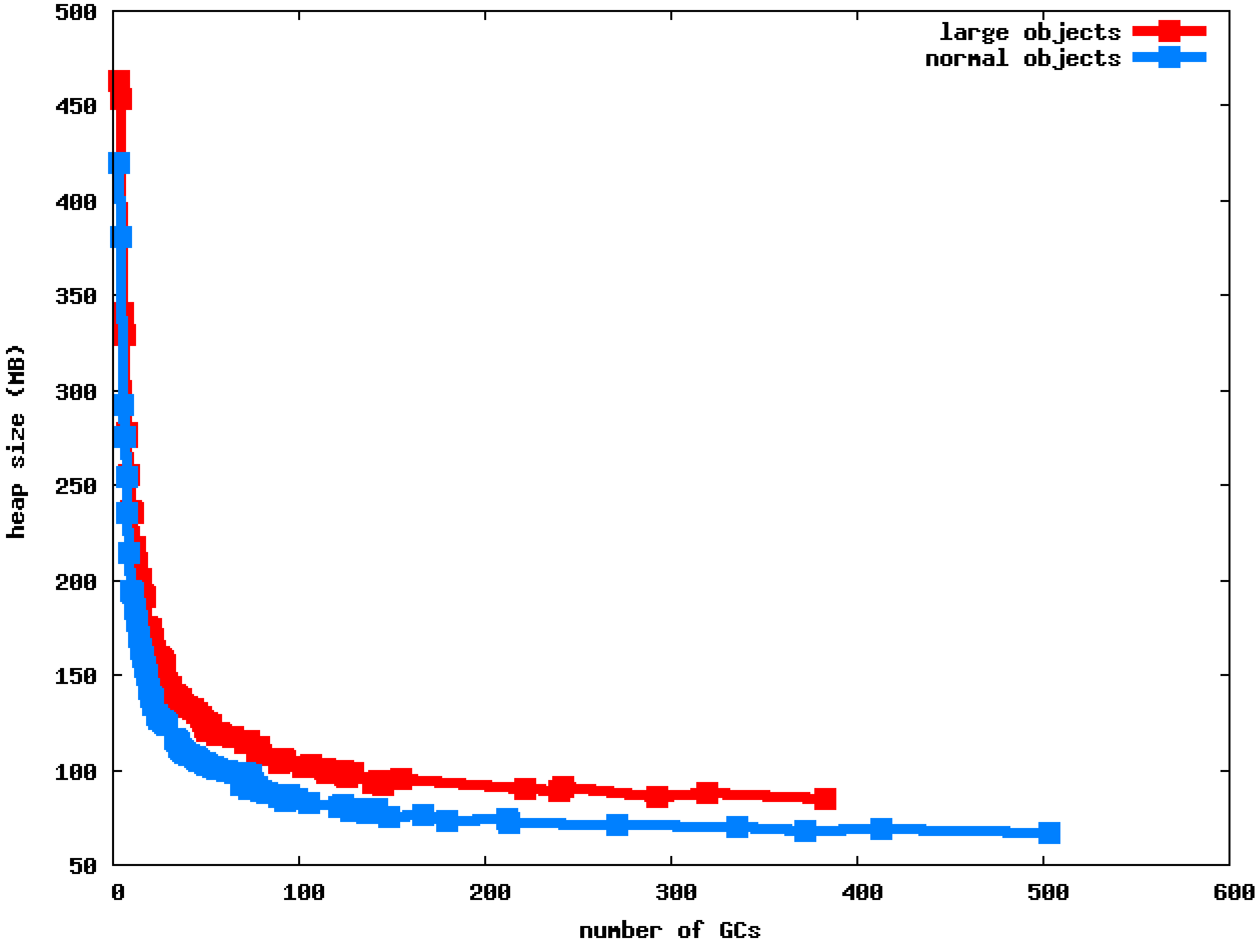
fop


130
120
110
100
90
80
70
60
50
40
heap size (MB)
0
1
2
3
4
5
number of GCs
large objects
normal objects

