The "Boehm-Demers-Weiser" Conservative Garbage Collector

Hans-J. Boehm
HP Labs

Outline

- Introduction
  - Interface
  - Implementation basics & goals
- Implementation details and issues
  - Core collector
  - Enhancements
- Experiences and a few measurements

What is it?

- A garbage collecting replacement for C's **`malloc()`**.
  - Calls to **`free()`** are optional.
  - "Unreachable" memory is automatically reclaimed, and made available to future **`malloc()`** calls.
- A tracing (mark/sweep) garbage collector.
  - It periodically determines which objects can be reached by following pointers.
  - The rest can be reused for other purposes.
- An easy way to add garbage collection to a runtime system.
  - Easy to interface to.
  - Interacts well with C/C++ code.
  - Gcj (Java), Mono (C#, .NET), Bigloo (Scheme), MzScheme.
- A leak detector for programs that call **`free()`**.
  - Unreachable un**`free`**d memory is a memory leak.

Example: Lisp S-expressions

```
#include "gc.h"

typedef union se {struct cons * cp; int i;} sexpr;

struct cons { union se head; union se tail; };

#define car(s) (s).cp->head
#define cdr(s) (s).cp->tail
#define from_i(z) ({sexpr tmp; tmp.i=z; tmp;})
#define to_i(s) (s).i

sexpr cons(sexpr a, sexpr b) {
  sexpr tmp = {GC_MALLOC(sizeof(struct cons))};
  car(tmp) = a; cdr(tmp) = b;
  return (tmp);
};

int main() {
  return to_i(car(cons(from_i(0),from_i(1))));
}
```

Where did it come from?

- Began life (ca. 1980) as a simple GC for the Russell programming language. (Demers was original author.)
- Later (ca. 1985?) changed to remove restrictions on generated code, and allow use in the compiler itself.
  - Eliminate endless debugging of manual reference counting.
- Used for student compilers for a language with higher-order functions.
- Mark Weiser explored use as leak detector (ca. 1986).
- A variant served as the Xerox Cedar GC from the late 80s, replacing reference-count collector.
- Unrelated to an earlier garbage collector for C written by Doug McIlroy and apparently layered on top of malloc.

What else can it do?

- 20 years of creeping features, including:
  - Invoking finalizers after an object becomes unreachable.
  - Support for use in runtime systems.
    - If the compiler wants to help, it can.
  - Support for heap debugging.
    - What's in the heap?
    - Why is it still there? How can it still be referenced?
  - Support for threads and multiprocessor GC.
    - Maybe a way to speed up standard C applications on multiprocessors?
  - Various mechanisms for reducing GC pauses:
    - Incremental (but not hard real-time) GC.
    - "Generational" GC which concentrates effort on young objects. (But objects are not moved.)
    - Abortable collections.

What can't it do?

- Reclaim memory or invoke finalizers/destructors immediately.
  - Like all tracing garbage collectors, it only checks for unreachable memory occasionally.
  - And synchronous heap finalizers are broken anyway …
- Reclaim "all" unreachable objects.
  - Generally a few will still have pointers to them stored somewhere.
  - The GC doesn't know which registers will be referenced.
  - And there are other issues …
  - And "unreachable" isn't well-defined anyway…
  - But we generally avoid growing leaks.

Dealing with C: Conservative Garbage Collection

- For C/C++ programs, we may not know where the pointer variables (roots) are.
  - We may want to use a standard compiler. (Slightly risky with optimization, but popular.)
  - Program may use C unions.
- Even layout of heap objects may be unknown.
- It's easier to build a Java/Scheme/ML/… compiler if pointer location information is optional.
- Conservative collectors handle pointer location uncertainty:
  - **If it might be a pointer it's treated as a pointer.**
  - Objects with "ambiguous" references are not moved.
    - And we never move any objects.
  - May lead to accidental retention of garbage objects.

C Interface overview

Debugging support: GC_xyz() vs. GC_XYZ() functions:

- GC_xyz() is the real function.
- GC_XYZ(x) expands to either GC_xyx(x) or GC_debug_xyz(x, <source position, etc>).
- Clients should:
  - Use the all-caps version.
  - Always include gc.h.
  - Define GC_DEBUG before including gc.h for debugging.
- This is becoming obsolete technology.
  - Requires too much recompilation.
  - Libunwind, addr2line allow better alternatives.

