Free-Me: A Static Analysis for Automatic Individual Object Reclamation

Samuel Z. Guyer, Kathryn McKinley, Daniel Frampton

Presented by: Jason VanFickell

Thanks to Dimitris Prountzos for slides adapted from the original PLDI talk

Motivation

- Automatic memory reclamation (GC)
  - No need for explicit "free"
  - C
  -

- **Pr**
  -

Can we combine the software engineering advantage of garbage collection with the low-cost incremental reclamation of explicit memory management ?

Reclaim memory quickly (minimize memory footprint), with high overhead

  - Infrequent GCs:

Lower overhead, but lots of garbage in memory

Example

```
void parse(InputStream stream) {
  while (not_done) {
    String idName = stream.readToken();
    Identifier id = symbolTable.lookup(idName);
    if (id == null) {
      id = new Identifier(idName);
      symbolTable.add(idName, id);
    }
    computeOn(id);
  }
}
```

Read a token (new String)

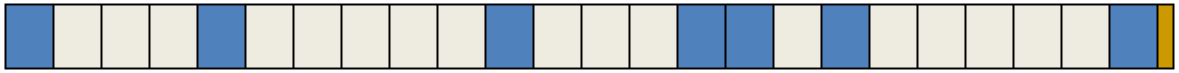
Look up in symbol table

If not there, create new identifier, add to symbol table

Compute on identifier

- ***Notice***: String `idName` is often garbage

Memory:



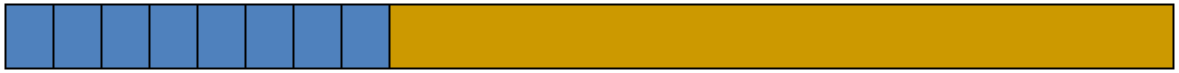
Explicit Reclamation as the solution

```
void parse(InputStream stream) {
  while (not_done) {
    String idName = stream.readToken();
    Identifier id = symbolTable.lookup(idName)
    if (id == null) {
      id = new Identifier(idName);
      symbolTable.add(idName, id);
    }
    else free(idName);
    computeOn(id);
}}
```

String idName is garbage, free immediately

- Garbage does not accumulate

Memory:



FreeMe as the solution

- Adds **`free()`** automatically
  - ***FreeMe*** compiler pass inserts calls to **`free()`**
  - Preserve software engineering benefits
- Can't determine lifetim
  - Works *with* the garbage
  - Implementation of **`free`**
- **Goal:**
  - Incremental, "eager" memory reclamation

➡ Results: reduce GC load, improve performance

Potential: 1.7X performance
`malloc/free` vs GC
in tight heaps

(Hertz & Berger, OOPSLA 2005)

FreeMe Analysis

- **Goal**:
  - Determine *when* an object becomes *unreachable*

Within a method,
for allocation site "`p = new A`"
where can we place a call to "`free(p)`"?

➡ Not a whole-program analysis*

I'll describe the interprocedural parts later

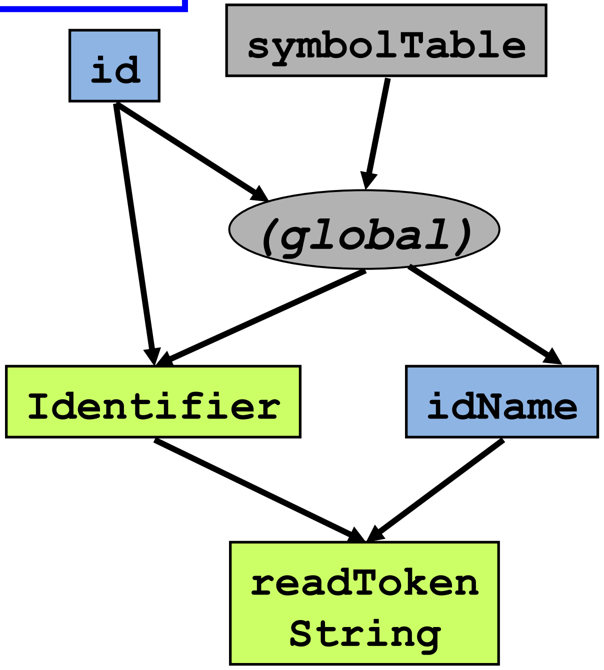
- **Idea**: pointer analysis + liveness
  - Pointer analysis for *reachability*
  - Liveness analysis for *when*

Pointer Analysis

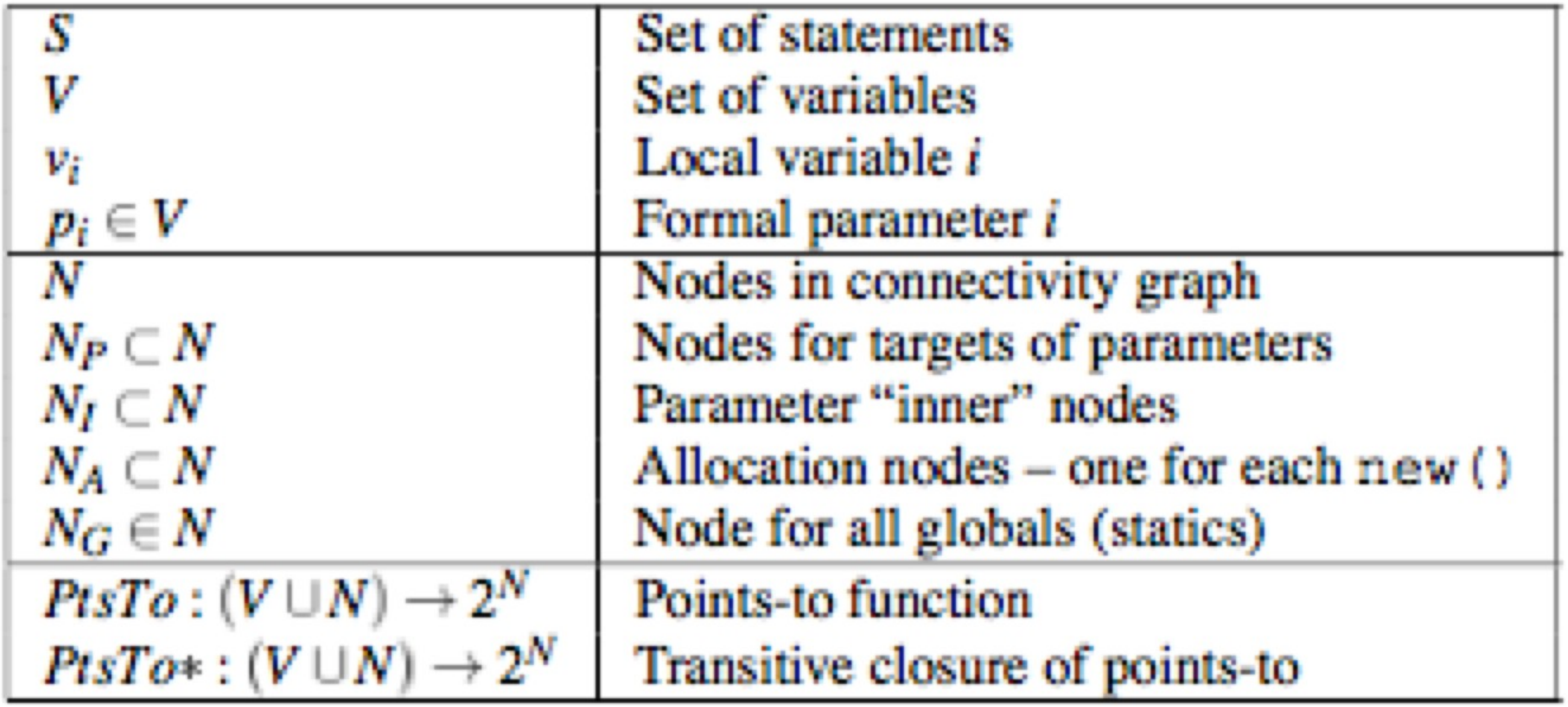
```
String idName = stream.readToken();
Identifier id = symbolTable.lookup(idName);
if (id == null) {
  id = new Identifier(idName);
  symbolTable.add(idName, id);
}
computeOn(id);
```

