UMassAmherst

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management

Matthew Hertz
Canisius College

Emery Berger
University of Massachusetts Amherst

Explicit Memory Management

- `malloc` / `new`
  - allocates space for an object
- `free` / `delete`
  - returns memory to system

- Simple, but tricky to get right
  - Forget to `free` ⇨ memory leak
  - `free` too soon ⇨ "dangling pointer"

Dangling Pointers

```
Node x = new Node ("happy");
Node ptr = x;
delete x;          // But I'm not dead yet!
Node y = new Node ("sad");
cout << ptr->data << endl;     // sad ☹
```

- Insidious, hard-to-track down bugs

Solution: Garbage Collection

- No need to call `free`
- Garbage collector periodically scans objects on heap
- Reclaims *unreachable* objects
  - Won't reclaim objects until it can prove object will not be used again

No More Dangling Pointers

```
Node x = new Node ("happy");
Node ptr = x;
// x still live (reachable through ptr)
Node y = new Node ("sad");
cout << ptr->data << endl; // happy! ☺
```

So why not use GC
all the time?

Conventional Wisdom

- "GC worse than **`malloc`**, because…"
  - Extra processing (*collection*)
  - Poor cache performance (*ibid*)
  - Bad page locality (*ibid*)
  - Increased footprint (*delayed reclamation*)

Conventional Wisdom

- "GC improves performance, by…"
  - Quicker allocation
    (*fast path inlining & bump pointer alloc.*)
  - Better cache performance
    (*object reordering*)
  - Improved page locality
    (*heap compaction*)

Outline

- Motivation
- Quantifying GC performance
  - A hard problem
- Oracular memory management
  - Selecting & generating the Oracles
- Experimental methodology
- Results

Quantifying GC Performance

- Apples-to-apples comparison
  - Examine unaltered applications
  - Runs differ only in memory manager
- Examine impact on **time** & **space**

Quantifying GC Performance

- Evaluate state-of-art algorithms
  - Garbage collectors
    - Generational collectors
    - Copying collectors
      - Standard for Java, not compatible with C/C++
  - Explicit memory managers
    - Fast, single-threaded allocators

Comparing Memory Managers

```
Node v = malloc(sizeof(Node));
v->data= malloc(sizeof(NodeData));
memcpy(v->data, old->data,
           sizeof(NodeData));
free(old->data);
v->next = old->next;
v->next->prev = v;
v->prev = old->prev;
v->prev->next = v;
free(old);
```

BDW Collector

Using GC in C/C++ is easy:

Comparing Memory Managers

```
Node v = malloc(sizeof(Node));
v->data= malloc(sizeof(NodeData));
memcpy(v->data, old->data,
           sizeof(NodeData));
free(old->data);
v->next = old->next;
v->next->prev = v;
v->prev = old->prev;
v->prev->next = v;
free(old);
```

BDW Collector

…ignore calls to `free`.

Comparing Memory Managers

```
Node node = new Node();
node.data = new NodeData();
useNode(node);
node = null;
...
node = new Node();
...
node.data = new NodeData();
...
```

Lea Allocator

Adding `malloc/free` to Java: not easy…

Comparing Memory Managers

```
Node node = new Node();
node.data = new NodeData();
useNode(node);
node = null;
...
node = new Node();
...
node.data = new NodeData();
...
```

free(node)?

free(node.data)?

Lea Allocator

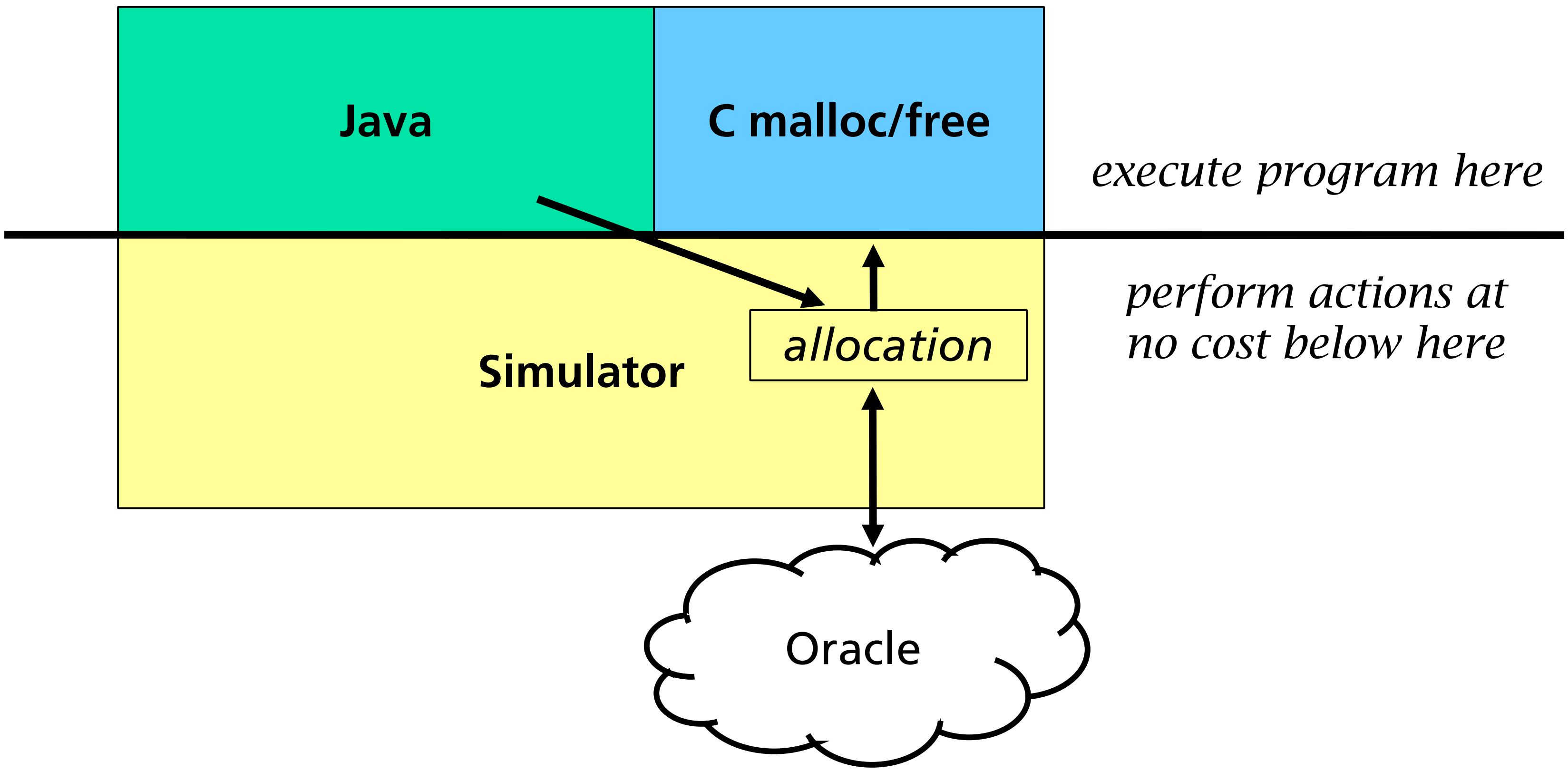
... where should **`free`** be inserted?