C interface, main functions

- GC_MALLOC(bytes)
  - In simple cases, this is enough.
- GC_MALLOC_ATOMIC(bytes)
  - Allocate pointer-free or untraced (but collected) memory.
- GC_MALLOC_UNCOLLECTABLE(bytes)
  - Allocate uncollectable (but traced) memory.
- GC_REALLOC(p, bytes)
- GC_REGISTER_FINALIZER(…)
  - Register (or unregister or retrieve) "finalizer" code to be called when an object is otherwise "unreachable".
  - Unlike Java, by default, an object is reachable if it can be referenced from other finalizers. (Also Java variant.)

C interface, some more functions

- GC_INIT() – Optional on most platforms. (Must be called from main program on a few.)
- GC_FREE() – If you insist. (Usually helps performance for large objects, hurts for small ones.)
- GC_MALLOC_IGNORE_OFF_PAGE() – Like GC_MALLOC(), but for large arrays with pointers to (near) the beginning.
- Plus statistics, control of incremental GC, more allocator variants, heap limits, GC frequency controls, fast inline allocators, etc.

C++ interface

- "gc_cpp.h" provides a base class gc:
  - Overrides new to be GC_MALLOC for subclasses of gc.
  - Overrides ::new to be GC_MALLOC_UNCOLLECTABLE.
  - Provide gc_cleanup class which registers destructor as finalizer.
  - Built by Detlefs, Hull, based on Ellis, Detlefs work.
  - …
- "gc_allocator.h" defines STL allocators:
  - gc_allocator
  - traceable_allocator
- Particularly gc_cpp.h is annoyingly brittle.
  - Perhaps more so than some of the gross hacks we'll hint at later.
  - Replacing global operator new seems problematic for many compilers.

Environment variables

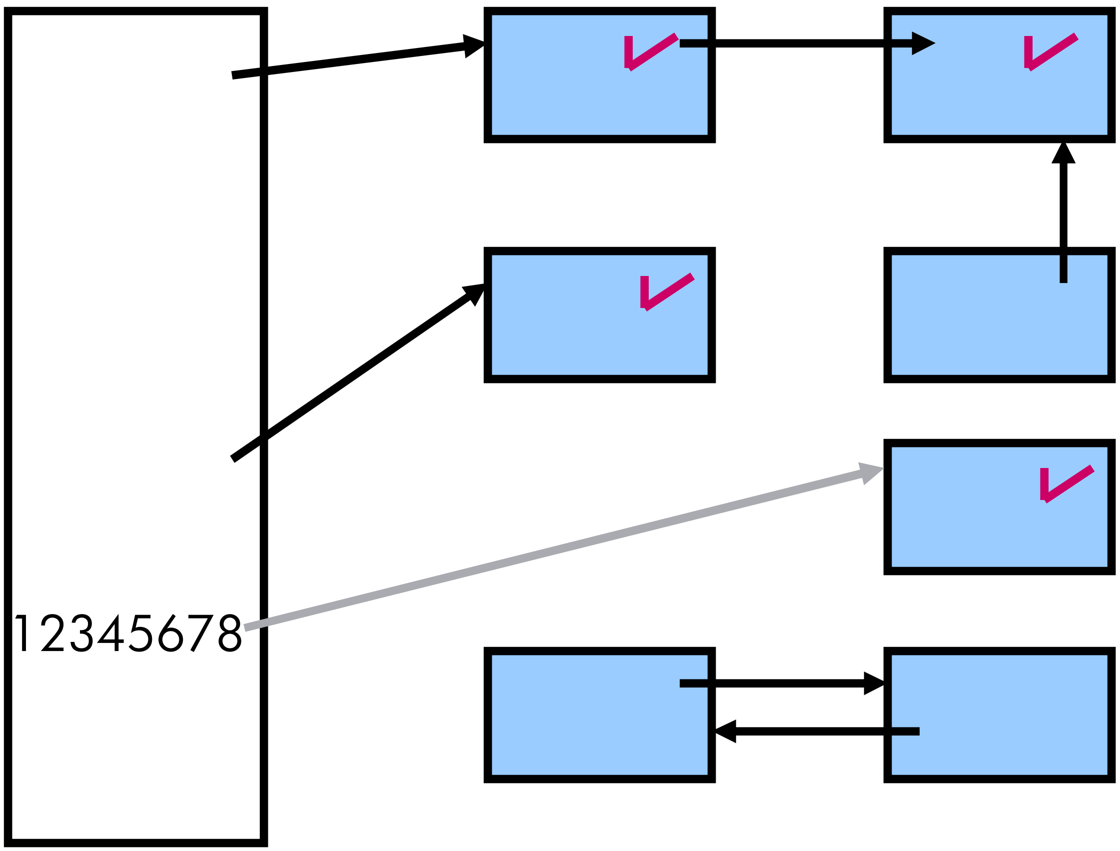
- Collector can be influenced by various environment variables:
  - GC_INITIAL_HEAP_SIZE
  - GC_MAXIMUM_HEAP_SIZE
  - GC_PRINT_STATS
  - GC_DUMP_REGULARLY
  - GC_ENABLE_INCREMENTAL (caution!)
  - GC_PAUSE_TIME_TARGET
  - GC_DON'T_GC
  - GC_IGNORE_GCJ_INFO – ignore compiler-provided pointer location information.
  - GC_MARKERS – Set the number of GC threads (where supported).
  - …