hsqldb

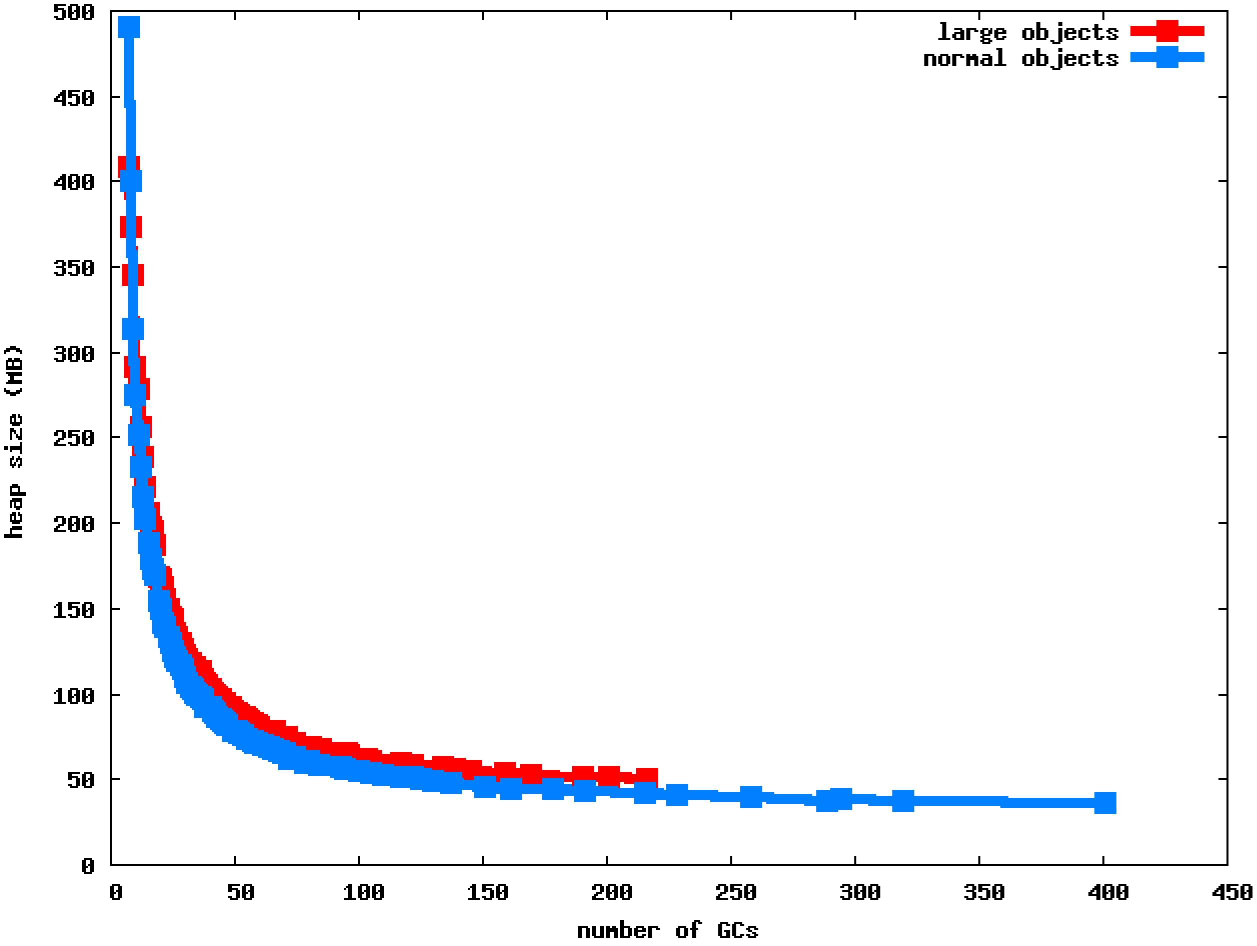

large objects
normal objects
heap size (MB)
300
280
260
240
220
200
180
160
140
120
0
1
2
3
4
5
6
number of GCs

jython

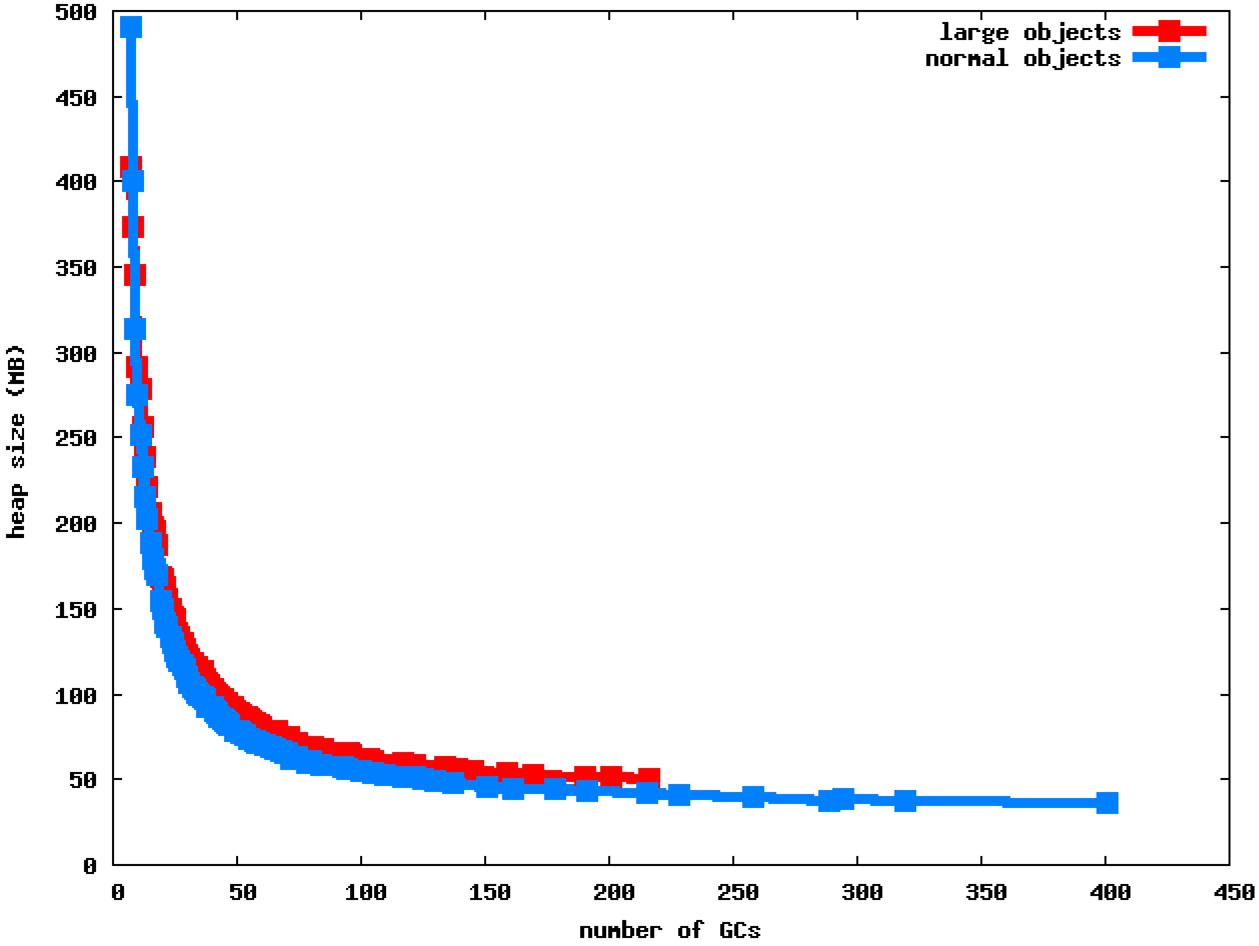


large objects
normal objects
heap size (MB)
number of GCs
50
100
150
200
250
300
350
400
450
500
0
100
200
300
400
500
600