Inserting Free Calls

- Do not know where programmer would call **`free`**
  - Hints provided from **`null`**-ing pointers
  - Restructure code to avoid memory leaks?
    - Tests programming skills, not memory manager

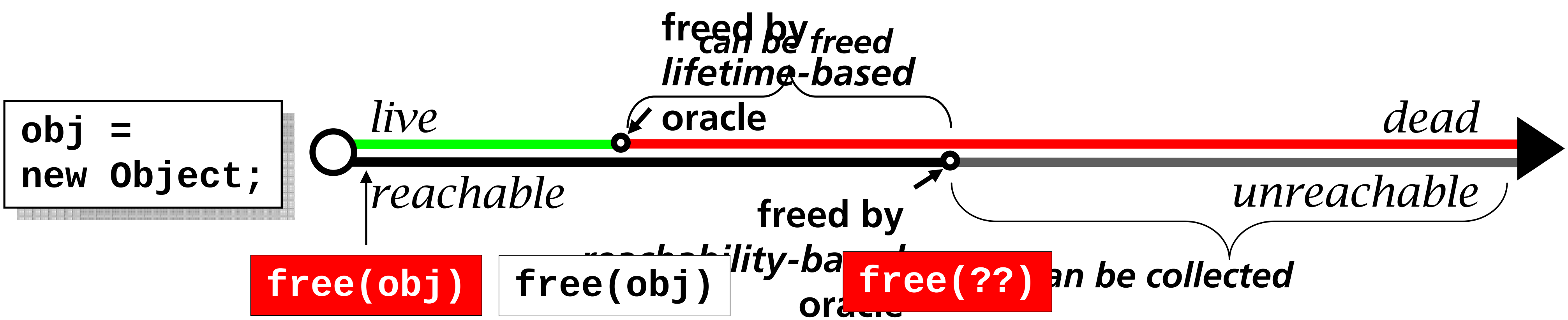
- Want *unaltered* applications

Oracular Memory Manager



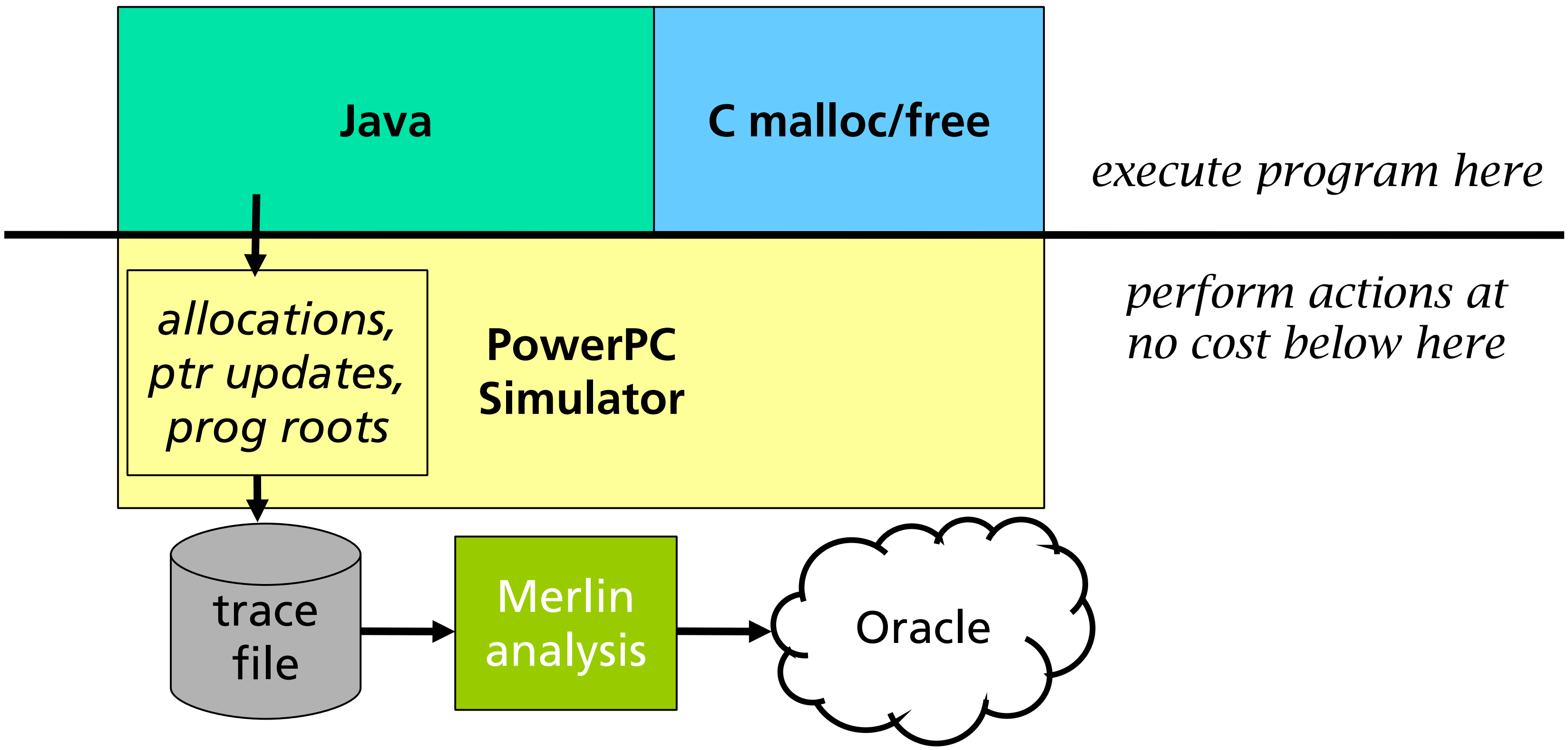
- Consult oracle to place **`free`** calls
  - Oracle does not disrupt hardware state
  - Simulator inserts **`free`**…

