Performance of Non-Moving Garbage Collectors

Hans-J. Boehm

HP Labs

Why Use (Tracing) Garbage Collection to Reclaim Program Memory?

- Increasingly common
  - Java, C#, Scheme, Python, ML, ...
  - gcc, w3m, emacs, Amazon.com's web server, ...
  - C/C++ leak detectors
- For the usual reasons:
  - Frees user from explicit memory deallocation.
  - Eliminates common source of "memory smashes".
  - Facilitates maintenance of large systems.
    - Symptoms of bugs stay local.
  - Simpler interfaces reduce development effort.
  - Essentially required for untrusted code (Java, .NET)
  - Performance of multithreaded applications?

Why is it still interesting?

- GC performance is often misunderstood.
- Some basic techniques are often overlooked, even by researchers.
- There are real tradeoffs between garbage collection and more explicit memory management. This isn't widely appreciated.
- It's an interesting algorithmic problem that influences many applications.

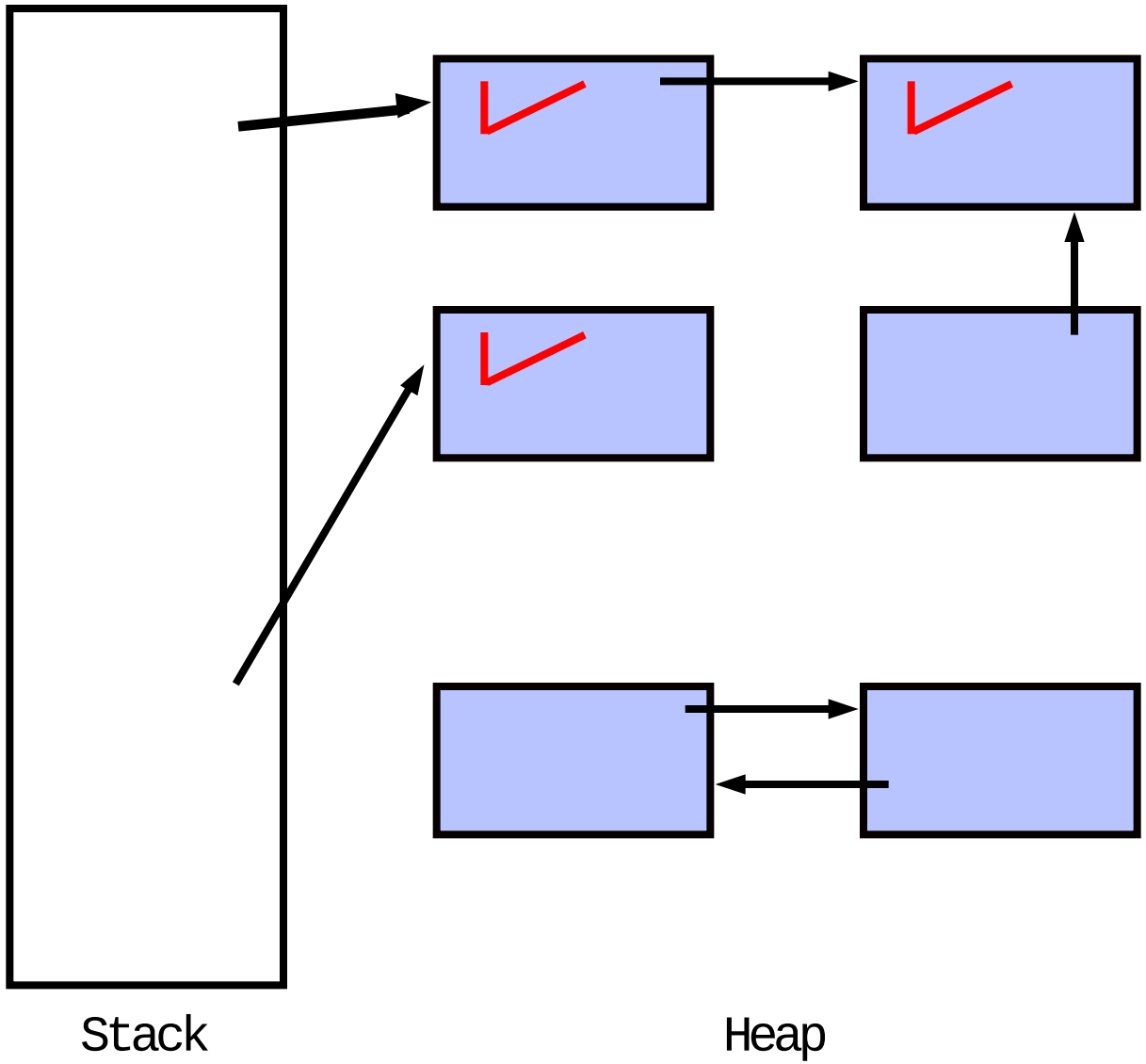
Why Non-Moving?

- Because I only have an hour ...
- Because it's hard to move objects for C programs.
- Because it's easy to compare to malloc/free.
- Because the fundamental performance considerations don't change much.
  - Many Java/ML/Scheme implementations have "faster" garbage collectors that may move objects, but
  - They're not consistently faster.

Basic Non-moving GC algorithm

- Mark all objects referenced directly by pointer variables (roots).
  - For c, we guess conservatively: writeable data, stack, registers.
- Repeatedly:
  - Mark objects directly reachable from newly marked objects.

- Identify unmarked objects (sweep).
  - E.g. put them on free lists, one per size.
  - Reuse to satisfy new allocation requests.


Stack
Heap

Easy Performance Issue 1

- Memory accesses tend to dominate performance.
- Each reclaimed object is touched twice per cycle:
  - sweep
  - allocate

- Solution:
  - Sweep lazily
    - in small increments
    - just before allocation

Easy Performance Issue 2

- If heap is nearly full, we collect frequently.
  - May collect once per allocation.
  - We touch all live objects in mark phase ==> expensive.

- Solution:
  - Make heap *e.g.* 1.5 times larger than necessary.
  - Each cycle, allocate n/3 bytes, trace 2n/3 bytes.
  - Trace/mark 2 bytes per byte allocated.

Basic Memory Management Comparison

GCBench

- Good as basic "sanity check"
- Mostly allocates and drops complete binary trees of various heights.
- Keeps some permanent data structures:
  - relatively large tree + pointerfree array
- Tree nodes are small:
  - 2 pointers plus 2 "int"s.