How does it work?

Occasionally (when we run out of memory?):

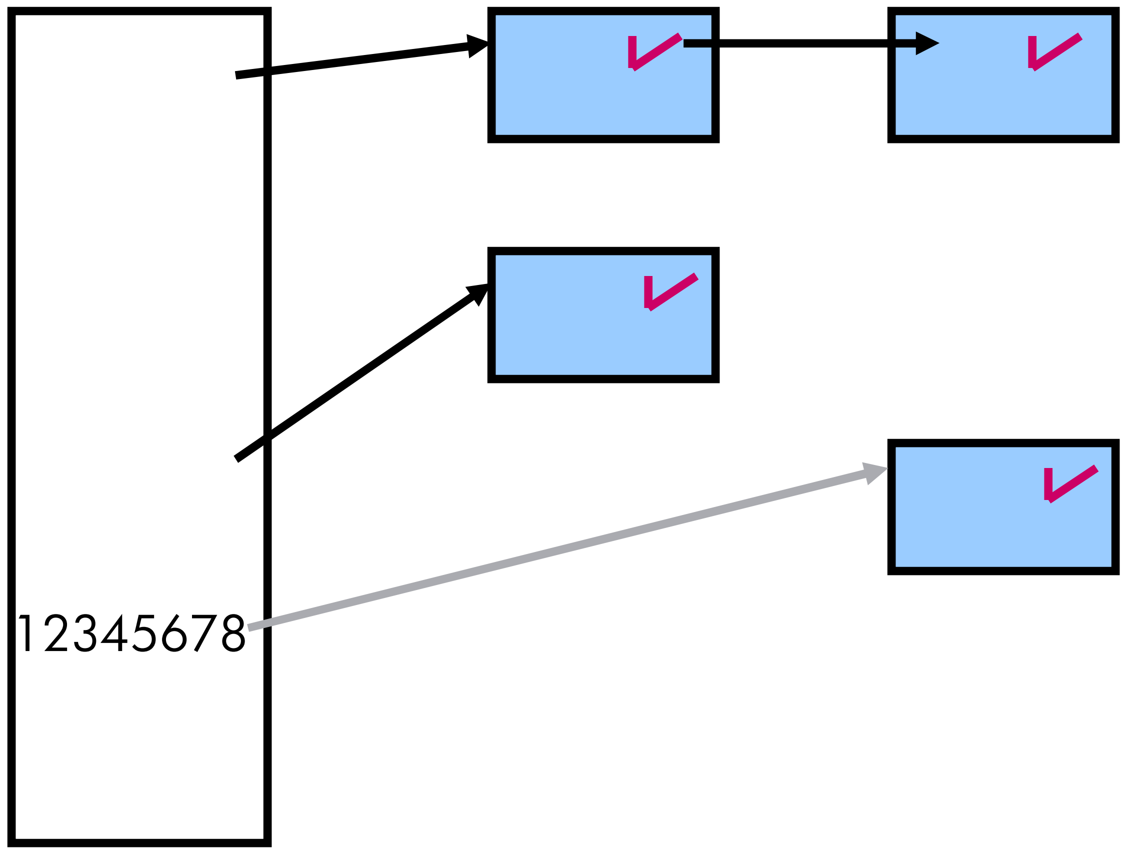
- Mark all objects referenced directly by pointer variables (roots)
- Repeatedly:
  - Mark objects directly reachable from newly marked objects.
- Finally identify unmarked objects (sweep)
  - E.g. put them in free lists.
  - Reuse to satisfy allocation requests.
- Objects are not moved.