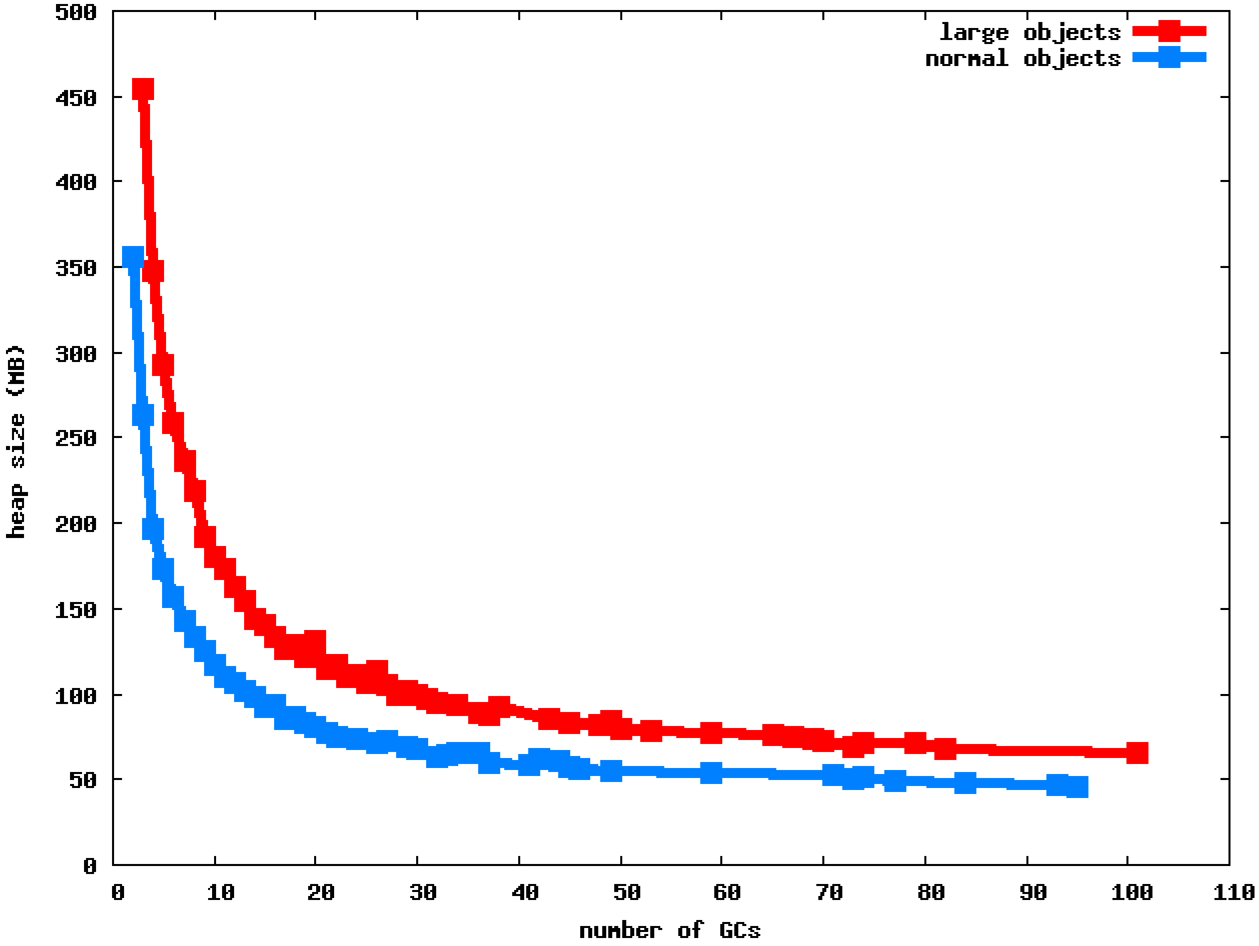
luindex


large objects
normal objects
heap size (MB)
number of GCs
0
50
100
150
200
250
300
350
400
450
500
0
50
100
150
200
250
300
350
400
450