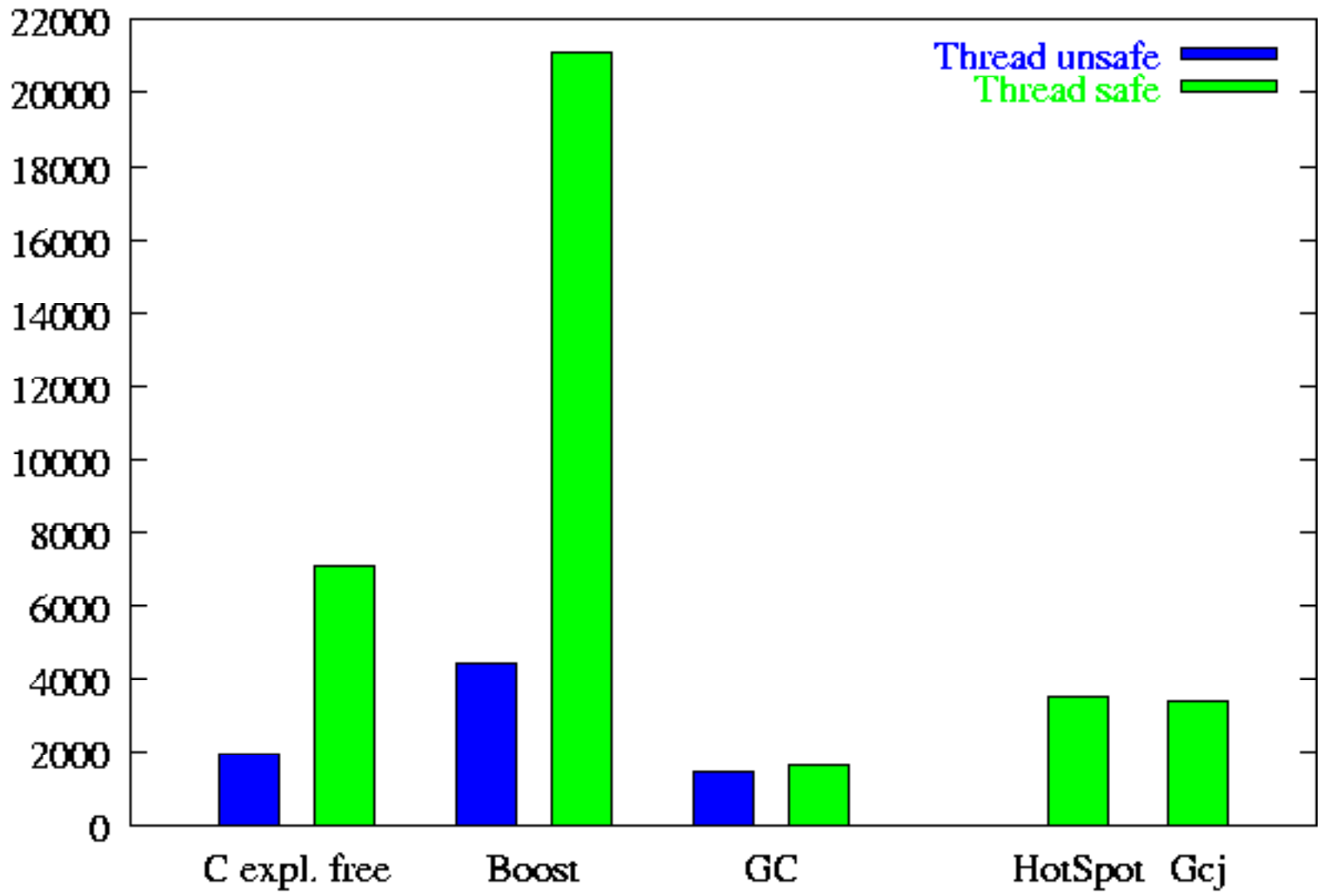
Compare:

- C with glibc (Doug Lea) malloc/explicit free
- Boost shared_ptr (C++ reference counting)
- Our tracing collector (C version of benchmark)

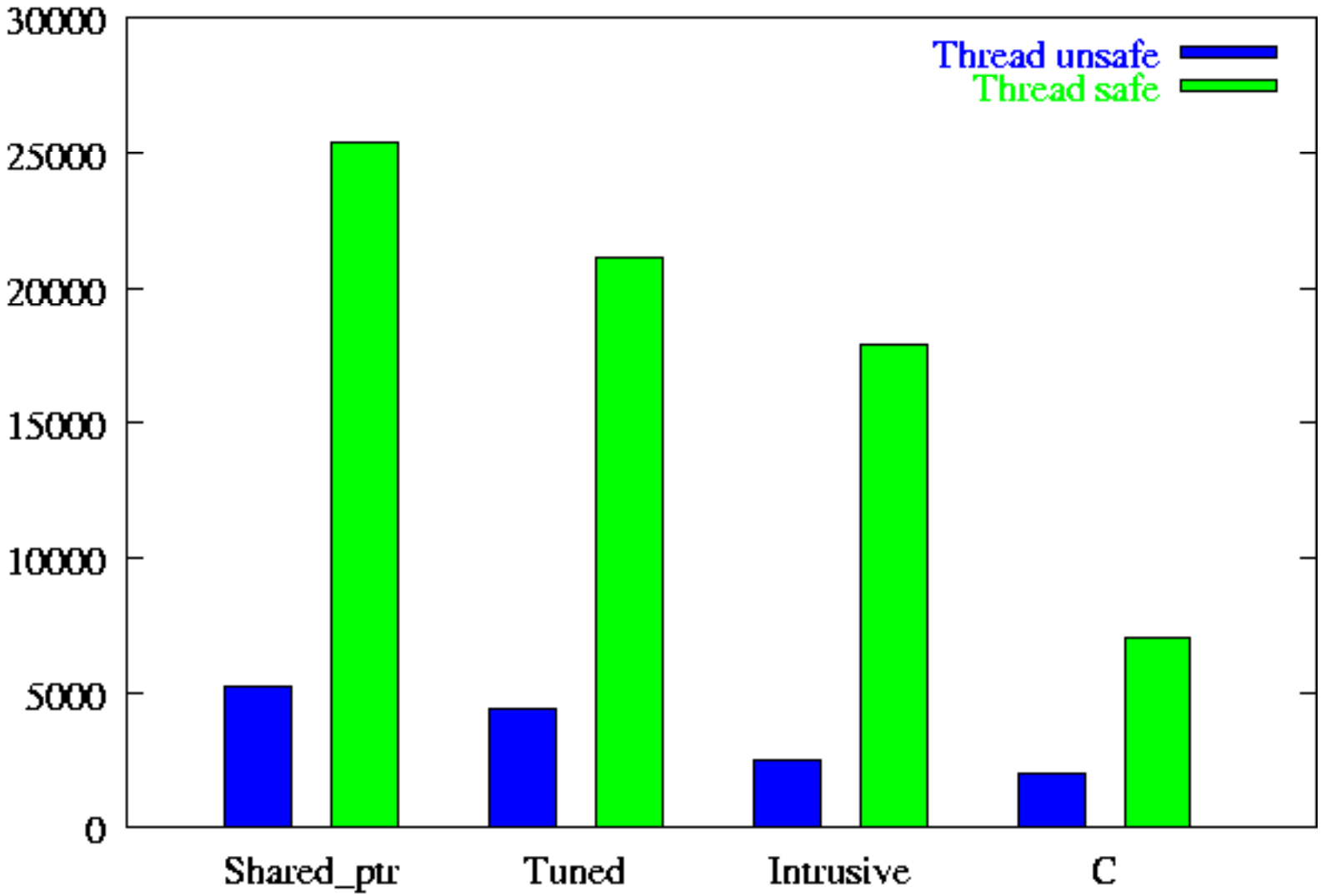
Reference Counting Review

- Each object has associated count of incoming references.
- Pointer assignments etc. update counts.
- When count reaches zero:
  - decrement counts for pointers in object.
  - deallocate object.
- Fails for circular references.
- Boost shared_ptr maintains count in separate object; counting hidden by C++ "smart pointer".
- Pointer assignment may involve two atomic adds.