Mark/sweep illustration



Stack w/ pointer variables

Mark/sweep illustration (2)



Stack w/ pointer variables

Easy performance issue 1

- If heap is nearly full, we collect too frequently.
  - May collect once per allocation.
  - We look at all reachable objects each time ➔ expensive
- Solution:
  - Always make sure that heap is e.g. 1.5 times larger than necessary.
  - Each cycle, allocate $n/3$ bytes, trace $2n/3$ bytes.
  - Trace 2 bytes per byte allocated.

Easy performance issue 2

- Performance is often dominated by memory accesses.
- Each reclaimed object is touched twice per cycle.
  - Once during sweep phase.
  - Once during allocation.
- Solution:
  - Sweep a little bit at a time before allocation.
  - Try to keep object in cache.
  - “Sweep phase” is a misnomer.
  - Imposes constraints on GC data structure.

Asymptotic Complexity of Mark-Sweep vs. Copying

- Conventional view:
  - Copying: O(live_data_size)
  - M/S:
    - Mark: O(live_data_size)
    - Sweep: O(heap_size)
    - Total: O(heap_size)
  - M/S more expensive (if heap_size >> live_data_size)
- Alternate view:
  - Sweep doesn't count; part of allocation.
  - M/S can avoid touching pointer-free data (strings, bitmaps)
  - M/S: O(pointer_containing_data)
  - Copying more expensive
    - (if pointer_containing_data << live_data_size)

Implementation details overview

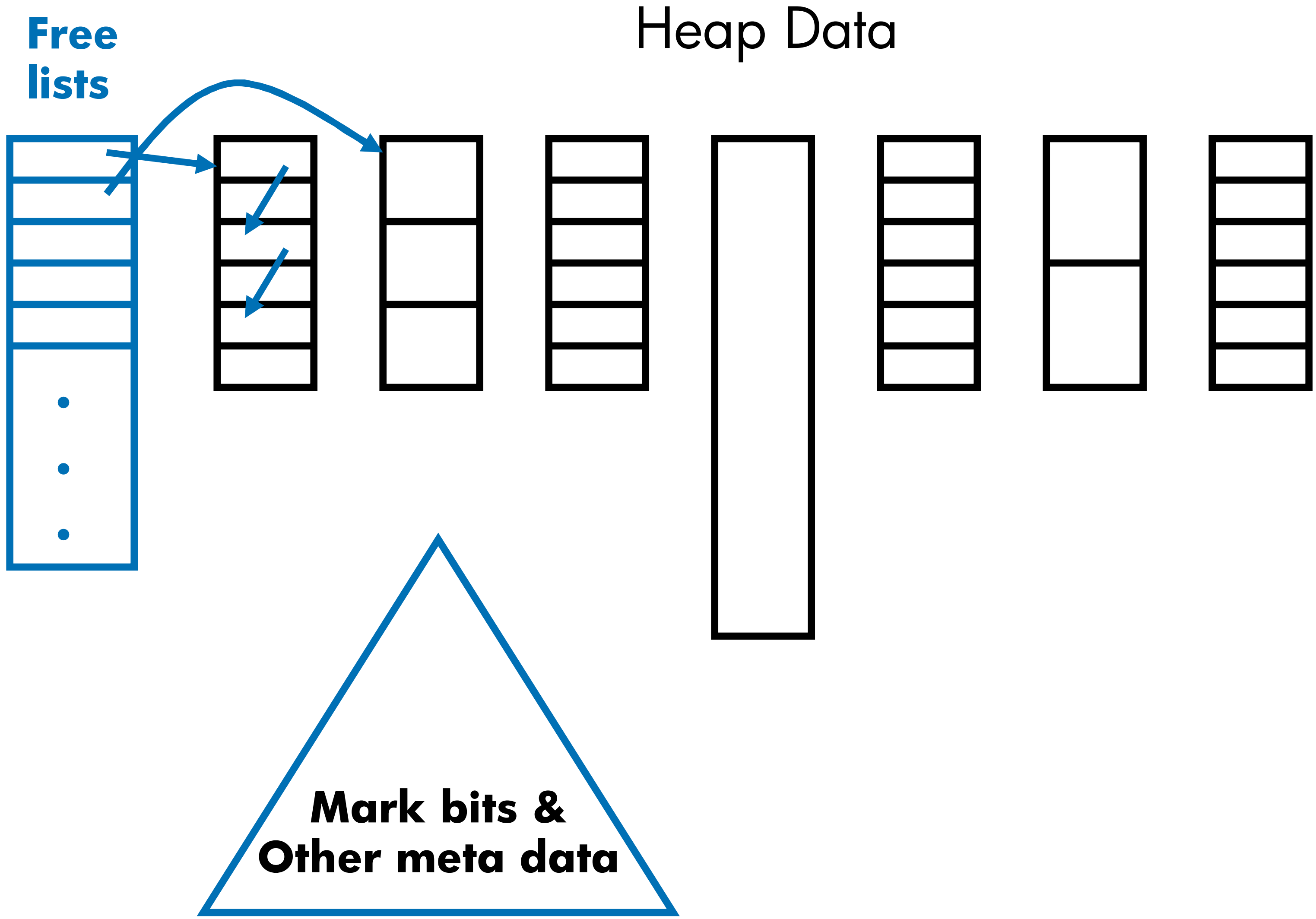
- General design issues:
  - The underlying allocator.
  - Pointer validity checks and mark bits.
  - Partial pointer location information.
  - Locating potential roots.
  - Mark algorithm and stack overflow.
  - Thread support.
- Enhancements:
  - Black-listing of “false pointers”
  - Incremental/Concurrent/Generational GC.
  - Parallel marking.
  - Thread-local allocation.
  - Finalization.
  - Debug support.