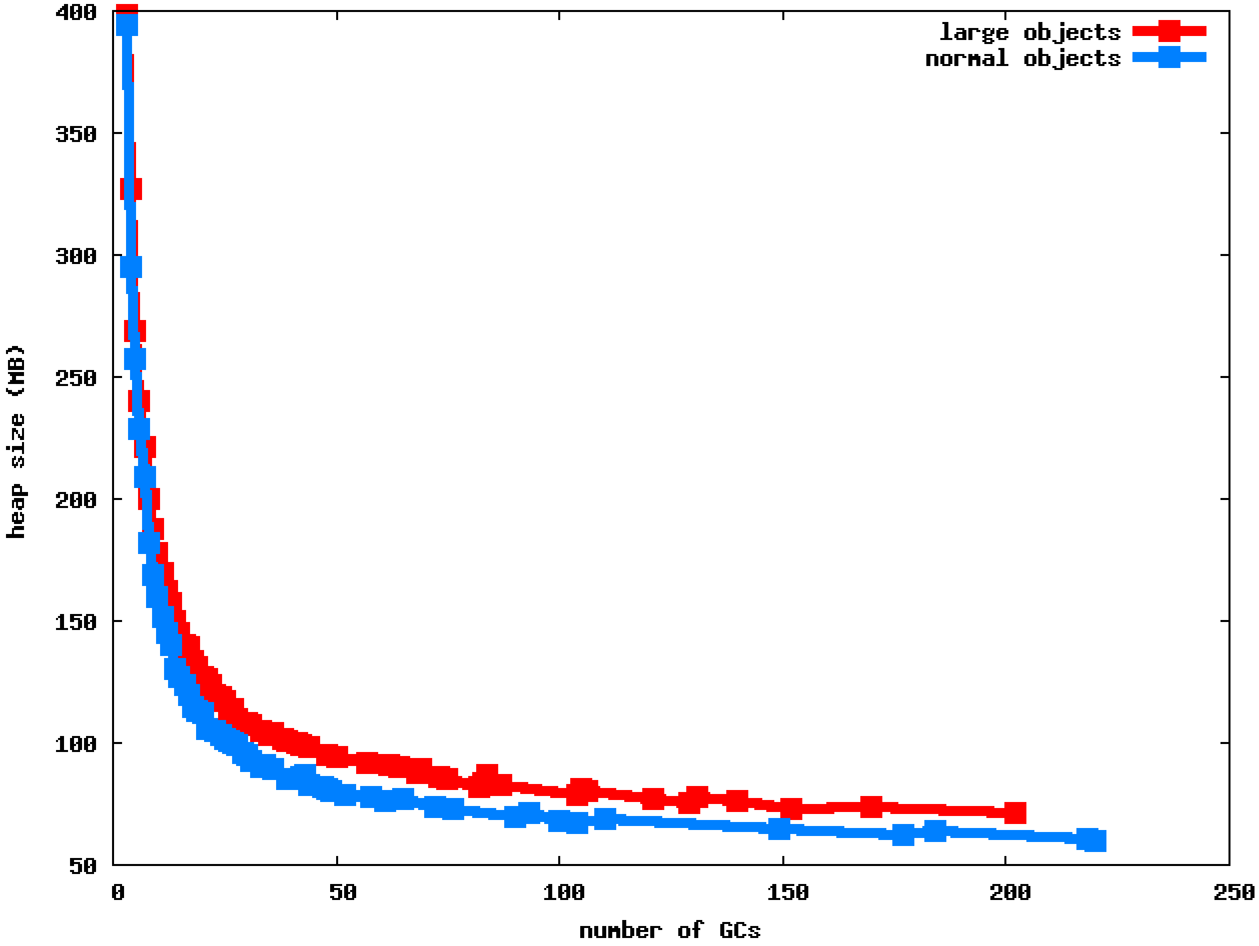
lusearch


large objects
normal objects
heap size (MB)
number of GCs
0
50
100
150
200
250
300
350
400
450
500
0
50
100
150
200
250
300
350
400
450

pmd


large objects
normal objects
heap size (MB)
500
450
400
350
300
250
200
150
100
50
0
0
10
20
30
40
50
60
70
80
90
100
110
number of GCs

xalan


large objects
normal objects
heap size (MB)
400
350
300
250
200
150
100
50
0
50
100
150
200
250
number of GCs

Allocation Elasticity

- *H* = heap size
- *G* = # garbage collections

- *E = % change in G / % change in H*

- *E = (dG/dH) * (H/G)*

Heap Growth Management

- Java applications often executed in variable size heap
- JVM controls heap size changes, based on GC load and live ratio.
- Heap size change is not controlled by user
  - (except possible –Xms –Xmx)

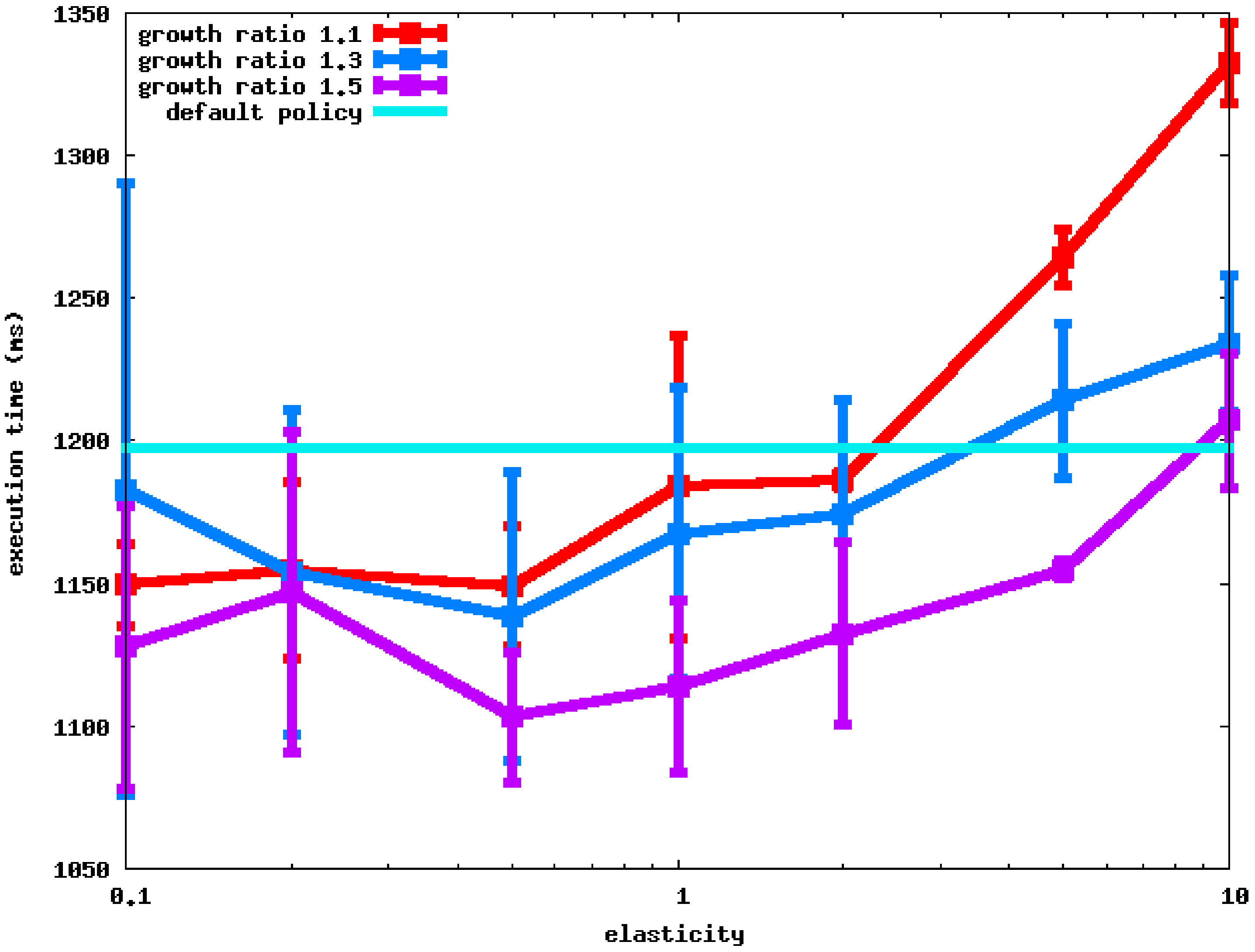
Current Elasticity of a running program in a variable sized heap