- Connectivity graph
  - *Variables* 🟦
  - *Allocation sites* 🟩
  - *Globals (statics)* ⬜
- Analysis algorithm

  Flow-insensitive, field-insensitive



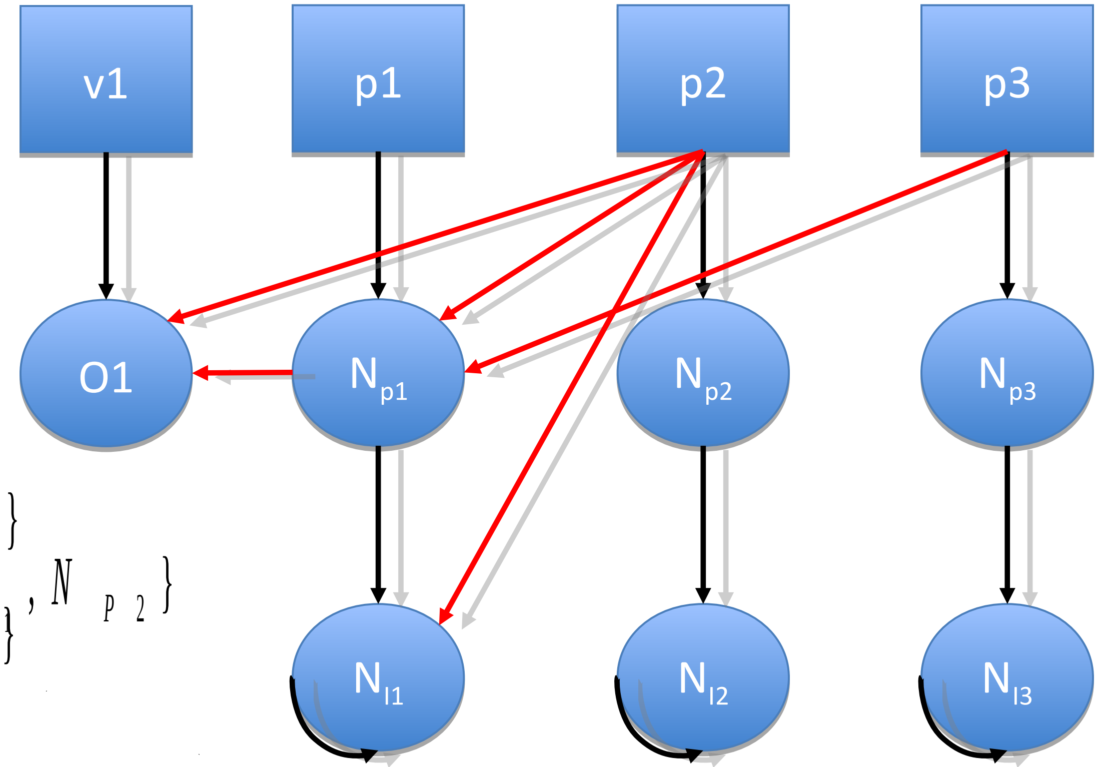
Pointer Analysis in more depth



S	Set of statements
V	Set of variables
v_i	Local variable i
$p_i \in V$	Formal parameter i
N	Nodes in connectivity graph
$N_P \subset N$	Nodes for targets of parameters
$N_I \subset N$	Parameter "inner" nodes
$N_A \subset N$	Allocation nodes – one for each `new()`
$N_G \in N$	Node for all globals (statics)
$PtsTo : (V \cup N) \rightarrow 2^N$	Points-to function
$PtsTo* : (V \cup N) \rightarrow 2^N$	Transitive closure of points-to

Calculating the Points-To relation

```
void function(A p1, A p2, A p3)
{
    v1 = new O
    p1.f = v1
    p2 = p1.f
    p3 = p1
}
```



$$\forall i, PtsTo(p_i) = \{N_{p_i}\}$$

$$PtsTo(p_1) = \{N_{p_1}, O_1\}$$

$$\forall i, PtsTo(N_{p_i}) = \{N_{l_i}\}$$

$$PtsTo(p_2) = \{N_{p_1}, O_1, N_{l_1}, N_{p_2}\}$$

$$\forall i, PtsTo(N_{l_i}) = \{N_{l_i}\}$$

$$PtsTo(p_3) = \{N_{p_1}, N_{p_3}\}$$

Interprocedural component

- Detection of ***factory*** methods

```
String idName = stream.readToken();
```

  - Return value is a new object
  - Can be freed by the caller

```
Hashtable.add:
```
$(0 \rightarrow 1)$

$(0 \rightarrow 2)$

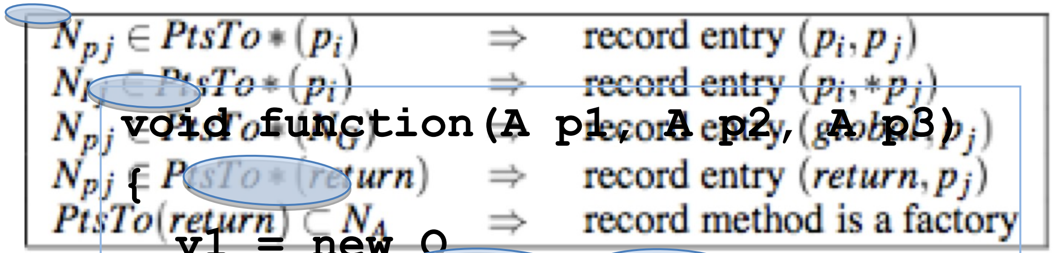
- Effects of methods called

```
symbolTable.add(idName, id);
```

  - Describes how parameters are connected

- ***Compilation strategy***:
  - Summaries pre-computed for all methods
  - Free-me only applied to hot methods

Generating summaries in more depth



$N_{pj} \in PtsTo*(p_i)$	$\Rightarrow$	record entry (p_i, p_j)
$N_{pj} \in PtsTo*(p_i)$	$\Rightarrow$	record entry $(p_i, *p_j)$
$N_{pj} \in PtsTo*(N_G)$	$\Rightarrow$	record entry $(global, p_j)$
$N_{pj} \in PtsTo*(return)$	$\Rightarrow$	record entry $(return, p_j)$
$PtsTo(return) \subseteq N_A$	$\Rightarrow$	record method is a factory

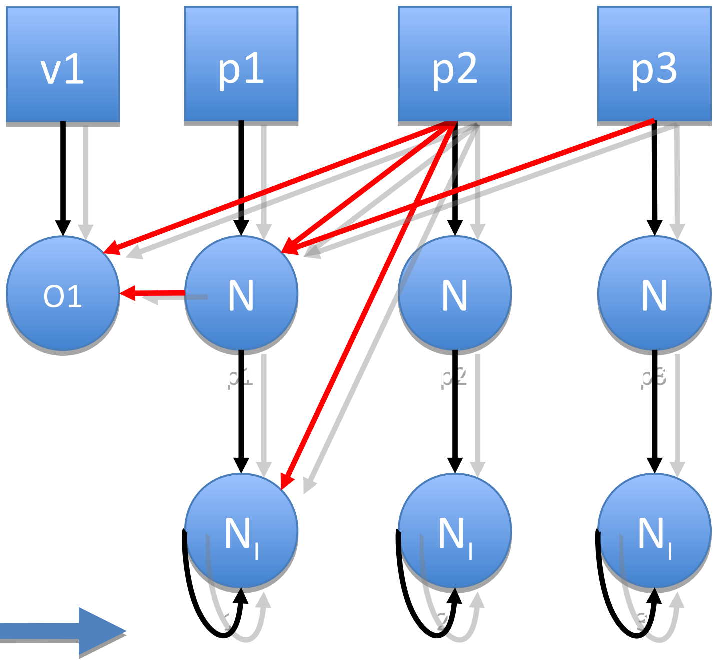
```
void function(A p1, A p2, A p3)
{
    v1 = new O
    p1.f = v1
    p2 = p1.f
    p3 = p1
}
```