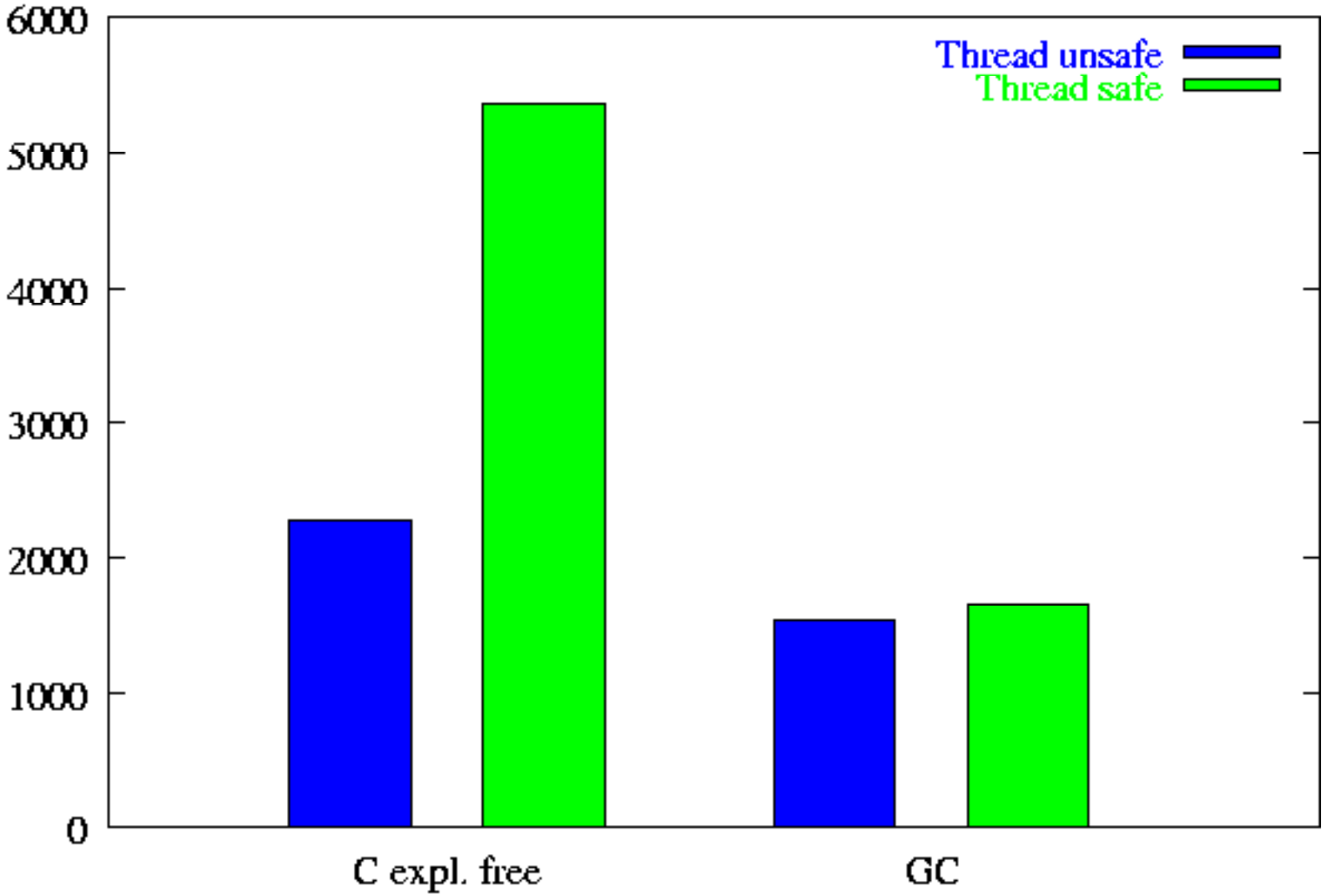
Execution Time (msecs, 2GHz Xeon)



Boost Reference Counting Details



Execution Time (msecs, 1Ghz Itanium 2)