$$\text{currElasticity} = -1.\underbrace{\frac{\text{numGCs since last heap expansion}}{\text{heap size change at last expansion}}}_{dG\,/\,dH} \cdot \underbrace{\frac{\text{heap size before last expansion}}{\text{numGCs from start to last heap growth}}}_{H\,/\,G}$$

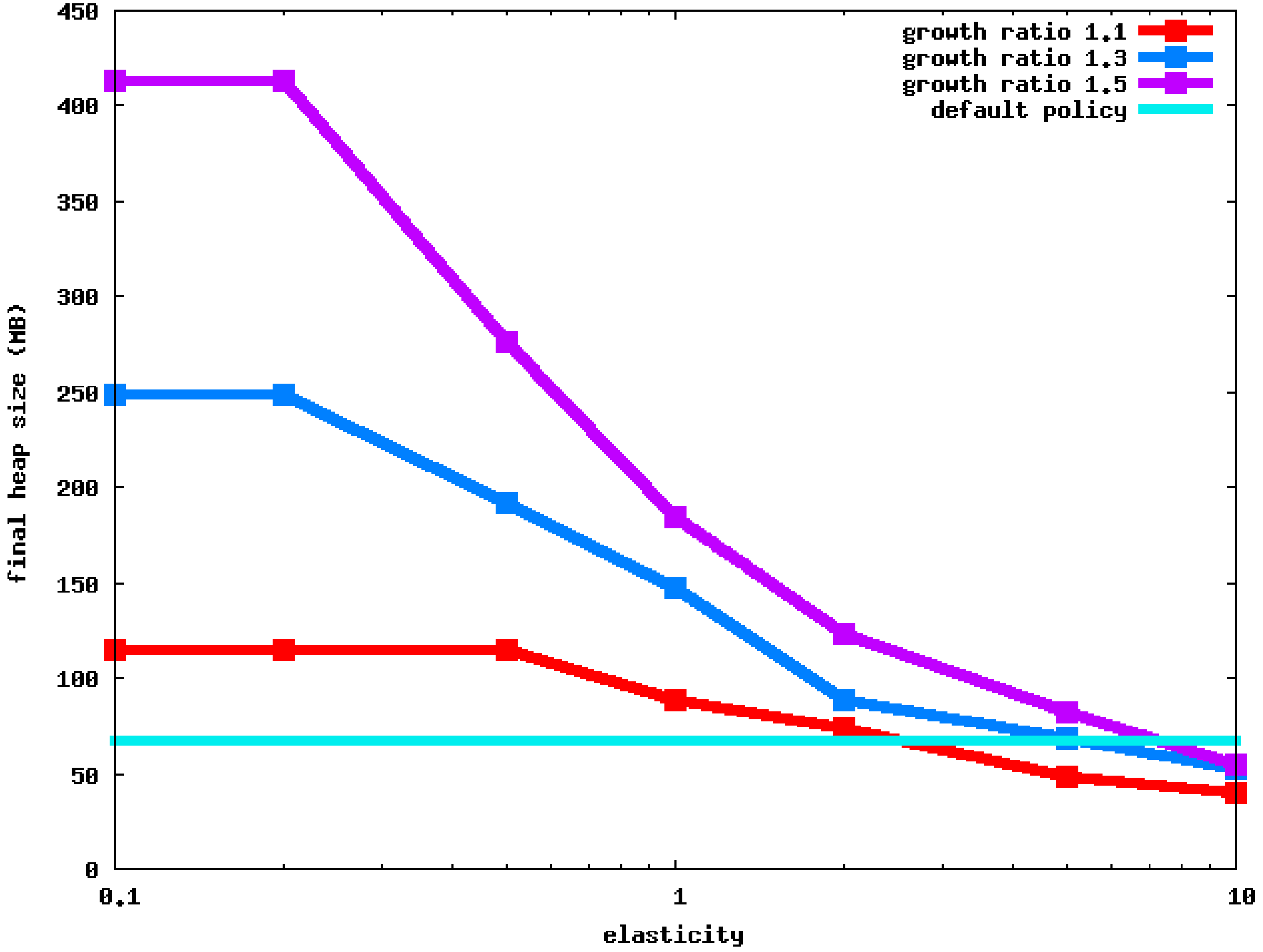
Use elasticity to control rate of heap growth