Object Lifetime & Oracle Placement



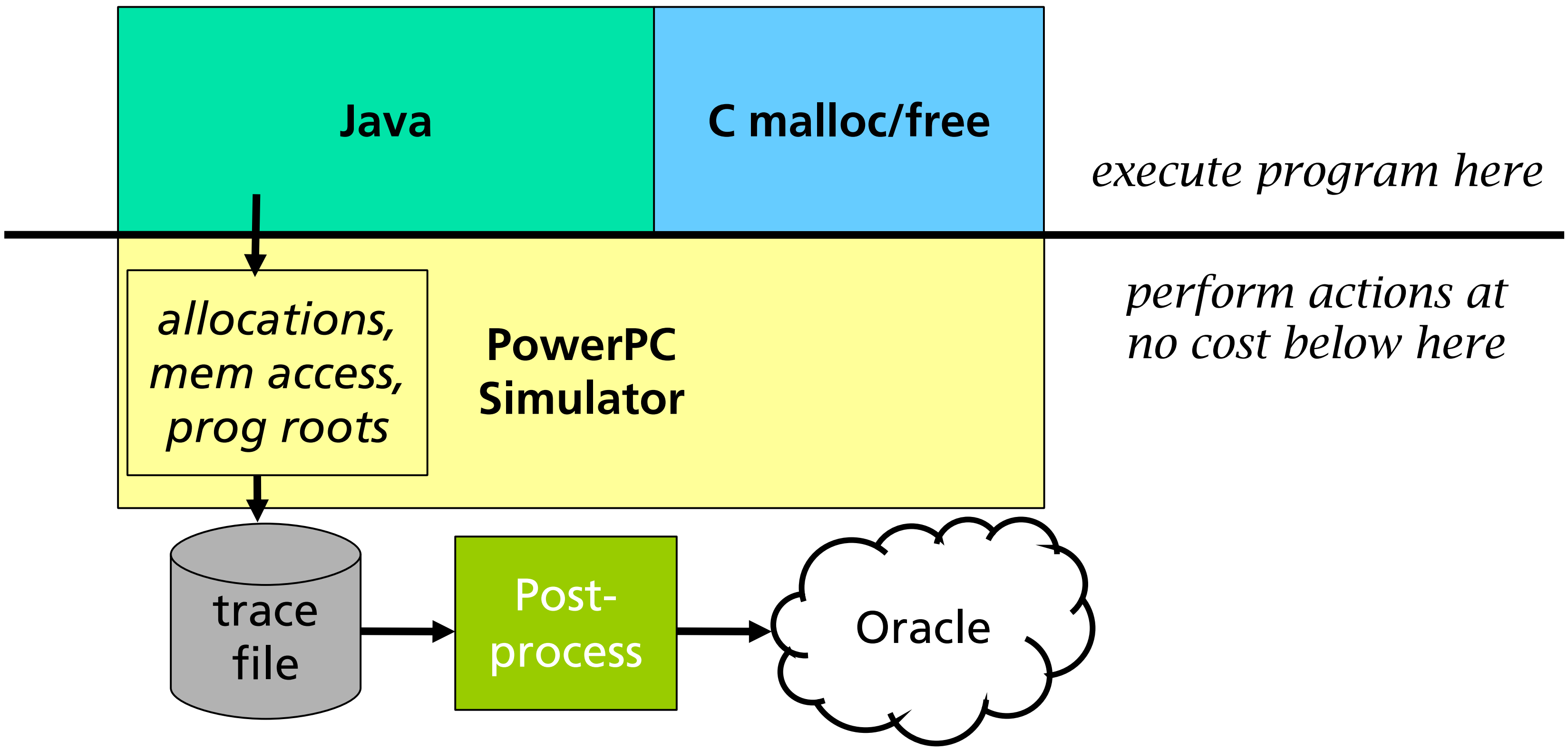
- Oracles bracket placement of `frees`
  - **Lifetime-based**: most aggressive
  - **Reachability-based**: most conservative

Reachability Oracle Generation



- Illegal instructions mark heap events
  - Simulated identically to legal instructions

Liveness Oracle Generation



- Record allocations, memory accesses
  - Preserve code, type objects, etc.
  - May use objects without accessing them