Execution Time Observations

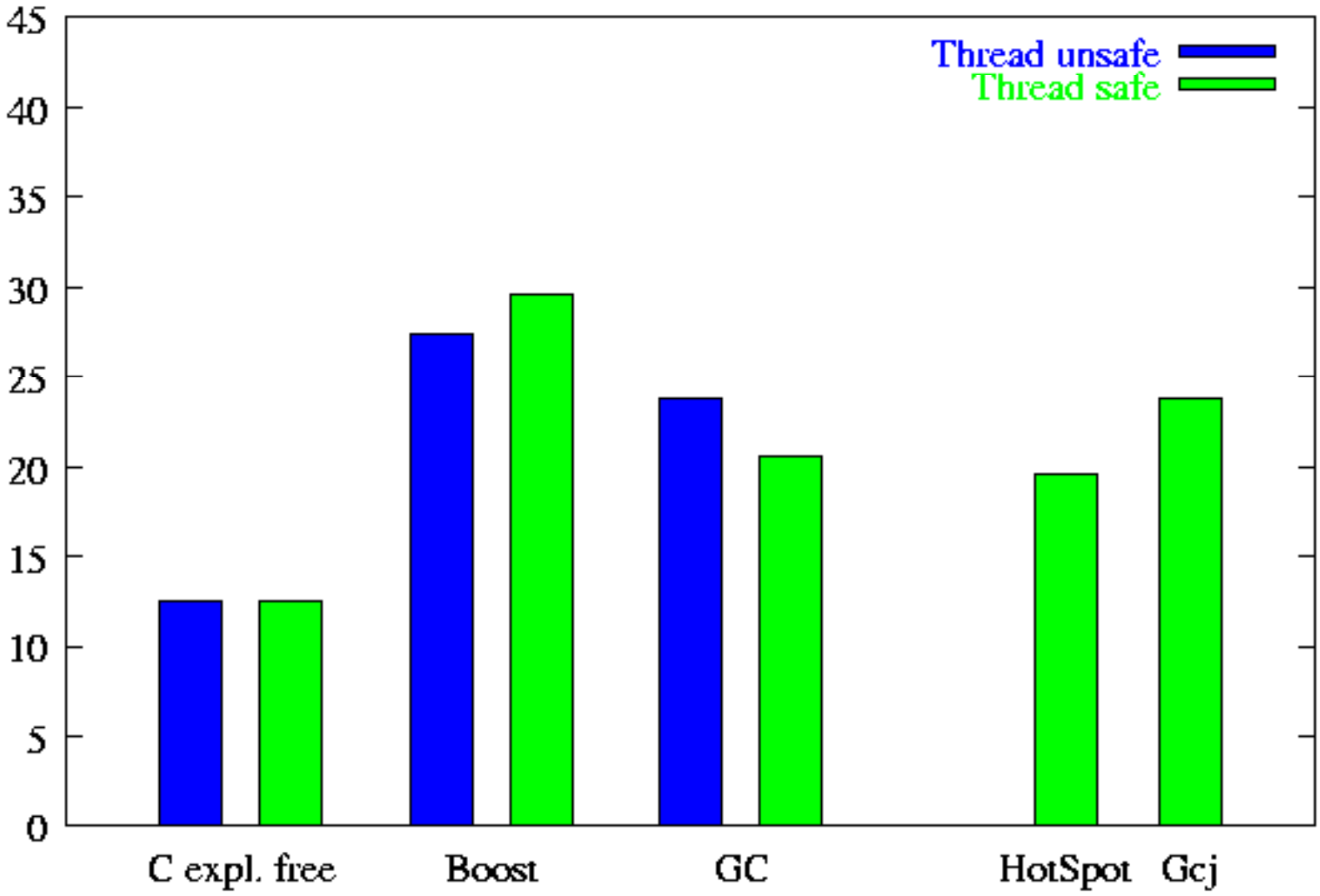
- Reference counting small objects is expensive.
  - Thread-safe reference counts are *very* expensive.
  - And benchmark just allocates & deallocates.
- GC wins for small objects.
  - En masse deallocation is cheaper.
  - Collecting allocator generally trades space for time.
  - May outweigh malloc/free locality benefits.
  - Address-ordered allocation may help.

Execution time Observations 2

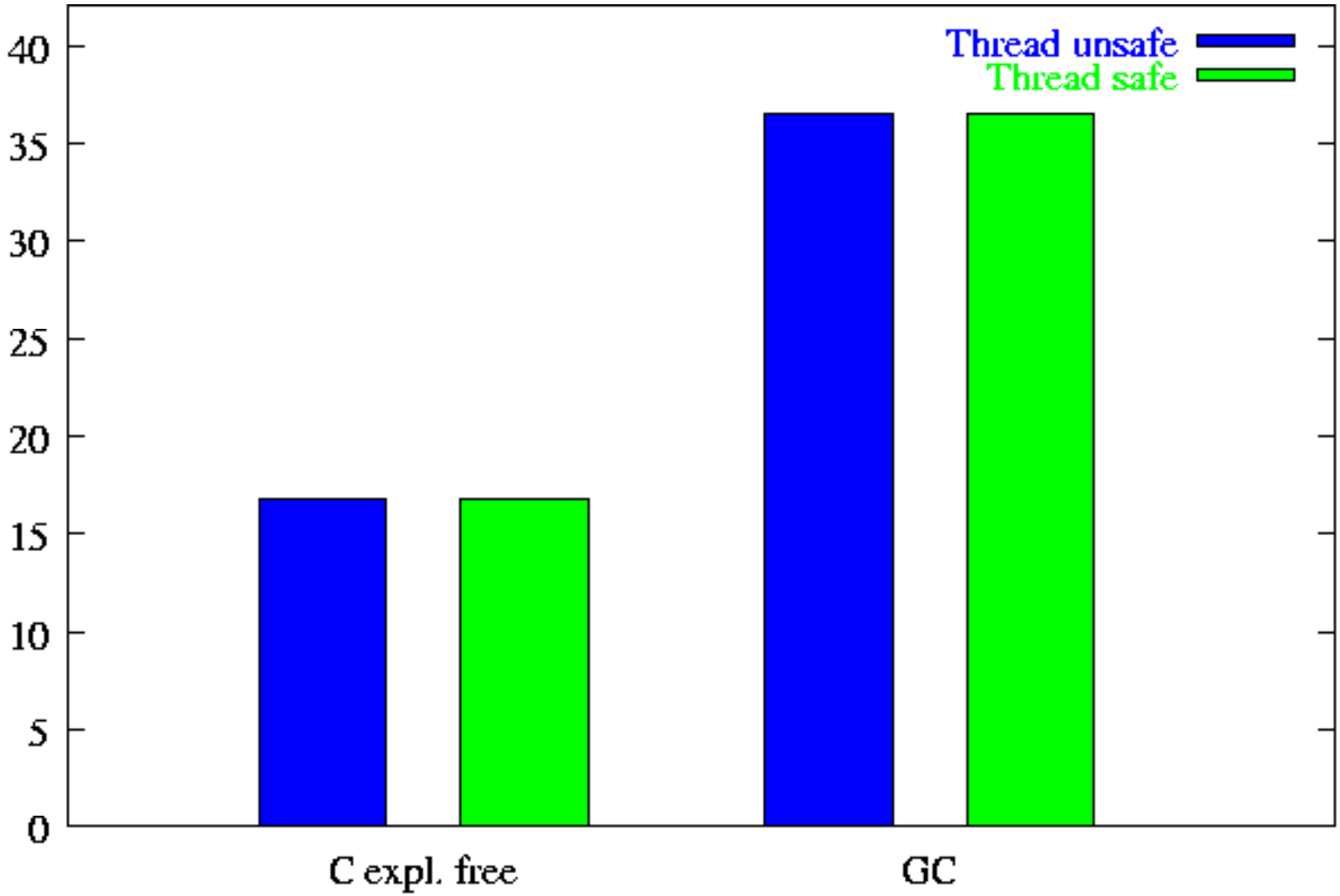
- Thread-safety is much cheaper for GC.
  - Uses per-thread allocation buffers.
  - Lock is acquired:
    - rarely for allocations.
    - only once per GC for deallocations.

  - Pervasive strategy for Java.
  - Less natural for malloc/free.
    - Need to transfer object from free() thread to malloc() thread.