- at program start, user specifies a *target elasticity value*
- at each GC, JVM computes *currE*
- if *currE* > *targetE*, grow heap, otherwise maintain current size
  - large *targetE* – many GCs must occur at current size before heap grows
  - small *targetE* – few GCs at current size will trigger heap growth

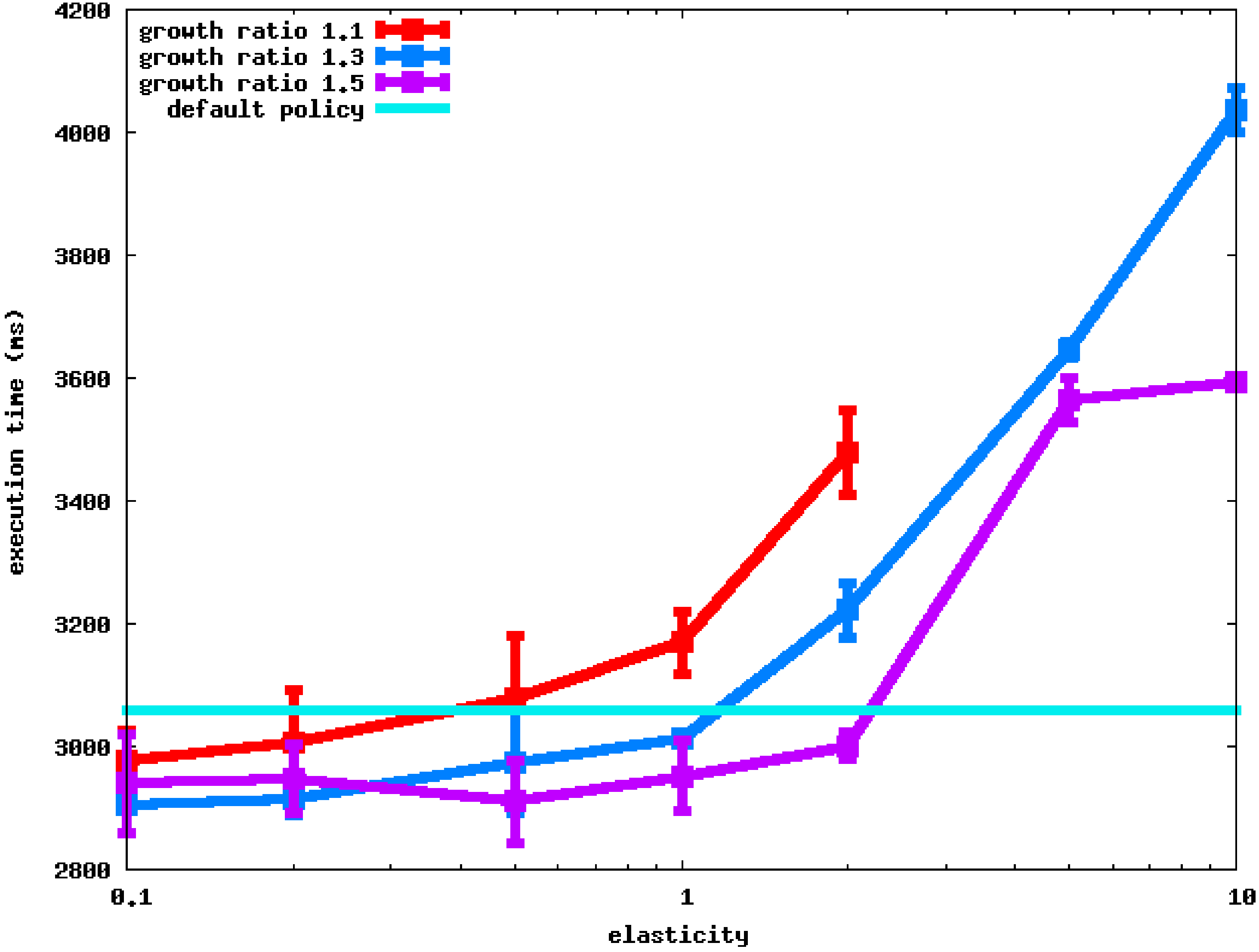
antlr


growth ratio 1.1
growth ratio 1.3
growth ratio 1.5
default policy
execution time (ms)
1350
1300
1250
1200
1150
1100
1050
0.1
1
10
elasticity