Blue items specific To Conservative GC.

Allocator design

- Segregate objects by size, pointer contents…
- Each "page" contains objects of a single size.
- Separate free lists for each small object size.
- Large object allocator for pages, large objects.
- Characteristics:
  - No per object space overhead (except mark bits)
  - Small object fragmentation overhead factor:
    - < #size classes = O(log(largest_sz/smallest_sz))
    - Asymptotically optimal (Robson 71)
  - Fast allocation.
  - Partial sweeps are possible.
  - Can avoid touching pointer-free pages.

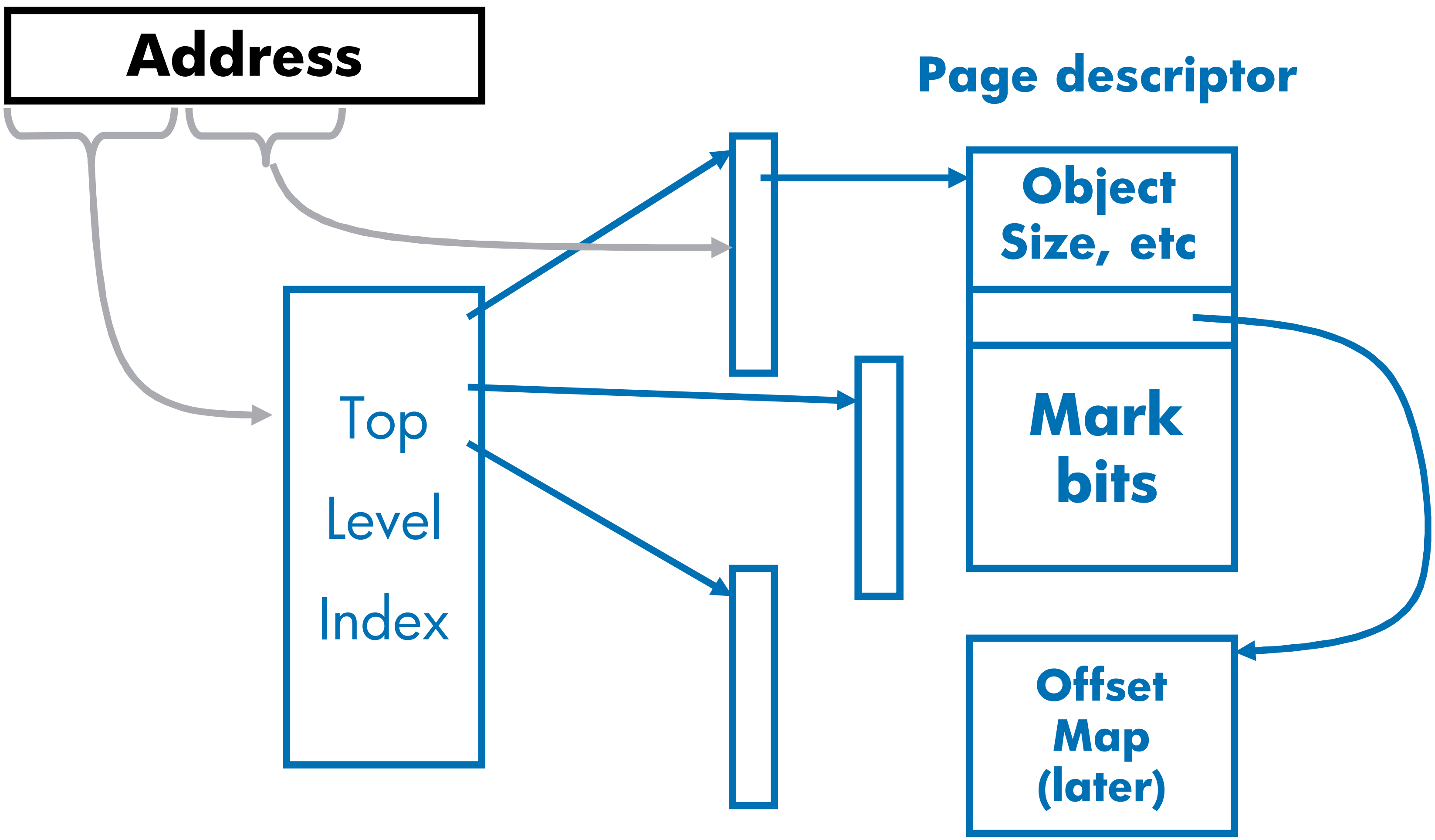
Heap layout



Meta-data

- Need mark bit for each object.
- Information for pointer validity & object size, etc.
- Support discontiguous heaps
- Options for mark bits:
  - In object:
    - Objects: must be aligned.
    - Stealing a bit may require a word.
  - At beginning of each block:
    - All mark bits are mapped to few cache lines.
    - Must touch pages with pointer-free objects.
  - In separate data structure.
    - More instructions for each access.
    - Pointer-free pages are not touched, fewer cache issues.

Meta-data lookup



Pointer validity check

- Get page descriptor. Valid heap page?
  - About three memory references.
    - Simple top level hashing scheme for 64-bit addresses.
  - Two with a small cache.
- If not first page of object, adjust.
- Valid offset in valid object?
  - Remainder computation on offset in page gives object start.
  - Remainder can be looked up in table of “valid offsets”.
  - Allows pointers to only certain offsets in object to be considered valid. Check is constant time.
  - Small constant number of memory references.

Partial pointer location (type) information.

- It's often easy to determine location of pointers in heap objects (e.g. gcj (Java), Mono (.Net)).