Max Heap Size (MB, 2GHz Xeon)



Max Heap Size (MB, 1Ghz Itanium 2)



Heap Size Observations

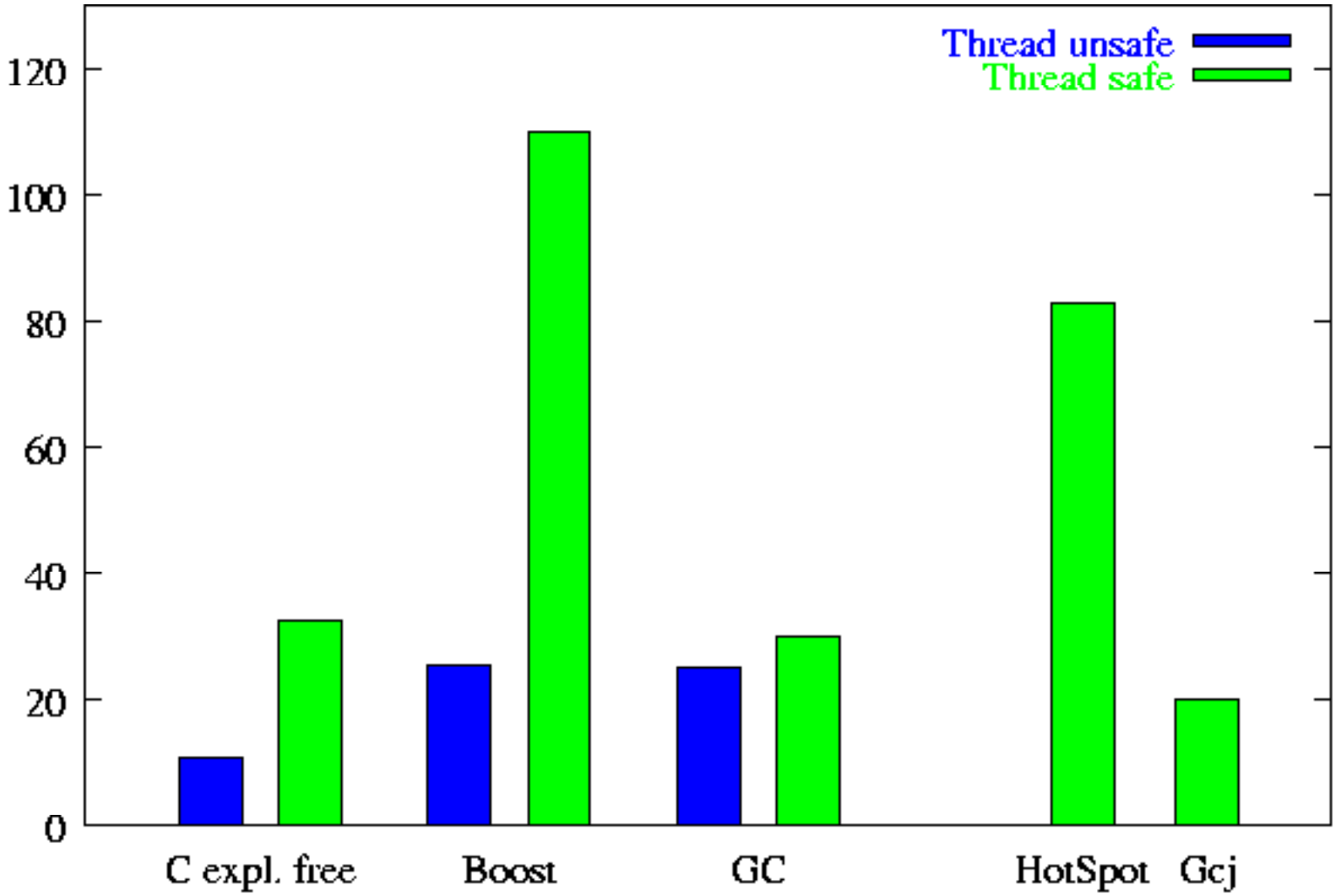
- Reference count overhead can be substantial.

- GC overhead is usually substantial:
  - Extra space needed to reduce GC frequency.
  - Allocator is less space efficient.
  - Conservative collector overallocates by 8 bytes.
  - But explicit free required no overhead here.