getfield is needed because a single pointer link in summary may represent multiple pointers in the callee

$$PtsTo(p_1) = \{N_{p_1}, O_1\}$$

$$PtsTo(p_2) = \{N_{p_1}, O_1, N_{I_1}, N_{p_2}\}$$

$$PtsTo(p_3) = \{N_{p_1}, N_{p_3}\}$$

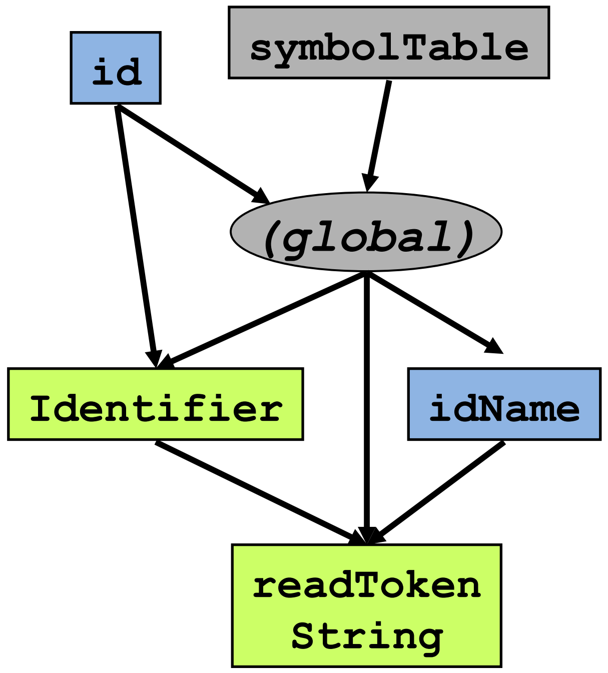


$(p_2, p_1), (p_2, *p_1)$

(p_3, p_1)

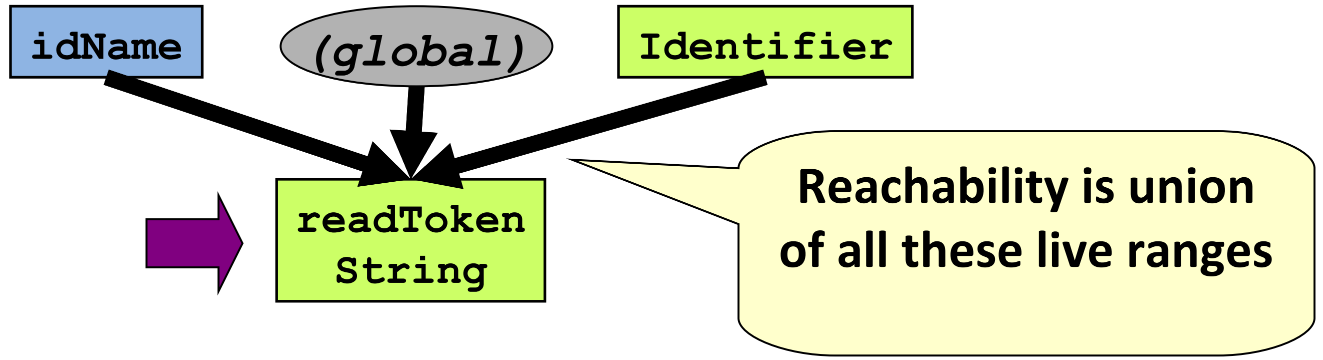
The need for liveness analysis

- ***When*** objects become unreachable, not just whether or not they escape



Adding Liveness

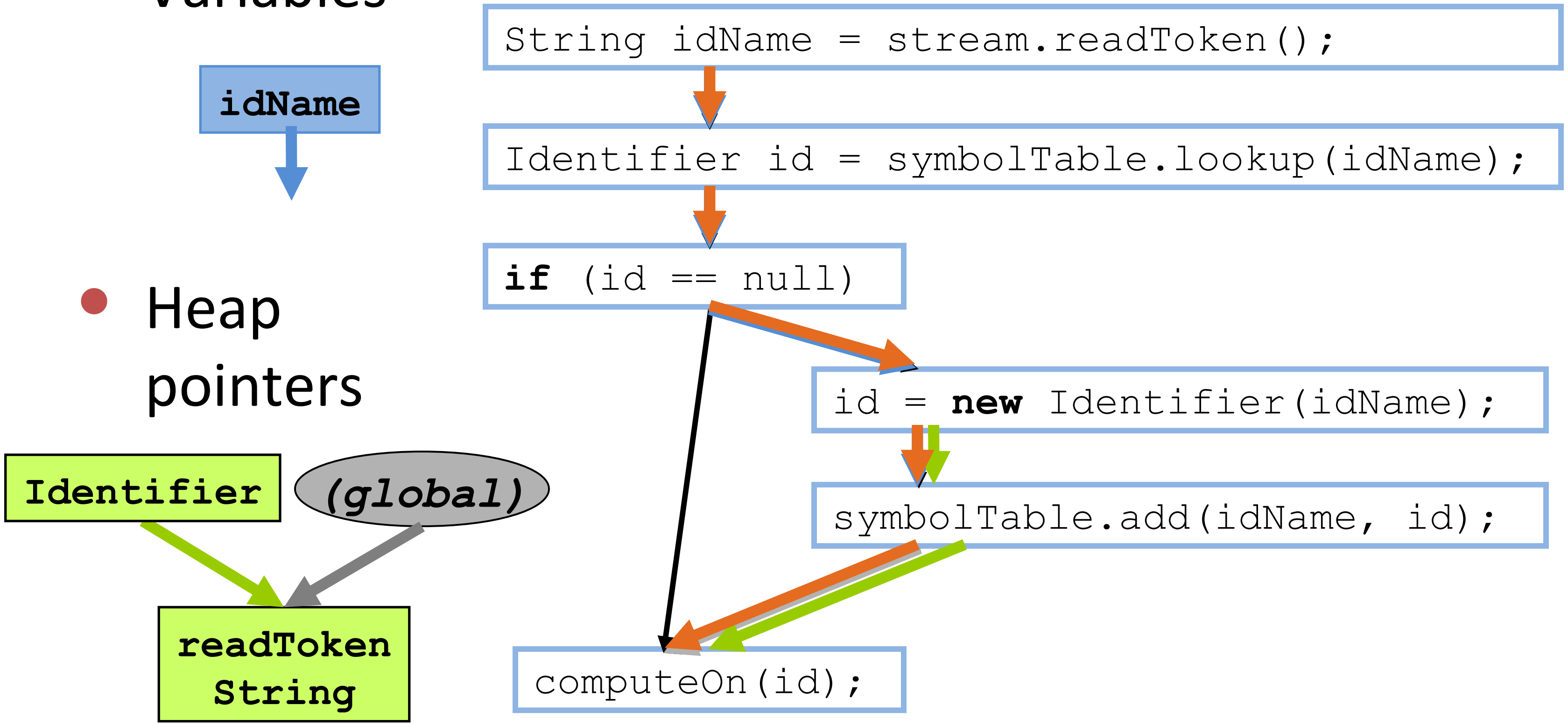
- ***Key*** : An object is reachable only when all incoming pointers are live



🟦 →	From a variable:	Live range of the variable
⬜ →	From a global:	Live from the pointer store onward
🟩 →	From other object:	Live from the pointer store until source object becomes unreachable

Liveness Analysis

- Computed as sets of edges
  - Variables
  - Heap pointers



```
String idName = stream.readToken();
Identifier id = symbolTable.lookup(idName);
if (id == null)
id = new Identifier(idName);
symbolTable.add(idName, id);
computeOn(id);
```