antlr

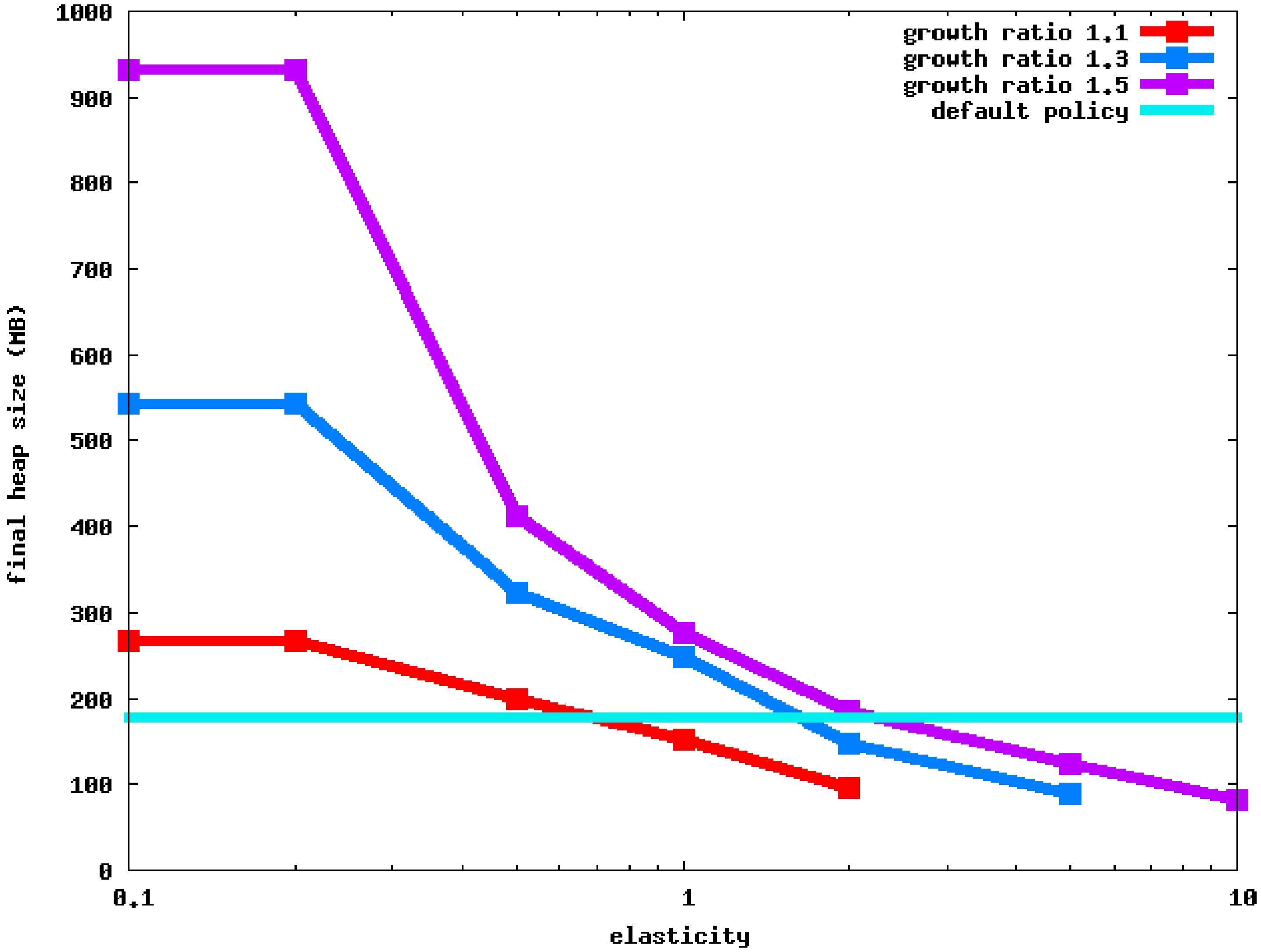

450
400
350
300
250
200
150
100
50
0
final heap size (MB)
0.1
1
10
elasticity
growth ratio 1.1
growth ratio 1.3
growth ratio 1.5
default policy

pmd



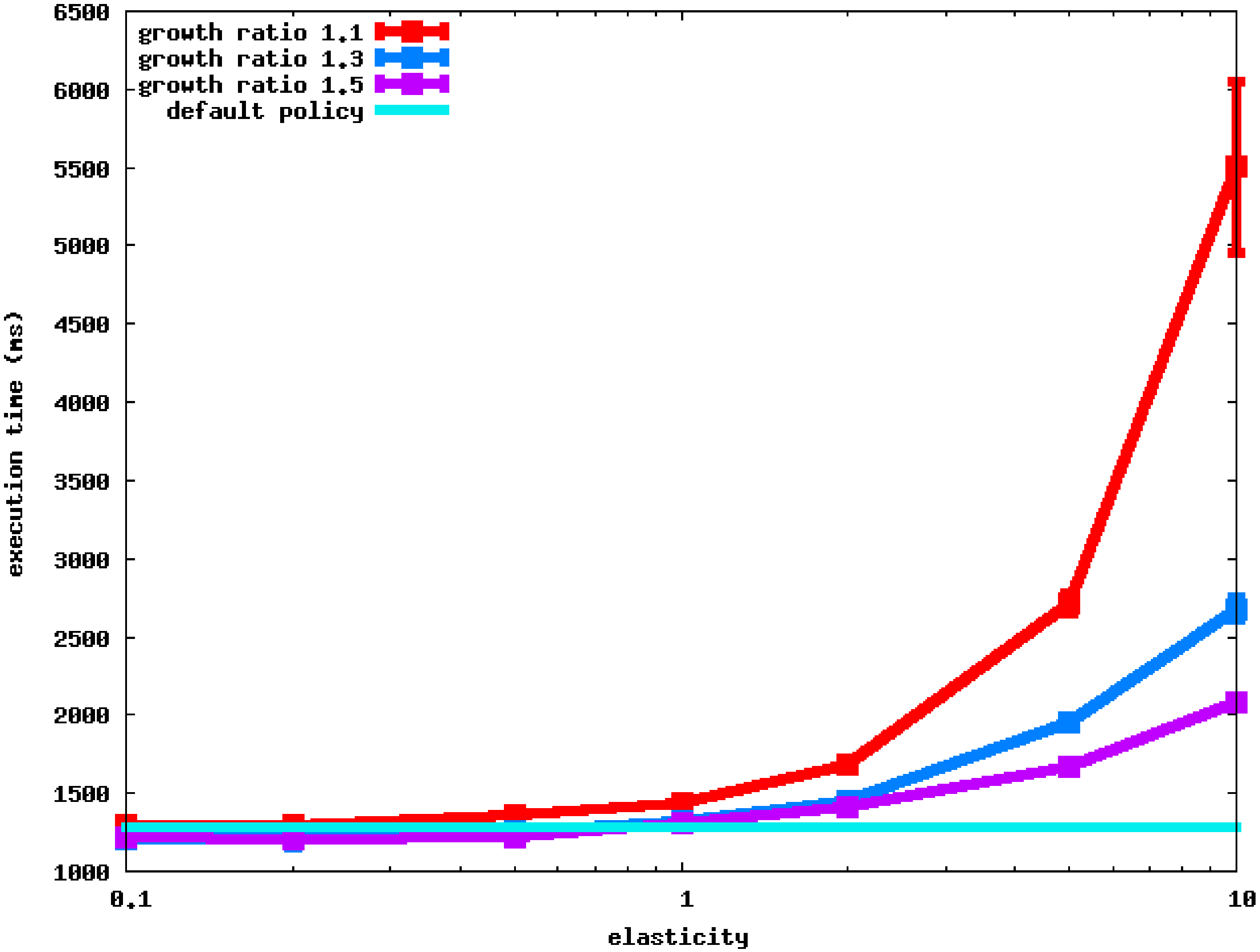
growth ratio 1.1
growth ratio 1.3
growth ratio 1.5
default policy
4200
4000
3800
3600
3400
3200
3000
2800
execution time (ms)
0.1
1
10
elasticity