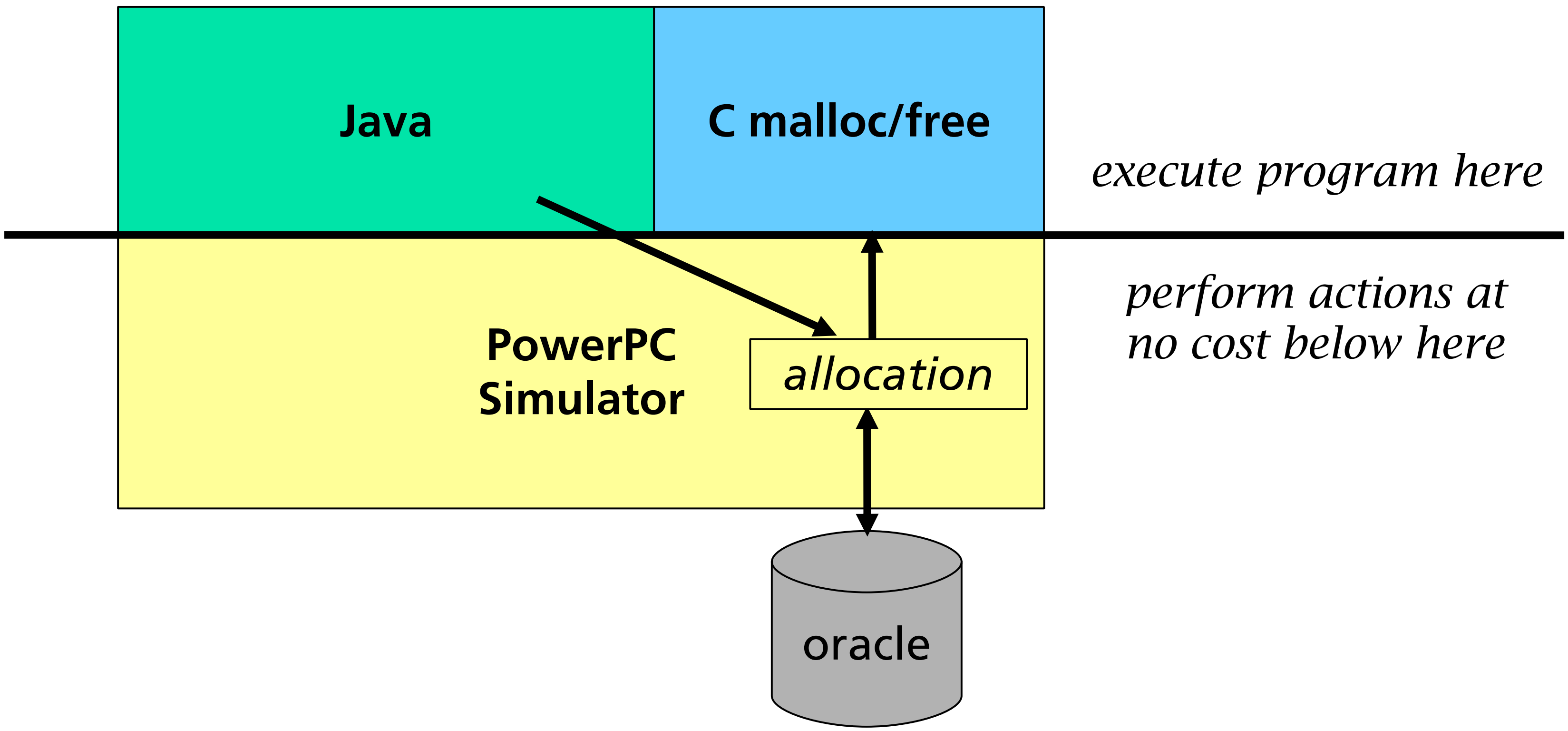
Liveness Oracle Generation


```
if (f.x == y) { … }
```

uses address of `f.x`,
but may not touch `f.x` or `f`

- Preserve code, type objects, etc.
- May use objects without accessing them

Oracular Memory Manager



- Consult oracle before each allocation
  - When needed, modify instructions to call **`free`**
  - Extra costs hidden by simulator

Experimental Methodology

- Java platform:
  - MMTk/Jikes RVM(2.3.2)
- Simulator:
  - Dynamic SimpleScalar (DSS)
  - Simulates 2GHz PowerPC processor
    - G5 cache configuration