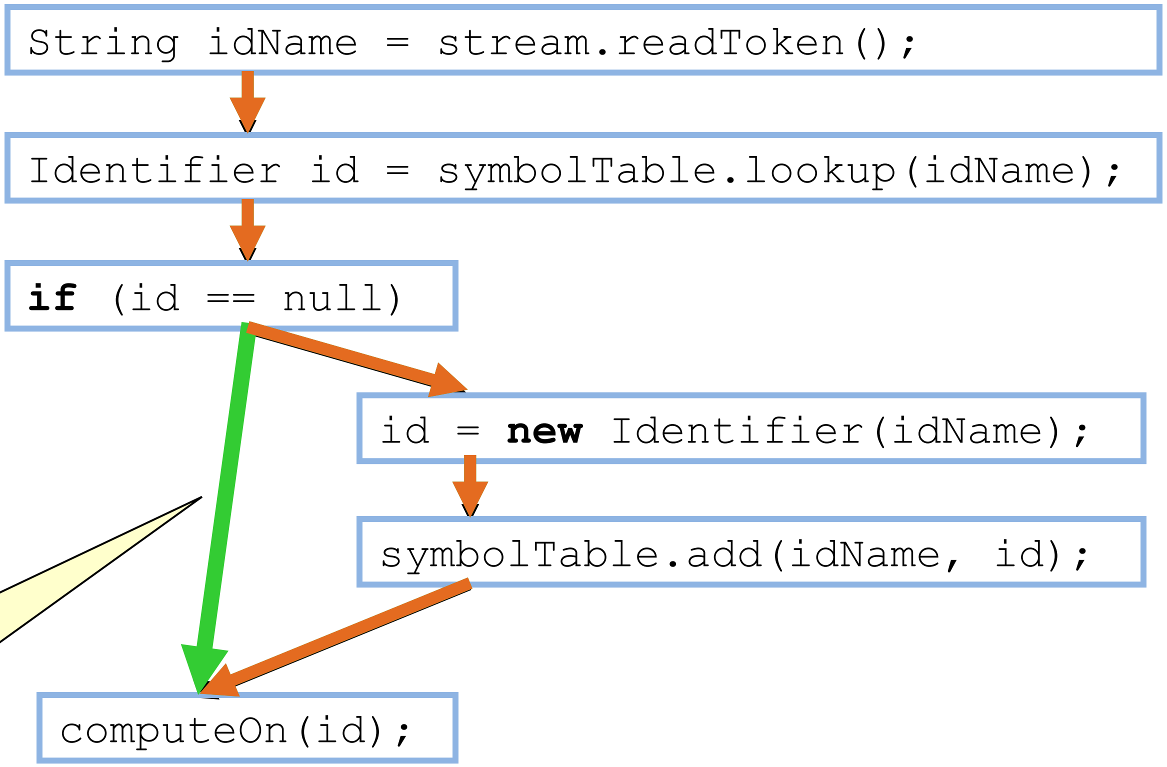
Where can we free it?

- Where object exists

`readToken String`

-minus-

- Where reachable

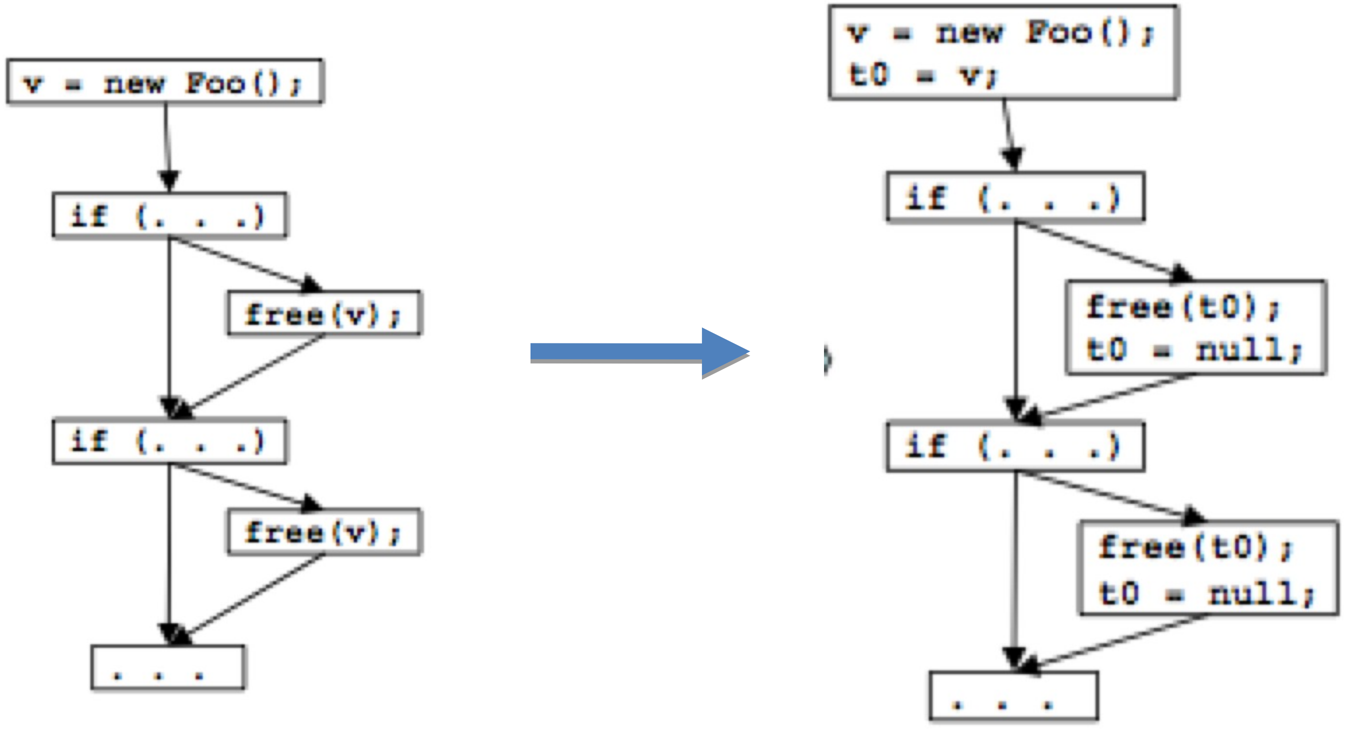


```
String idName = stream.readToken();
Identifier id = symbolTable.lookup(idName);
if (id == null)
    id = new Identifier(idName);
    symbolTable.add(idName, id);
computeOn(id);
```

Compiler inserts call to `free(idName)`

Free placement issues

- Select earliest point A,eliminate all B: A dom B
- Deal with double free's

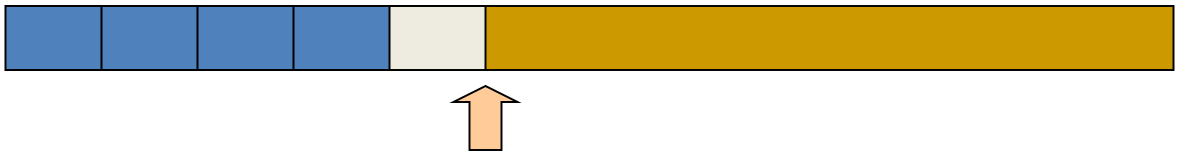


Runtime support for FreeMe

- Run-time: depends on collector
  - Mark/sweep

    Free-list: **`free()`** operation
  - Generational mark/sweep

    Unbump: move nursery "bump pointer" backward (LIFO frees)