- Collector provides different allocation calls to communicate this.
- Objects are segregated both by size and "kind".
- Each kind has associated object descriptor:
  - First n fields are pointers.
  - 30- or 62-bit bitmap identifies pointer locations.
  - Client specified mark procedure.
  - Indirect: descriptor is in object or vtable.

Locating roots

- By default roots consist of:
  - Registers
  - Runtime stack(s)
  - Statically allocated data regions
    - Main program + dynamic libraries
- How do we get their contents/location?
  - Registers: abuse setjmp, __builtin_unwind_init, …
  - Runtime stack(s): you don't really want to know.
    - Need consistent caller-save reg. snapshot
  - Static data segments: you don't want to know that either.
  - *Very* platform dependent
    - But you only have to do it once per platform.

Basic mark algorithm

- Maintain explicit mark stack of pairs:

address	**descriptor**

- Initially:
  - For each individual root, push object.
  - For each root range, push range.
- Then repeatedly:
  - Pop (addr, descr) pair from stack.
  - For each possible pointer in memory described by pair:
    - Check pointer validity. If valid and unmarked:
    - Set mark bit for target. (Already have page descriptor.)
    - Push object address and descriptor (from page descriptor)

Marker refinements

- Tune as much as possible.
  - This is where the GC spends its time.
- It's the memory accesses that matter.
  - Prefetch object as we push its descriptor on stack.
  - May save 1/3 of mark time.
- Range check possible pointers for plausibility first.
  - Eliminates almost all non-pointers.
- Minor benefit from keeping cache of recently looked up block descriptors.
  - Probably more important for 64 bit platforms.
  - But uncached lookup is already fast.