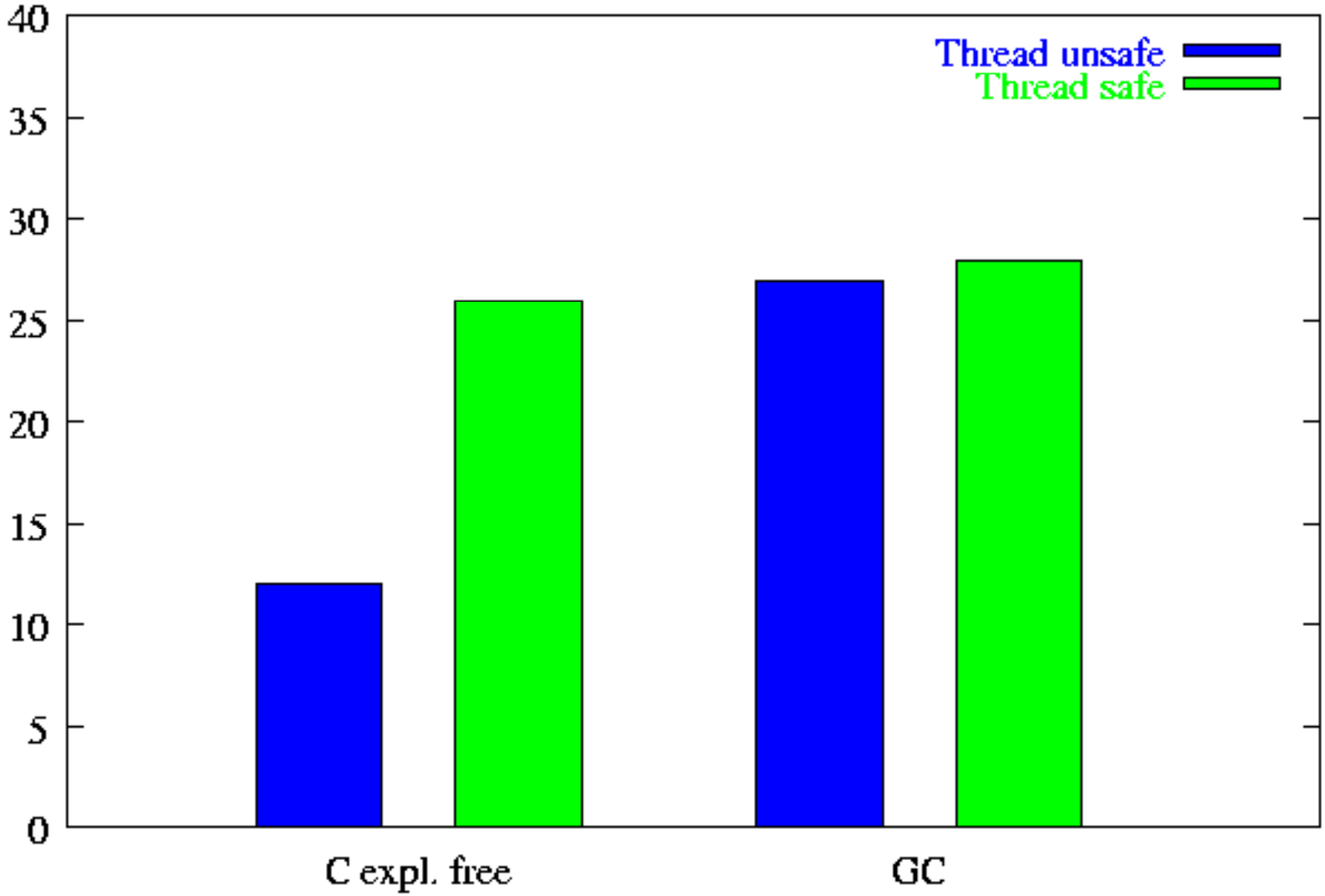
Pause Time

- All approaches have some "pause time":
  - GC configured to "stop the world" during GC.
  - A reference count assignment may remove the last reference to a large data structure.
    - Boost shared_ptr stops and deallocates it then.
    - Alternatives waste space.
  - Malloc/free benchmark recursively deallocates trees.

Max Pause Time (msecs, 2GHz Xeon)



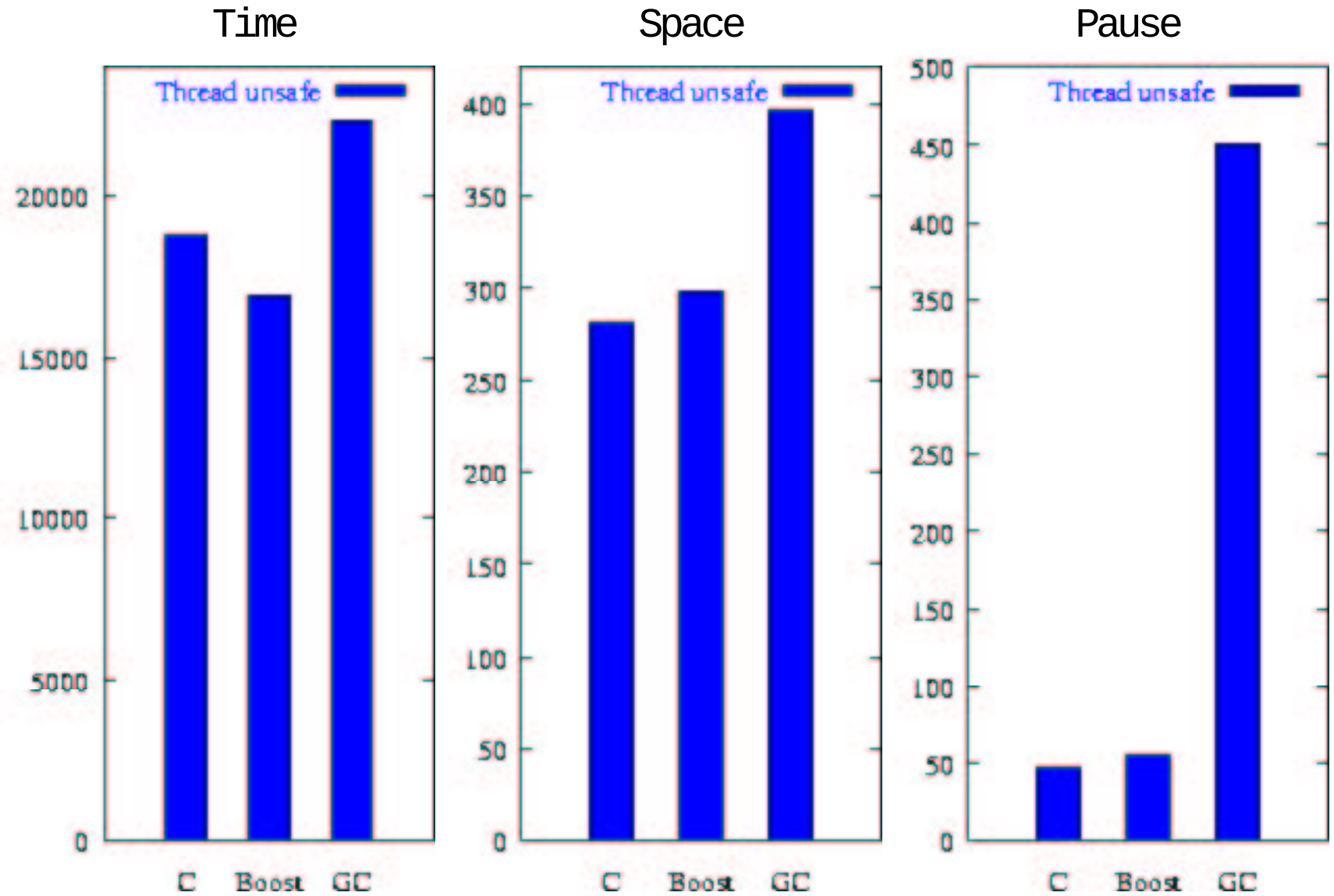
Max Pause Time (msecs, 1Ghz Itanium 2)