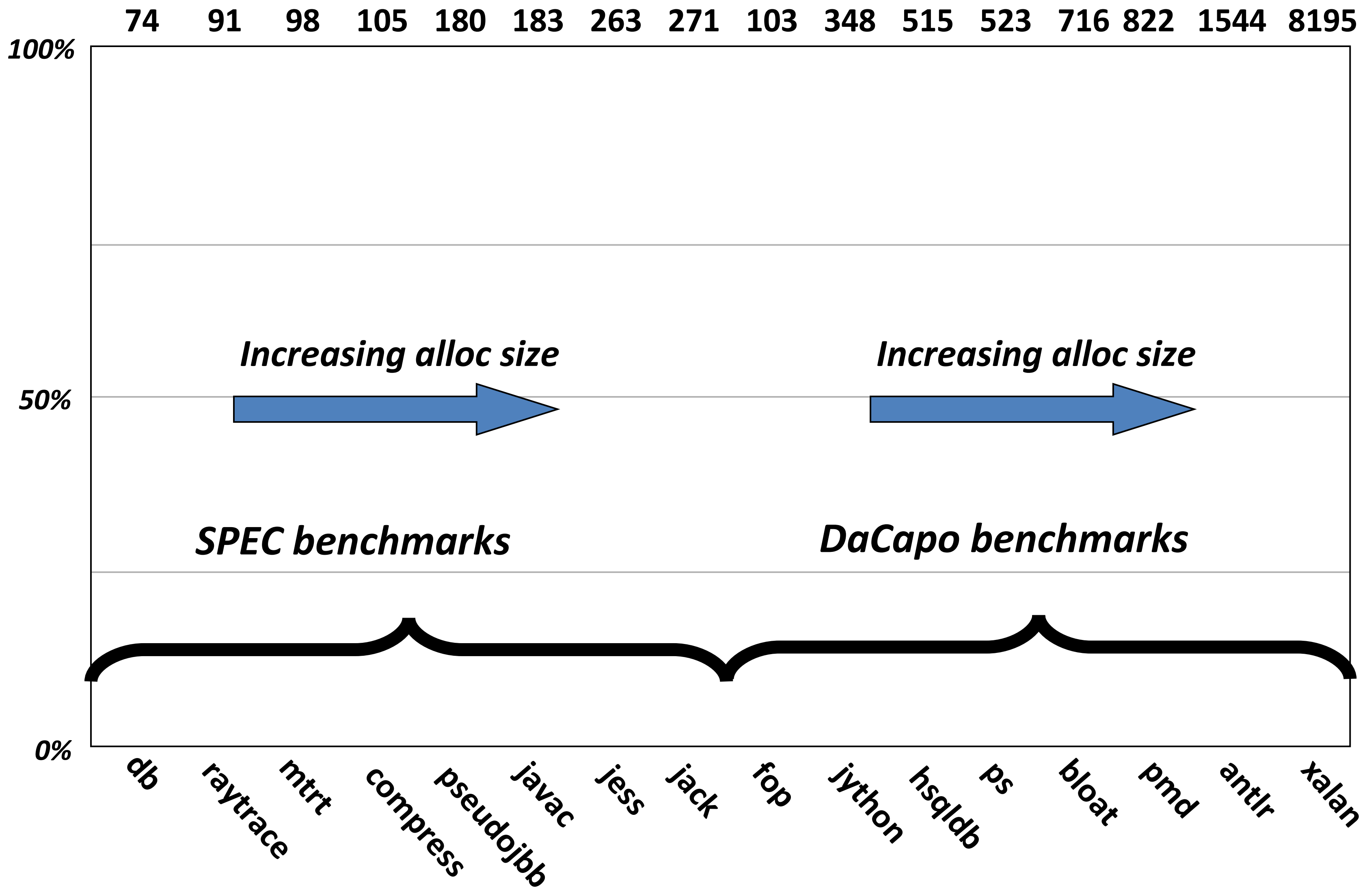


Unreserve: reduce copy reserve
  - Very low overhead
  - Run longer without collecting

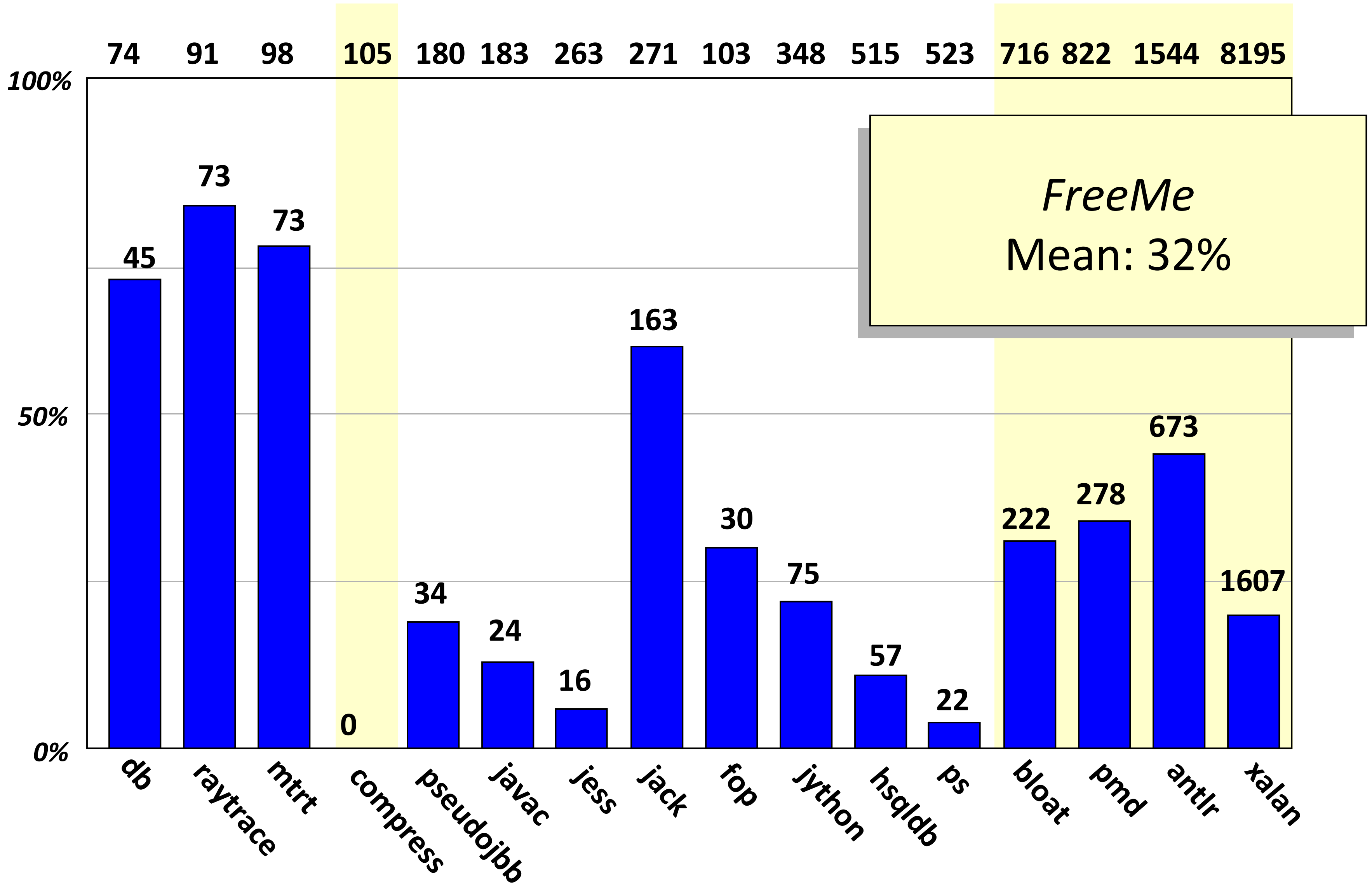
- Size to free defined statically/dynamically (query object)