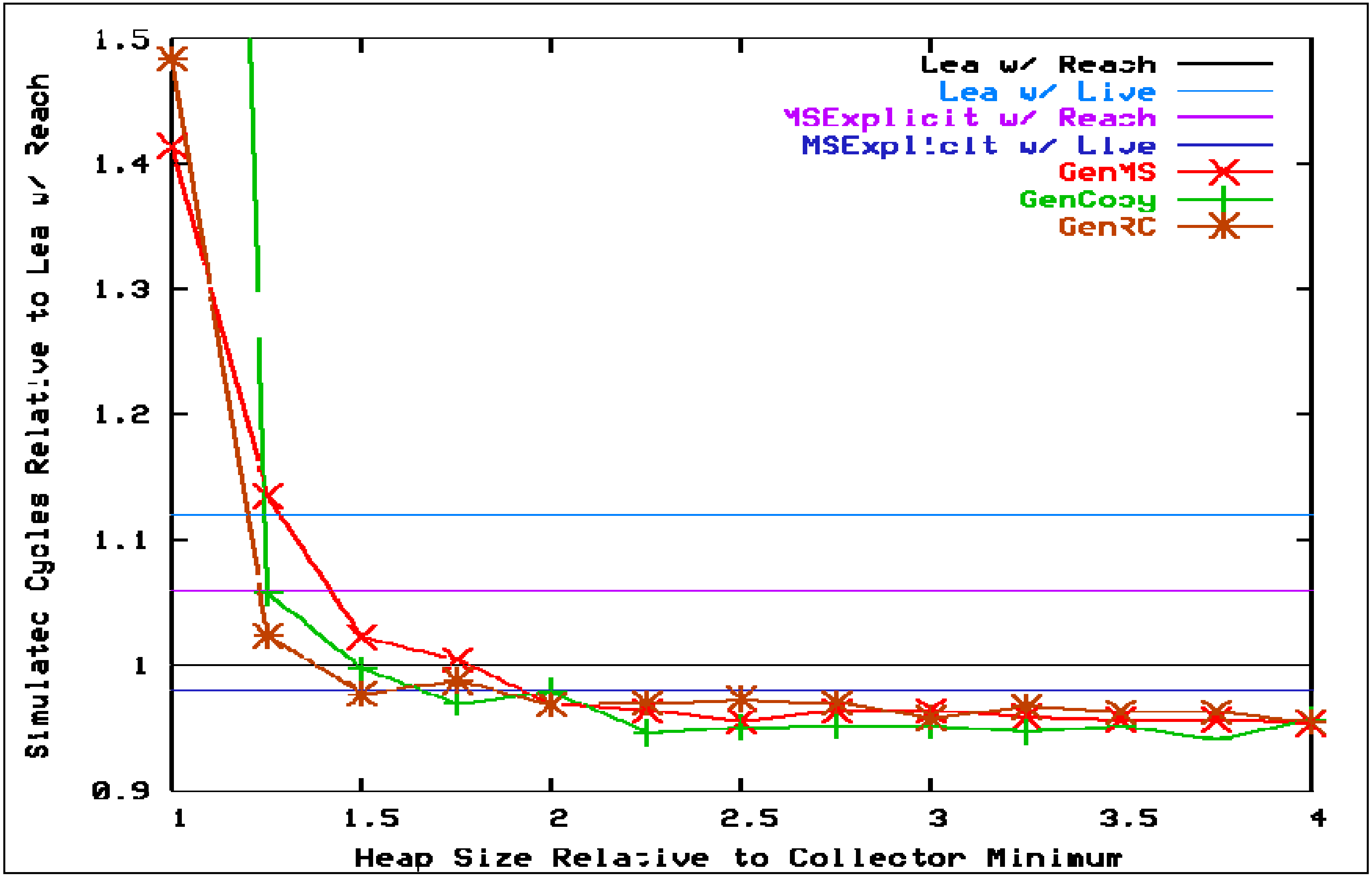
Experimental Methodology

- Garbage collectors:
  - GenMS, GenCopy, GenRC, SemiSpace, CopyMS, MarkSweep
- Explicit memory managers:
  - Lea, MSExplicit (MS + explicit deallocation)

Experimental Methodology

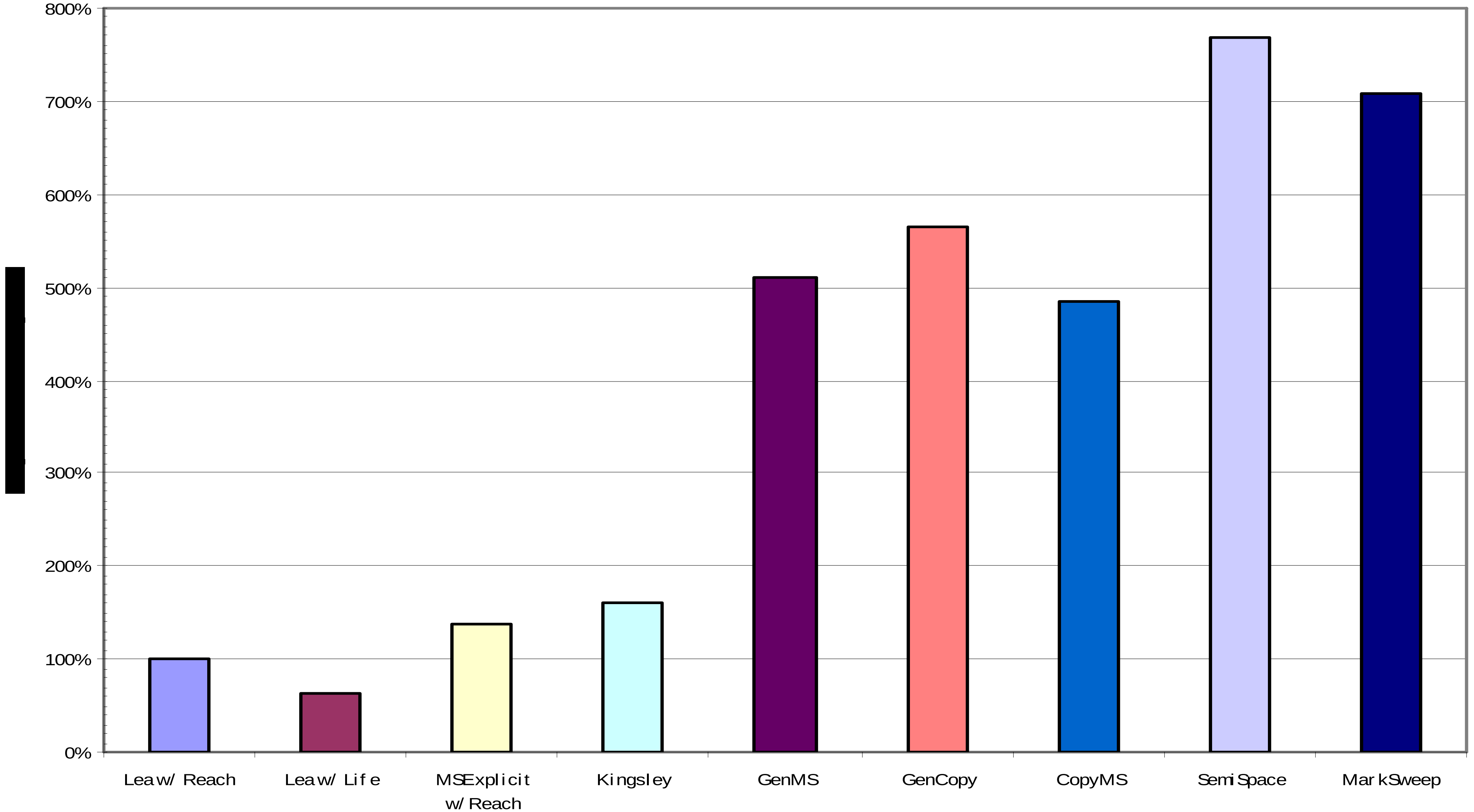
- Perfectly repeatable runs
  - *Pseudoadaptive* compiler
    - Same sequence of optimizations
    - Advice generated from mean of 5 runs
  - Deterministic thread switching
  - Deterministic system clock
  - Use "illegal" instructions in *all* runs