pmd

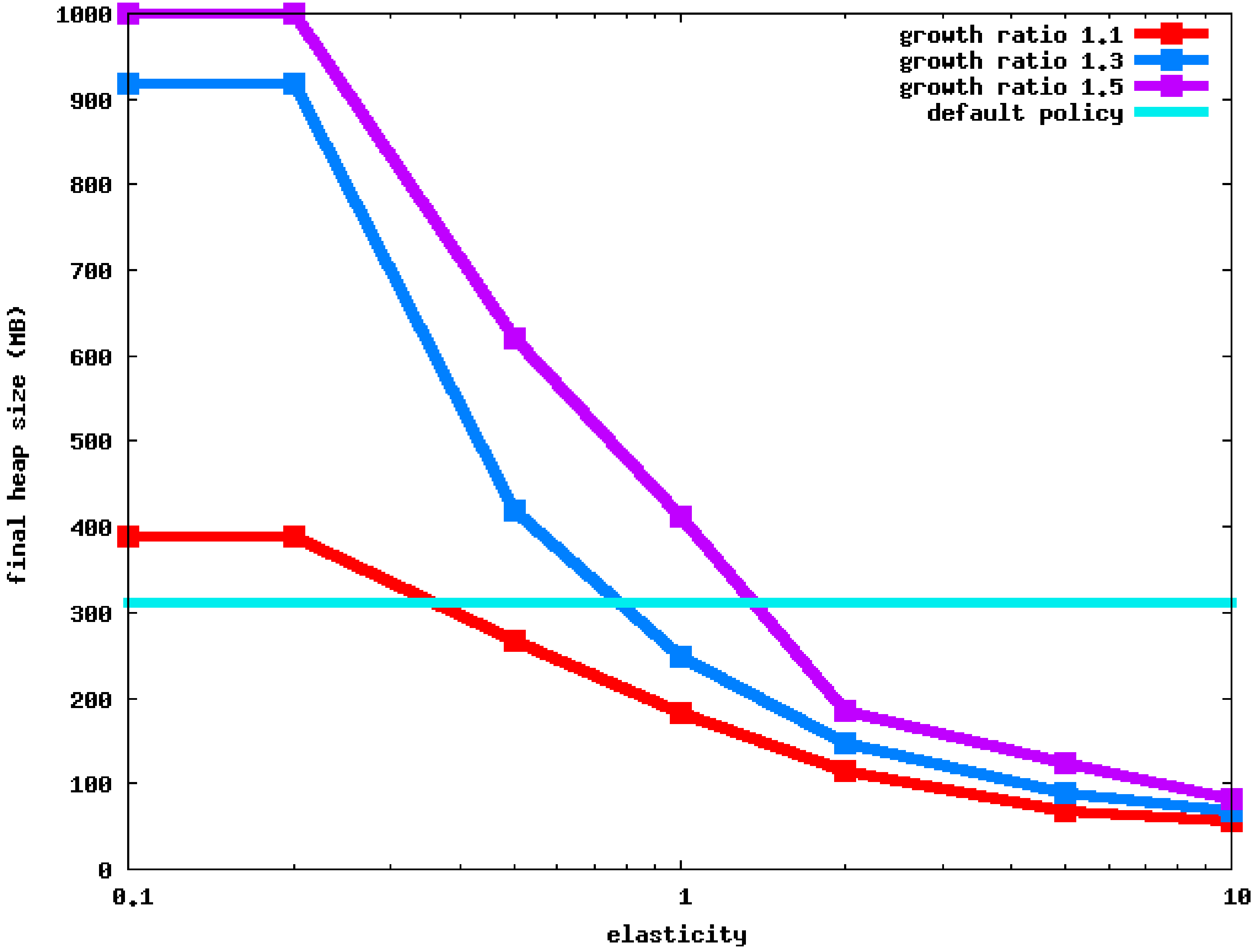


1000
900
800
700
600
500
400
300
200
100
0
final heap size (MB)
0.1
1
10
elasticity
growth ratio 1.1
growth ratio 1.3
growth ratio 1.5
default policy

xalan


growth ratio 1.1
growth ratio 1.3
growth ratio 1.5
default policy
execution time (ms)
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
0.1
1
10
elasticity

xalan


growth ratio 1.1
growth ratio 1.3
growth ratio 1.5
default policy
final heap size (MB)
elasticity
1000
900
800
700
600
500
400
300
200
100
0
0.1
1
10

Conclusions

- Interesting and closely-corresponding analogy between micro-economics and garbage collection
- Practical application – more systematic way to manage heap growth
- Further correspondences? Further analogies?