Experimental Evaluation

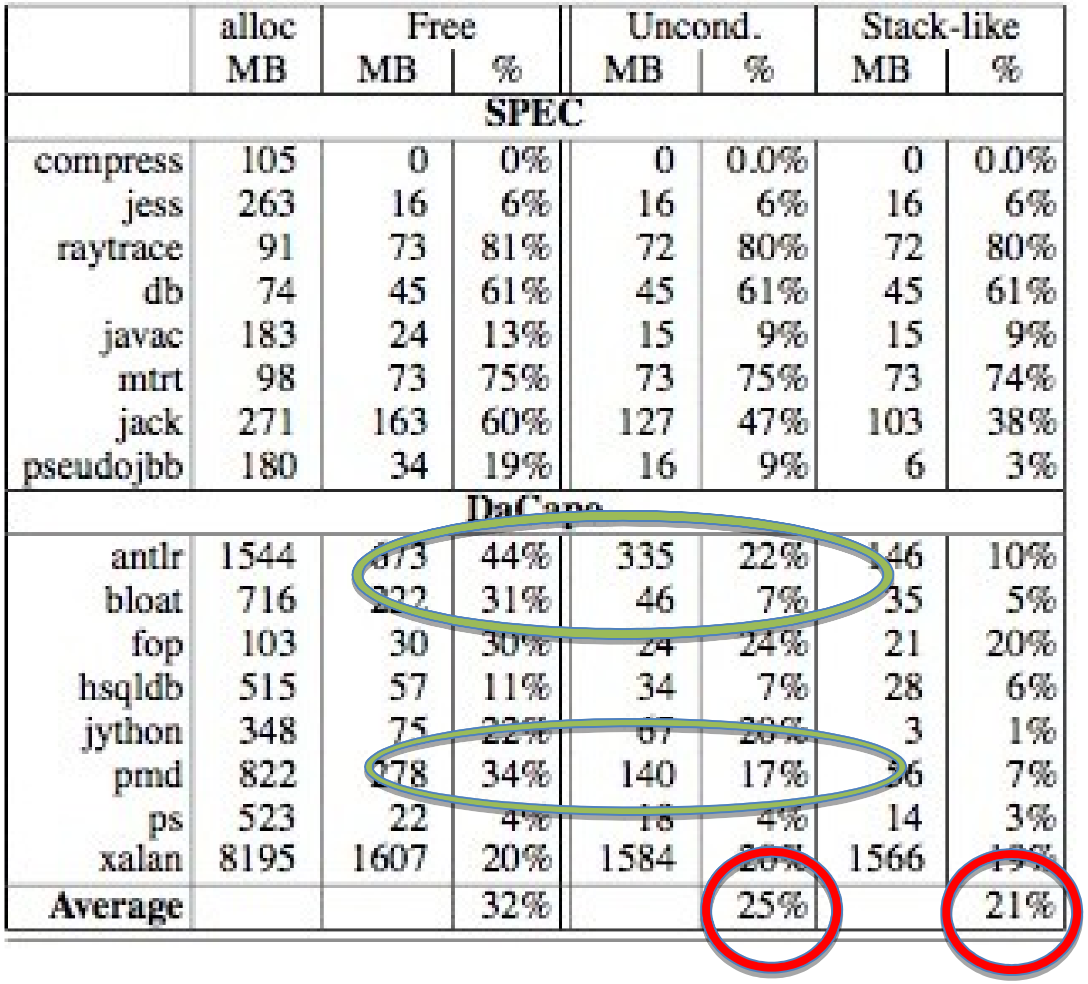
Volume freed – in MB



Volume freed – in MB



Comparing FreeMe & other approaches



	alloc	Free		Uncond.		Stack-like	
	MB	MB	%	MB	%	MB	%
SPEC							
compress	105	0	0%	0	0.0%	0	0.0%
jess	263	16	6%	16	6%	16	6%
raytrace	91	73	81%	72	80%	72	80%
db	74	45	61%	45	61%	45	61%
javac	183	24	13%	15	9%	15	9%
mtrt	98	73	75%	73	75%	73	74%
jack	271	163	60%	127	47%	103	38%
pseudojbb	180	34	19%	16	9%	6	3%
DaCapo							
antlr	1544	673	44%	335	22%	146	10%
bloat	716	222	31%	46	7%	35	5%
fop	103	30	30%	24	24%	21	20%
hsqldb	515	57	11%	34	7%	28	6%
jython	348	75	22%	67	20%	3	1%
pmd	822	278	34%	140	17%	56	7%
ps	523	22	4%	18	4%	14	3%
xalan	8195	1607	20%	1584	20%	1566	19%
Average			32%		25%		21%

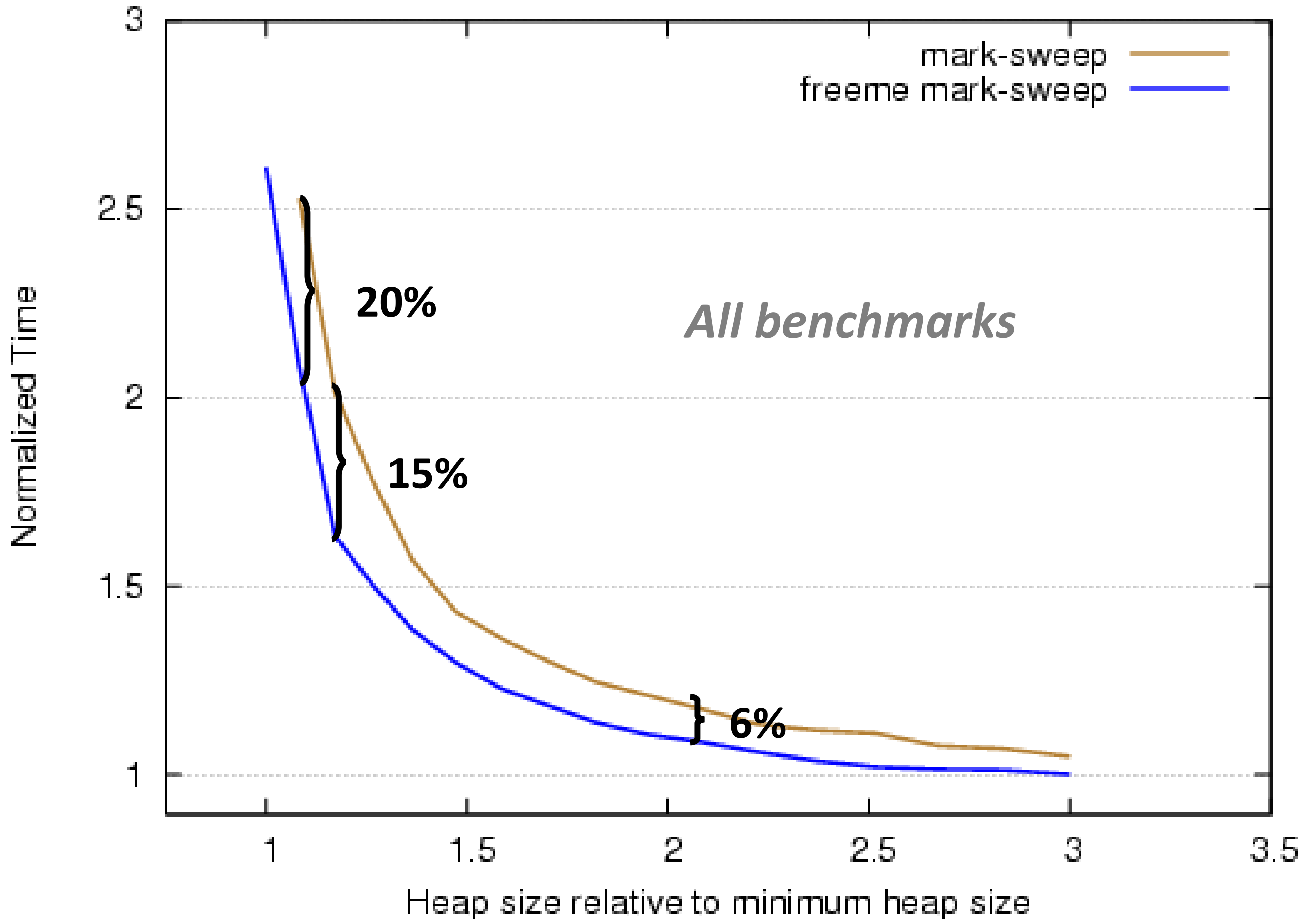
<u>Stack-like</u>

- free() allocations of same method
- Restrict free instrumentation to end of method
- No factory methods
- No conditional freeing