Execution Time for pseudoJBB



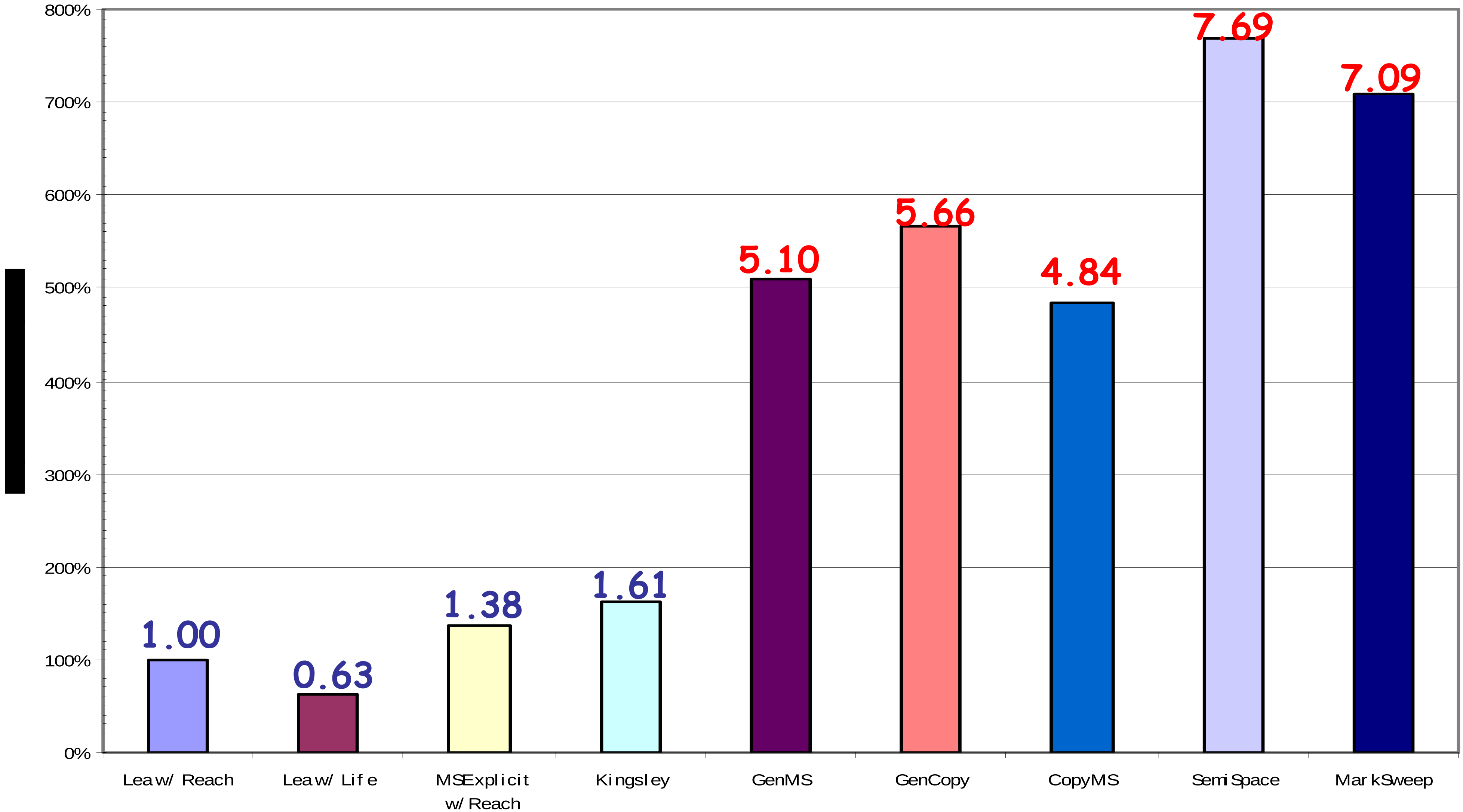
GC performance can be competitive

Footprint at Quickest Run



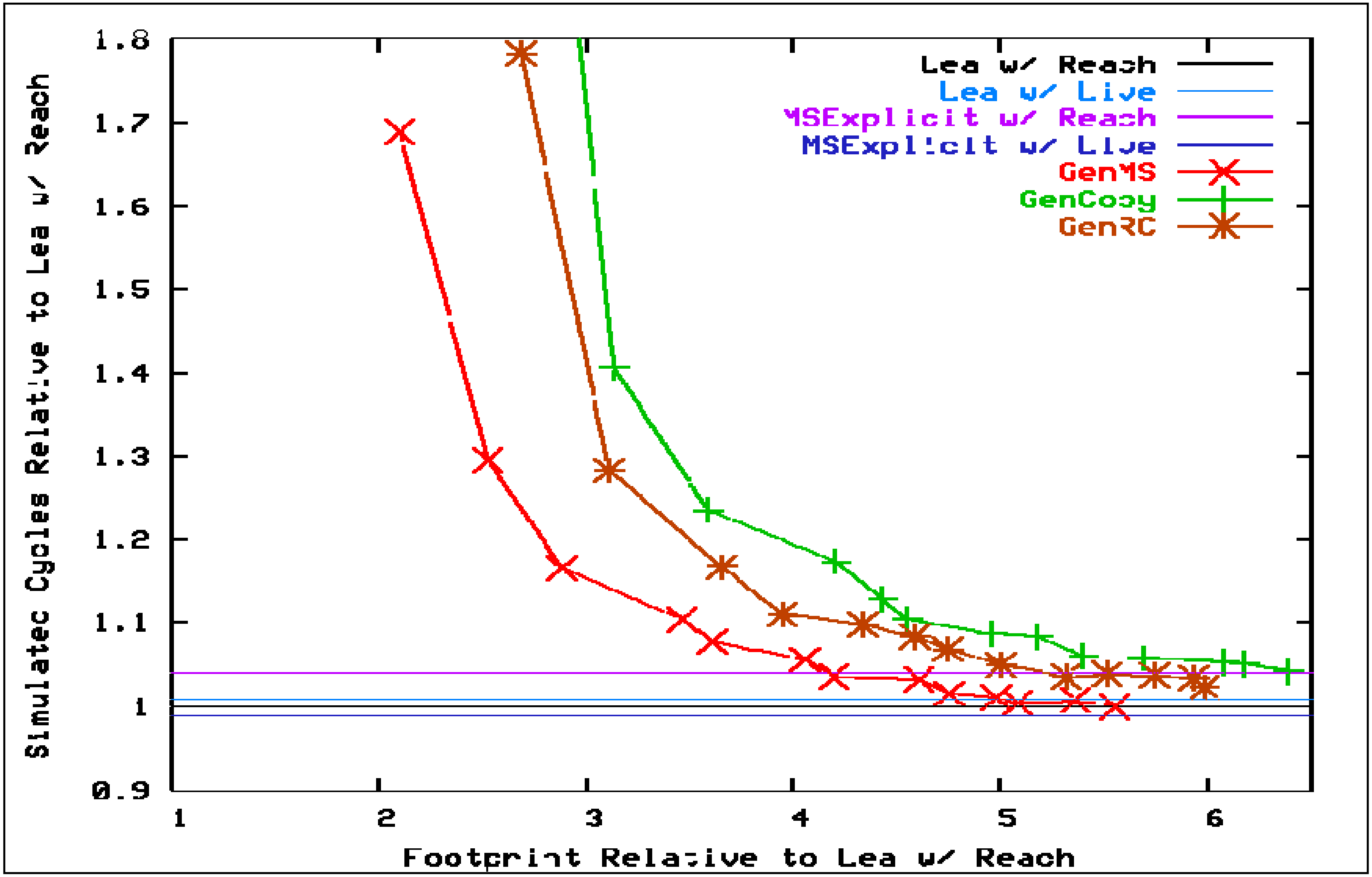
GC uses much more memory

Footprint at Quickest Run



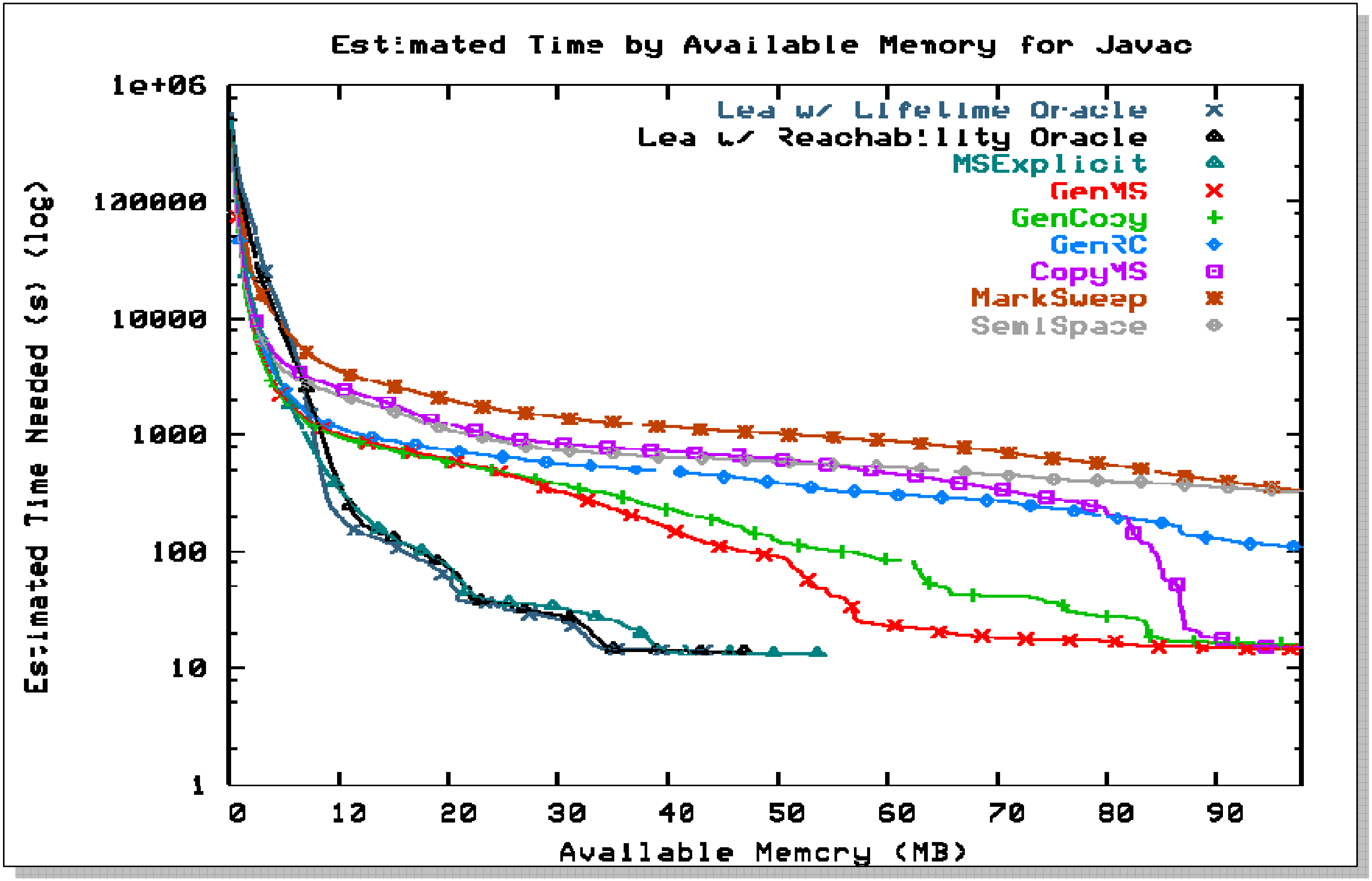
GC uses much more memory

Avg. Relative Cycles and Footprint



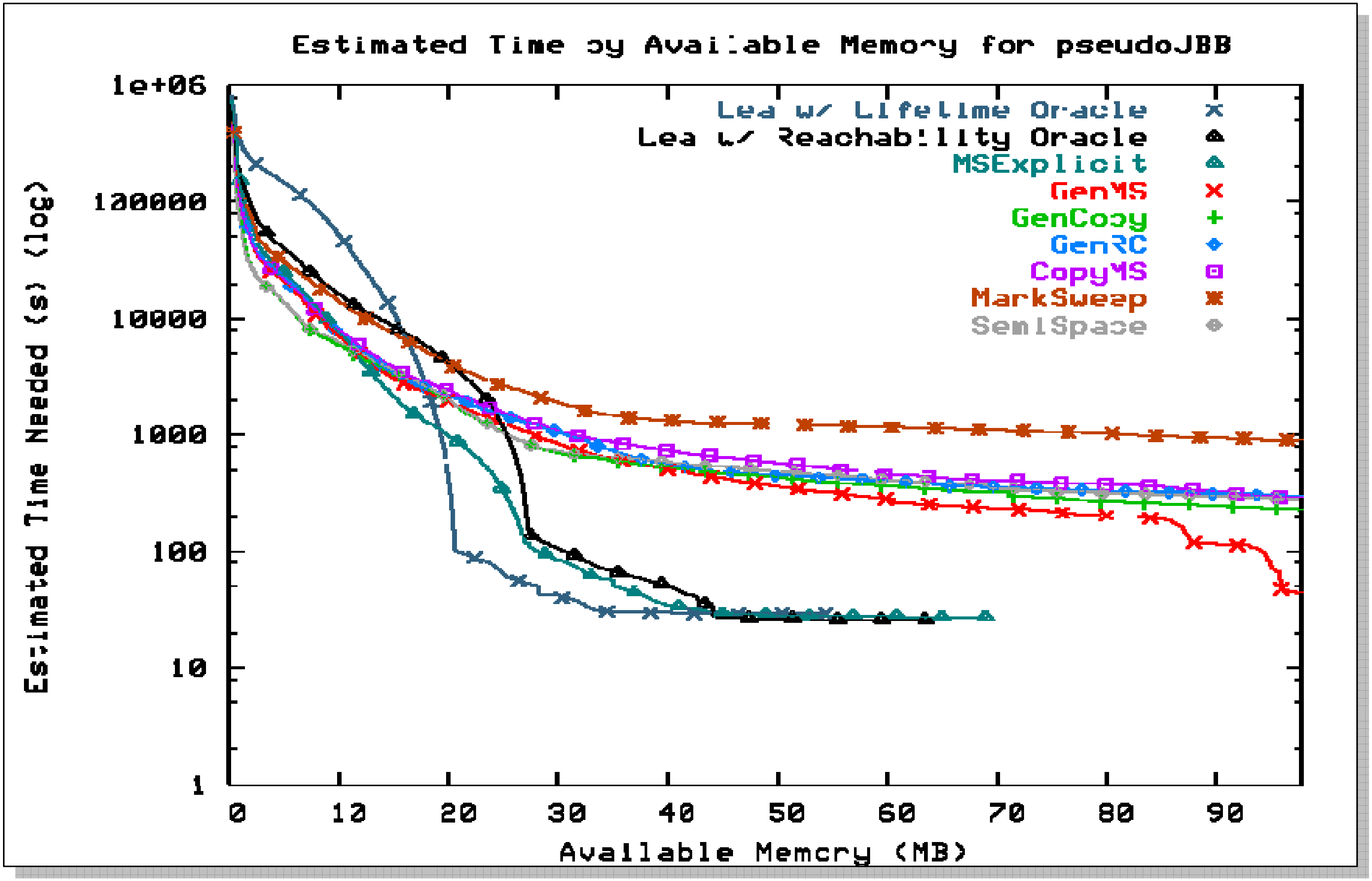
GC trades space for time

Javac Paging Performance



Much slower in limited physical RAM

pseudoJBB Paging Performance



Lifetime analysis adds little

Summary of Results

- Best collector equals Lea's performance…
  - Up to 10% faster on some benchmarks
- … but uses more memory
  - Quickest runs use 5x or more memory
  - At least twice mean footprint

Take-home: Practitioners

- GC ok if:
  - system has more than 3x needed RAM,
  - and no competition with other processes

- GC not so good if:
  - Limited RAM
  - Competition for physical memory
  - Relies upon RAM for performance
    - In-memory database
    - Search engines, etc.

Take-home: Researchers

- GC performance **already good enough** with enough RAM
- Problems:
  - Paging is a killer
  - Performance suffers when RAM limited

Future Work

- Obvious dimensions
  - Other collectors:
    - Bookmarking collector [PLDI 05]
    - Parallel collectors
  - Other allocators:
    - New version of DLmalloc (2.8.2)
    - VAM [ISMM 05]
  - Other architectures:
    - Examine impact of cache size

Future Work

- Other memory management methods
  - Regions, reaps

Conclusion

- Code available at: http://www-cs.canisius.edu/~hertzm