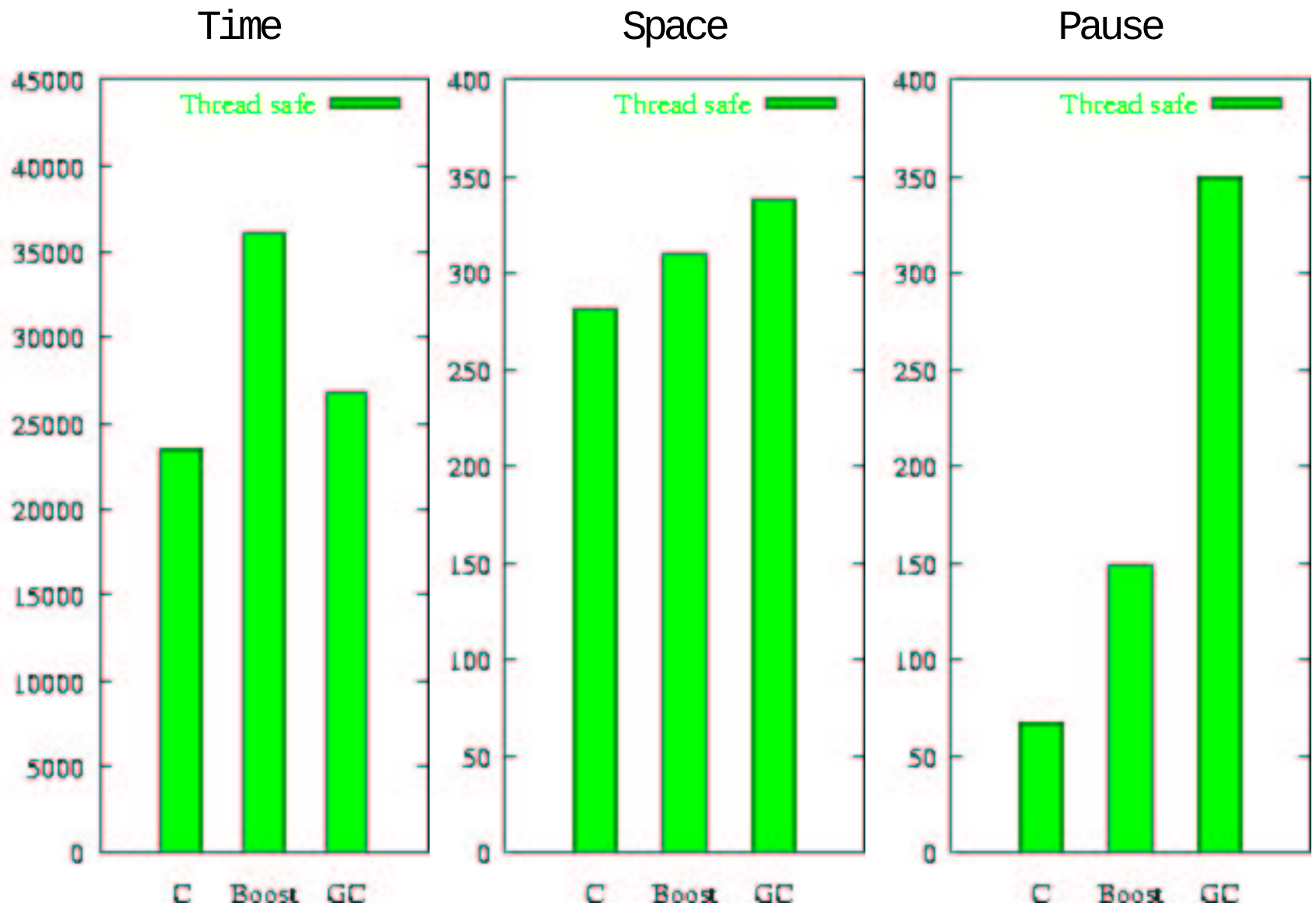
Effect of Object Size

- Malloc-free allocation cost is almost independent of object size.
  - Of course, initialization cost is not.
- With N "extra" bytes in a garbage collected heap, only N bytes can be allocated between collections.
  - The larger the objects, the fewer allocations per GC.
  - Total allocation/GC cost is roughly proportional to object size.
- Redo benchmark with additional 128 NULL pointers per tree node (528 bytes/object total):

Large Objects (msecs, MB, 2GHz Xeon)



Large Objects (thread-safe)



Can GC Do Better?

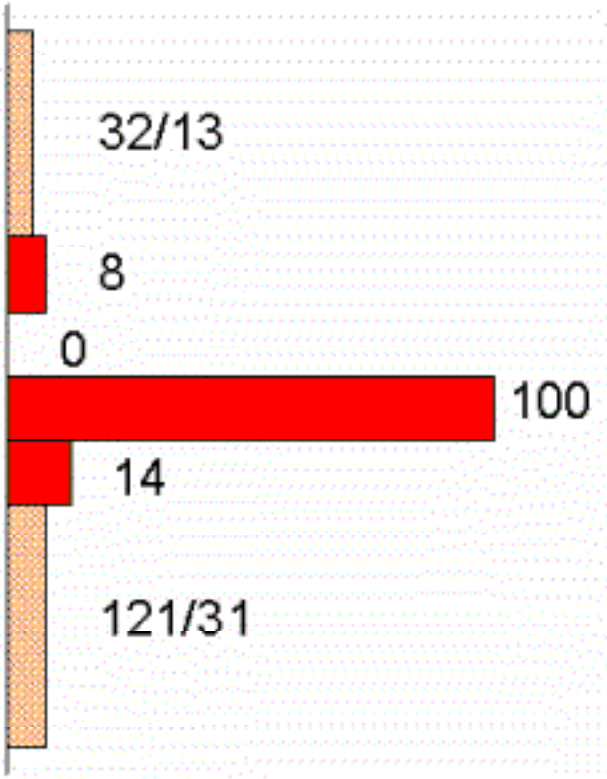
Marker profile

```
GC_mark_from_mark_stack:

...

mov   0xffffffec(%ebp),%edi
mov   (%edi),%ebx
cmp   0xfffffffdc(%ebp),%ebx
jb    0x804c51f

...
```



Prefetching during trace

- Much of the tracing cost is cache miss on initial object access.
- Idea:
  - We need to visit all objects O referenced by *X*.
  - Prefetch all O, and push them on "to be traced" stack".
  - As before trace top object on stack.
  - Usually prefetched object will still be in cache when traced.
- Unfortunately, benefit depends in microarchitecture.
  - X86 prefetch instructions are inconsistent

Prefetching during trace (contd.)

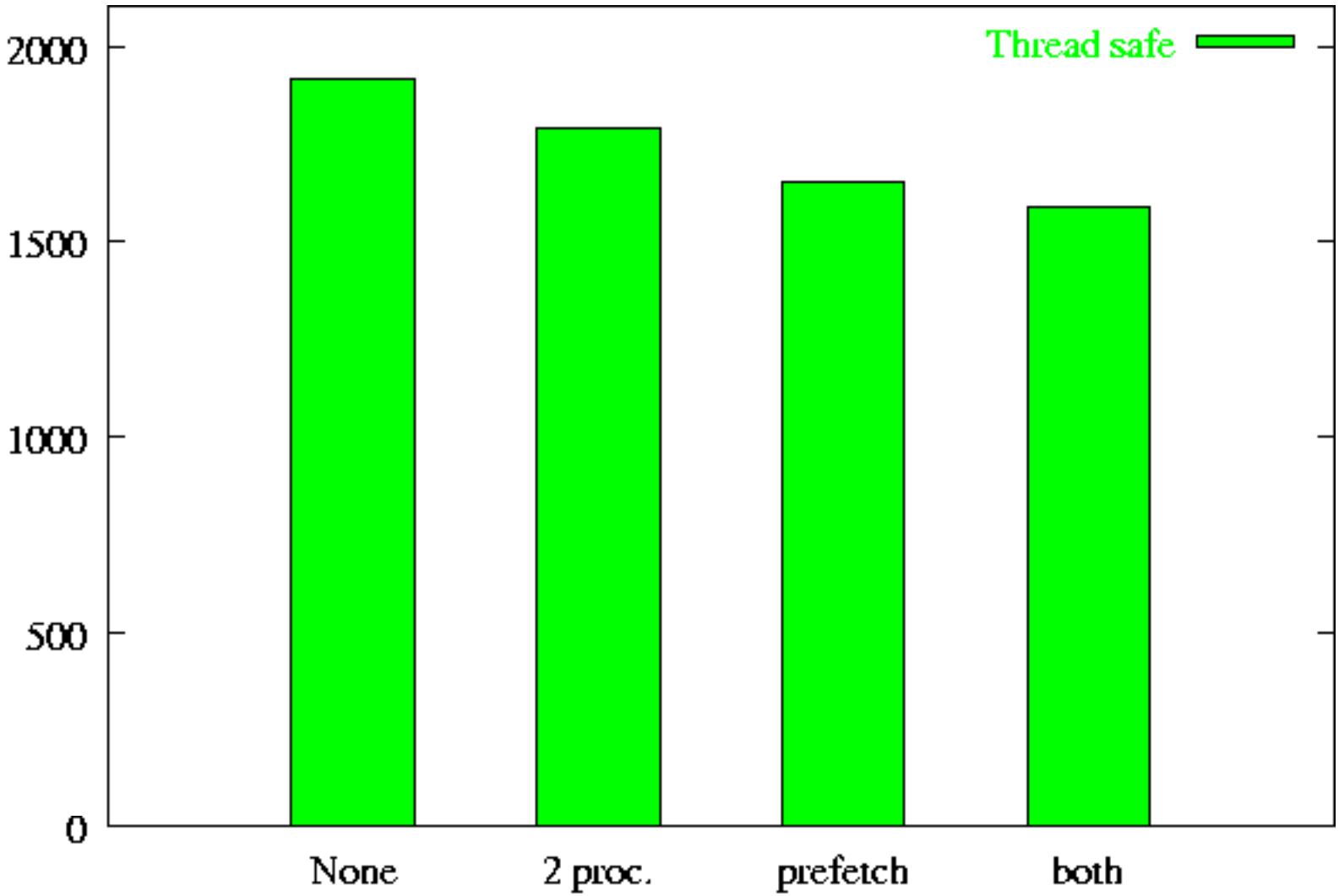
- Currently implemented for some architectures, *e.g.* Itanium.
- Portable X86 implementation is hard.
  - Instructions vary.
  - Less important on Pentium 4 (hardware prefetch)?