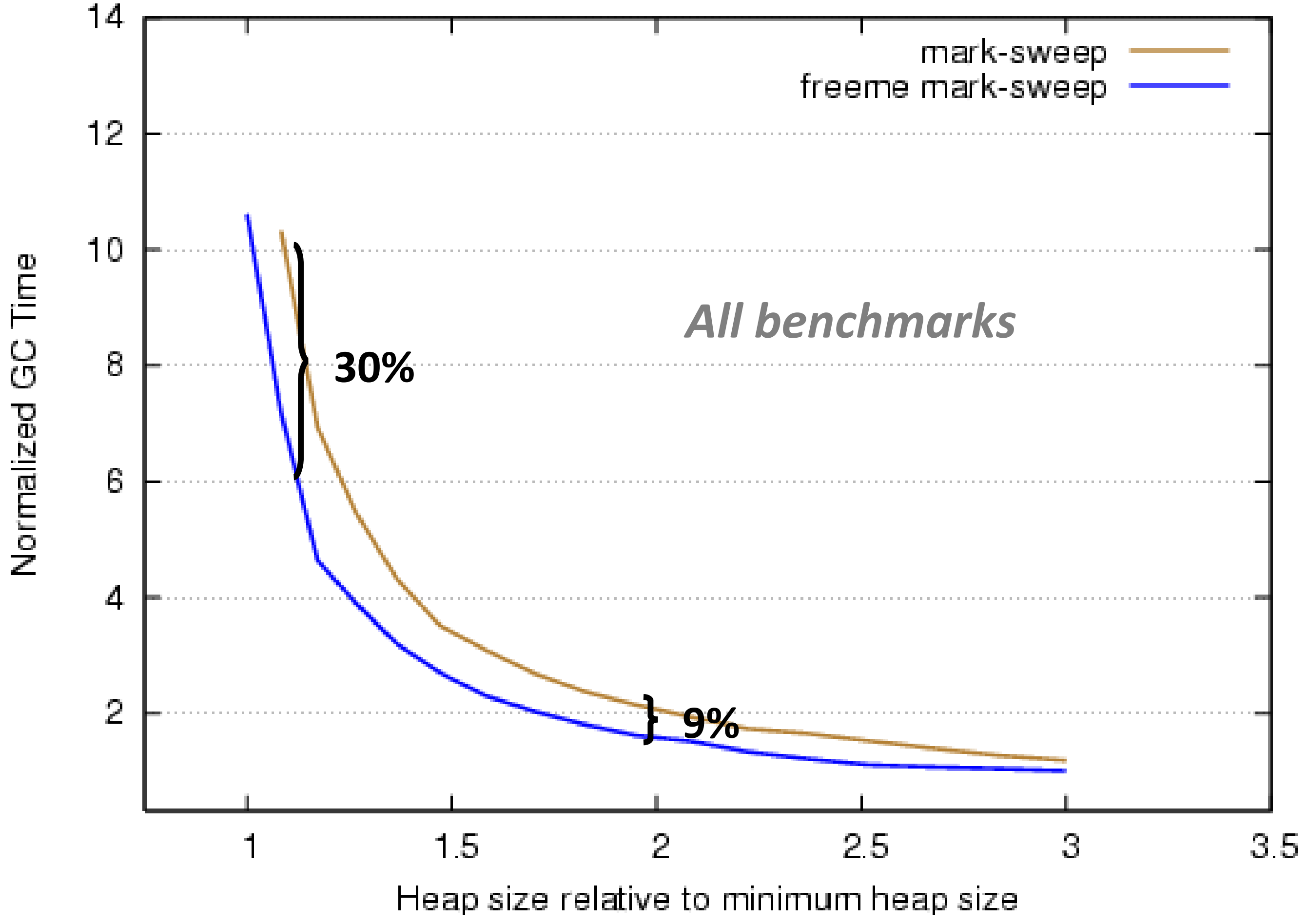
<u>Uncond</u>

- Prove objects dead on all paths
- Influence of free on some paths

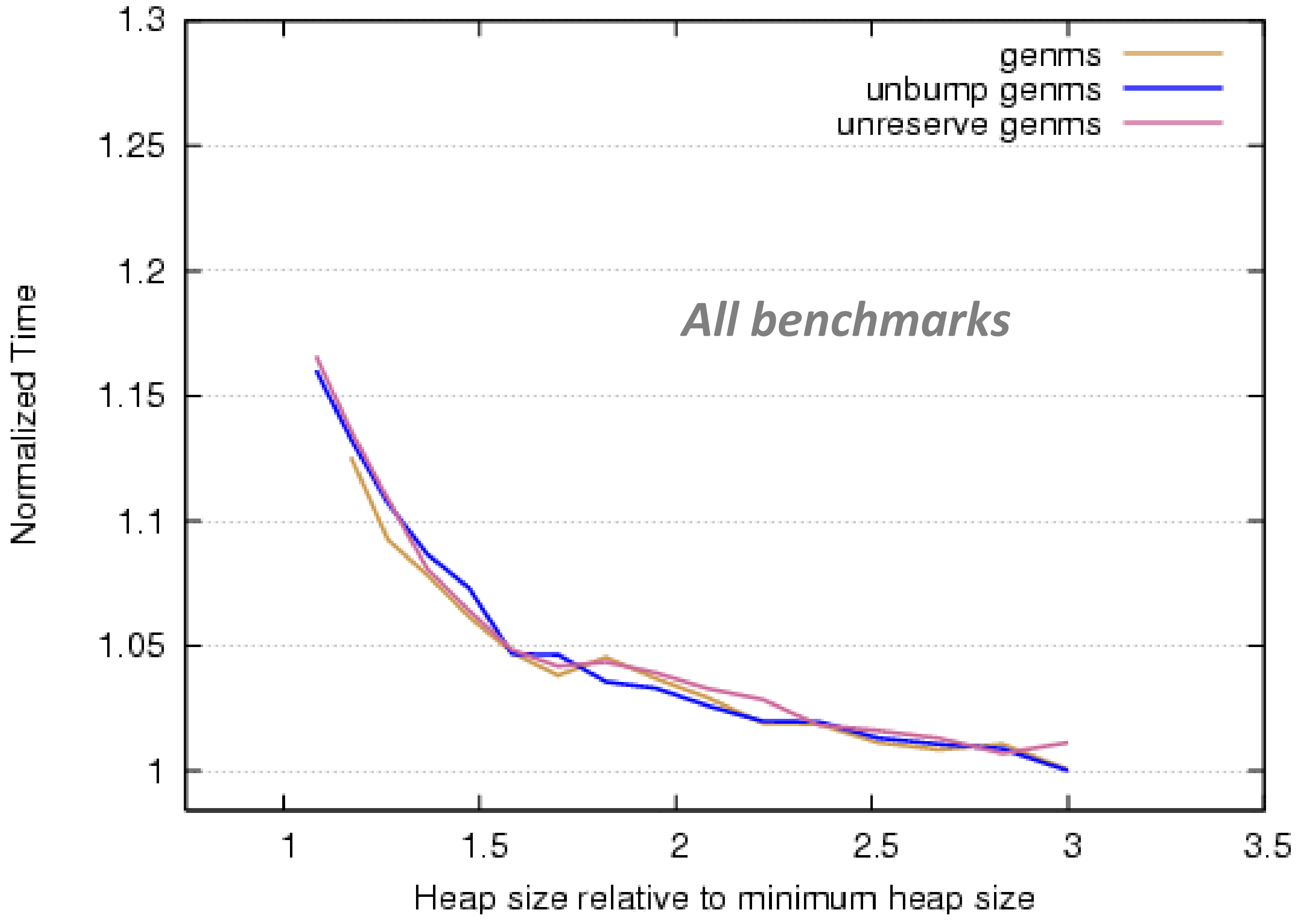
Mark/sweep – time



Mark/sweep – GC time

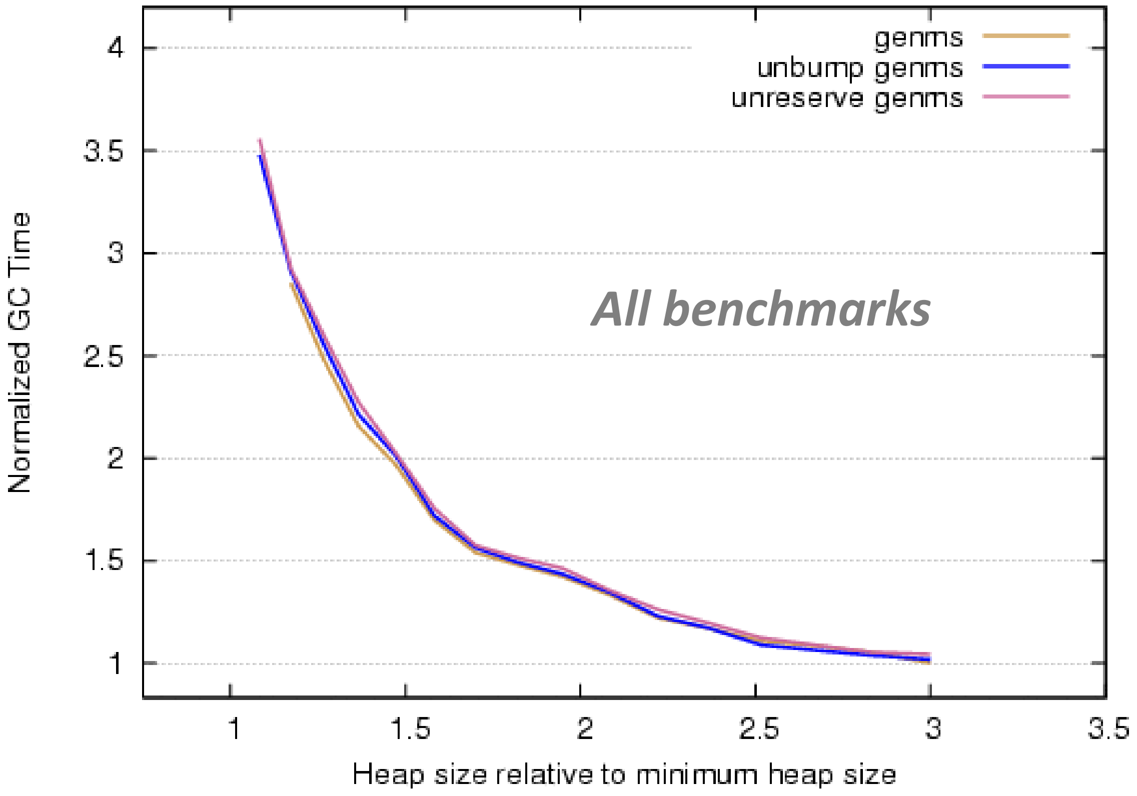


GenMS – time



Brings into question all techniques that target short-lived objects

GenMS – GC time



Why doesn't this help?

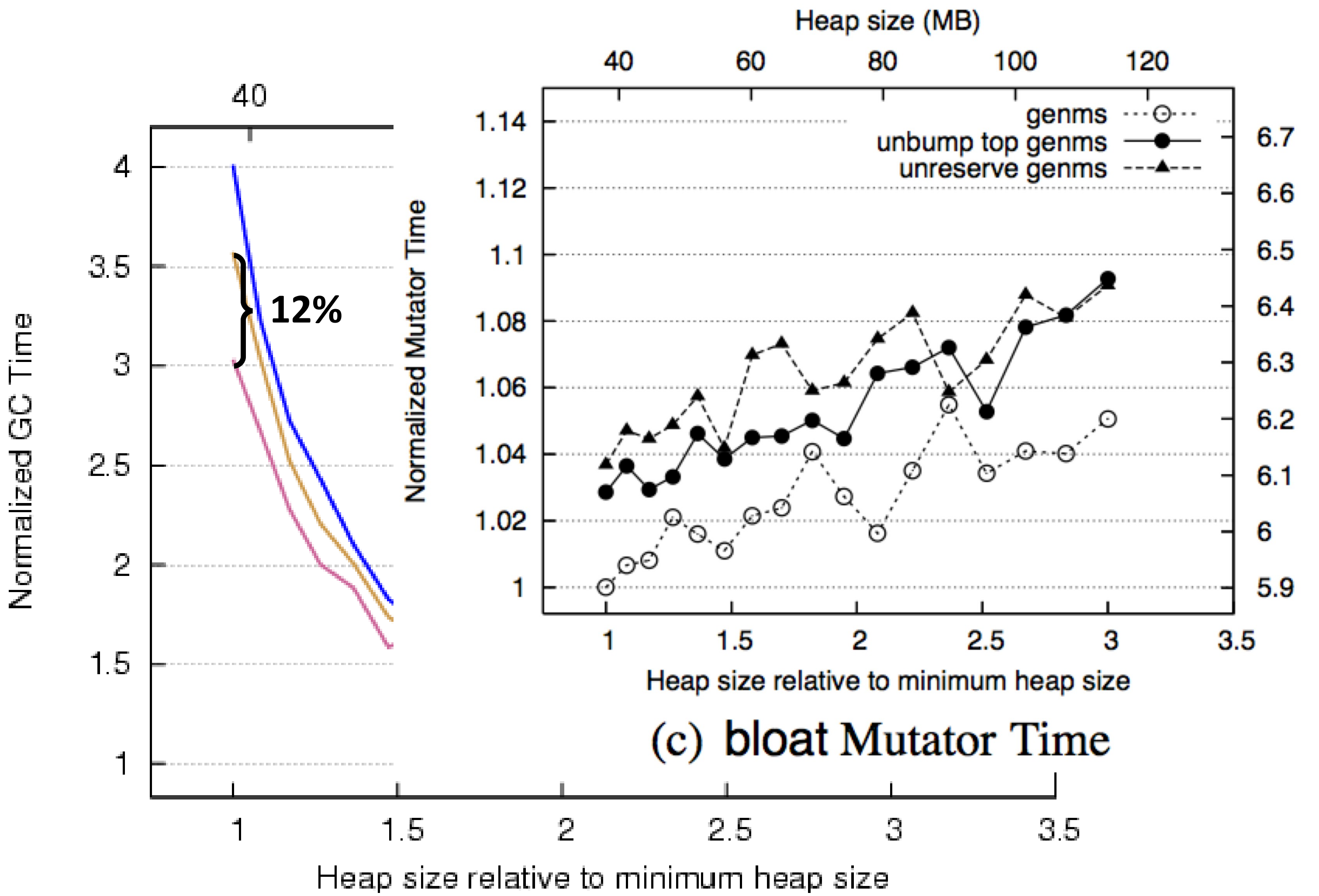
Note: the number of GCs is greatly reduced

FreeMe mostly finds short-lived objects

Nursery reclaims dead objects for ***free***

(cost ~ survivors)

Bloat – GC time



Conclusions

- **FreeMe** analysis
  - Finds many objects to free: often 30% - 60%
  - Most are short-lived objects
- GC + explicit **`free()`**
  - Advantage over stack/region allocation: no need to make decision at allocation time
- Generational collectors
  - Nursery works very well
- Mark-sweep collectors
  - 50% to 200% speedup
  - Works better as memory gets tighter

Embedded applications:

Compile-ahead

Memory constrained

Non-moving collectors

Discussion

- Is compile-time memory management inherently incompatible with generational copying collection?
- Is the amount of memory freed significant?
- Could static analysis allow mark-sweep collectors to compete with generational collectors?