- Corresponding cache miss problem on object allocation.
  - Possible solutions fit less well.
  - "Cache line clear" instruction might help.

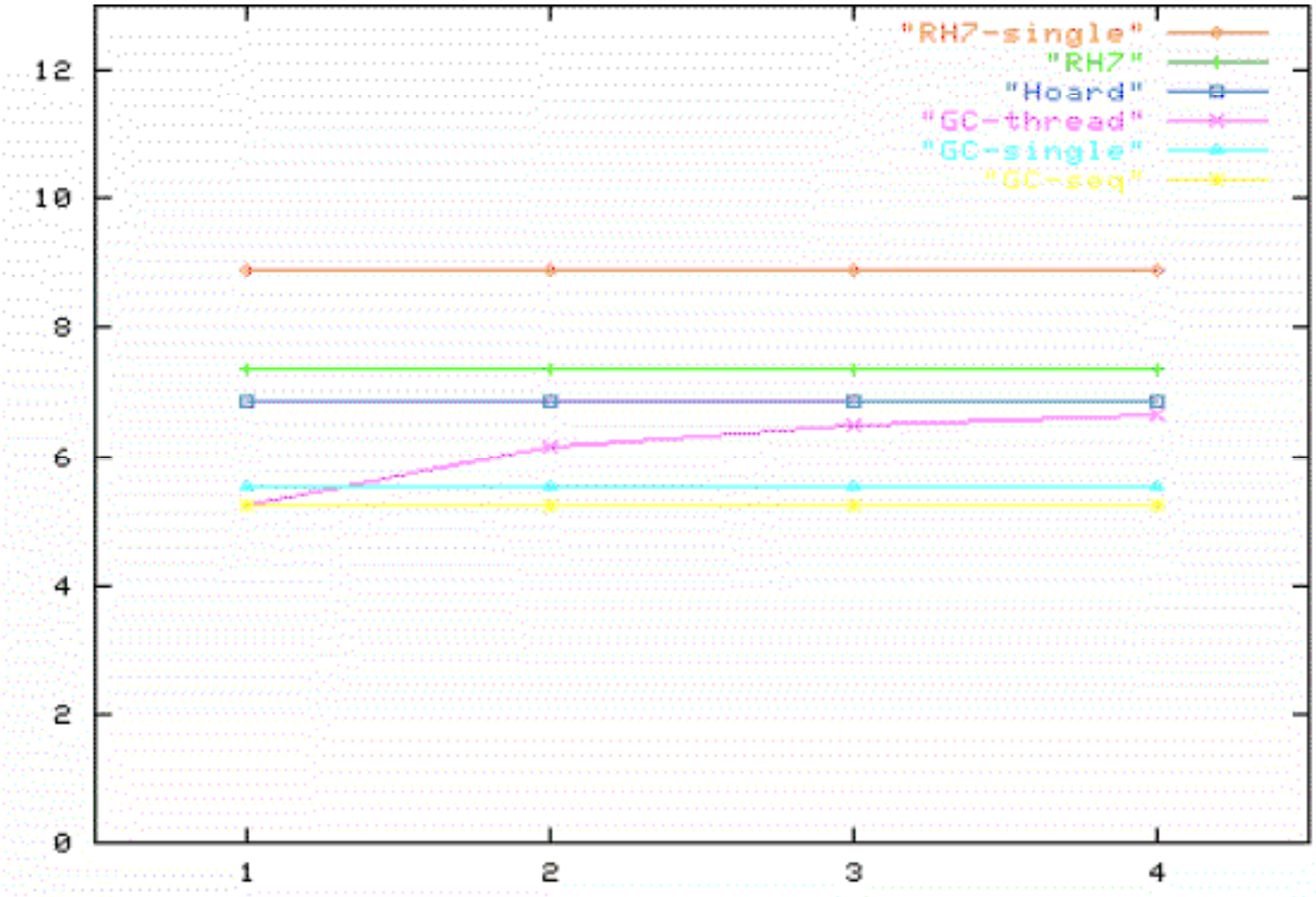
Taking advantage of multiprocessors

- Most garbage collection time is spent in the mark phase.
- The mark phase is easily parallelizable.
  - Garbage collector still stops all client threads, but
  - Wakes up several marker threads.
  - Each has separate stack of pending tracing work
  - Central work queue for load balancing.

GCBench Improvements (Itanium 2)



Ghostscript throughput



Things to remember

- GC generally performs well and simple reference counting performs poorly with:
  - threads, multiprocessors, small objects
- GC performs poorly and reference counting often performs well with
  - large, especially pointer-free, objects
- Beware of reference counting to avoid pause times.

- Prefetch instructions aren't just for compilers anymore.

Outtakes

- What went wrong with the measurements:
  - Prefetching doesn't work for toy benchmarks on Pentium 4
    - The hardware already does it.
  - For large objects, the "tuned" reference count implementation was far slower.
    - Seems to hit malloc performance glitch.