The marker core (version pre-7.0)

1. Retrieve mark descriptor from stack.
2. (Possibly retrieve "indirect" descriptor from object.)
3. Look for pointers in object satisfying range check. Immediately prefetch at that address.
4. For each likely nested pointer, processing first one last:
   - Look up header in cache (2 memory references).
   - Get offset from beginning of block.
   - "Divide" by object size to get object start, and displacement in object.
   - If displacement is nonzero, check table for validity.
   - Check mark bit in header.
   - If not set, set it, get descriptor from block header, push entry on mark stack.

Marker performance: Why GC needs a fast multiplier.

- On toy benchmark, small objects., 1x1.4GHz Itanium
  - 500MB/sec (Peak mem. Bandwidth 6.4GB/sec.)
  - About 90 cycles/object. (L3 cache miss ~200cycles)
- About 260MB/sec, 180 cycles/object on a 2GHz Xeon.
- Cache misses matter a lot.
- Divisions are a problem.
  - Can easily multiply by reciprocal.
  - Integer multiply has around 15 cycles latency on IA64.
    - Similar on Pentium 4?
  - Very hard to hide latency.
  - Table lookup of remainder, mark bit per allocation granule (not object) wins (~20% on P4 Xeon).
- Could we do multiple header lookups & “divisions” at once to hide latency? Maybe …

What if the mark stack overflows?

- Likely as you approach memory limit.
- Programmers expect to be able to recover from running out-of-memory

  … although it is almost never 100% reliable, GC or not.
- We
  - Drop part of stack.
  - Set "overflowed" flag.
- If flag is set at end of mark phase:
  - Rescan heap. Look for marked → unmarked pointers.
  - Mark again from such targets.
  - Repeat if necessary.
  - Grow mark stack if possible.
- Never push large number of entries without setting a mark bit.
  - Ensures forward progress.

The "sweep phase"

- Sweep large objects and completely empty pages eagerly.
- Completely empty pages are easily detectable and surprisingly common.
  - Effectively coalesces *some* small objects very cheaply.
- Sweep small object pages when we encounter an empty free list.
- Separate pages can be swept in parallel.
- Empirically, execution time is almost always dominated by marker.

Thread support

- Uncontrolled concurrent mutation of data structures can cause objects to be overlooked by marker:

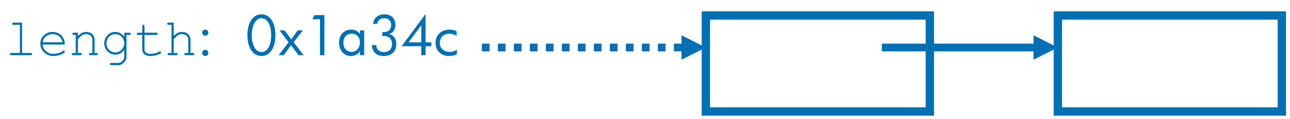
- Results in reclaimed reachable objects.

Thread support (2)

- We stop threads during critical GC phases.
  - Unlike most GCs, threads can be stopped anywhere.
- On most platforms, we send each thread a signal, with handshake in handler.
  - Ensures that thread is stopped.
  - Pushes register contents onto the (GC-visible) stack.
- Typically requires that thread creation calls be intercepted by GC.
  - GC substitutes its own thread start routine.
  - Keeps track of threads, shadowing thread library.

Enhancement 1: Black-listing

- Conservative pointer-finding can cause memory retention:



- In many cases, this is avoidable.
  - If we see an address near future heap growth:

  - Don't allocate at location 0x1a34c.
  - We track pages with bogus pointers to them.
    - Marker updates list.
    - Allocate at most small pointer-free objects there.

Black-listing (contd.)

- Can be substantial improvement, especially with large root sets containing random, but static data.
- Only dynamic data can cause retention.
  - But dynamically created data is also more likely to disappear later.
- Usually we see good results with conservative pointer finding, minimal layout information and
  - 32 bit address space, heaps up to a few 100MB, or
  - 64-bit address space.

Optional enhancements

- Remaining enhancements are (or were) implemented and available, but not all combinable.

Generational, Incremental, Mostly Concurrent GC

- Observation:
  - Running marker concurrently establishes invariant:
    - Pointers from marked objects or roots either
      - point to marked objects, or
      - were modified since object was marked.
  - Such a concurrent mark phase can be "fixed" if we can
    - Identify possibly modified objects (and roots)
    - Mark again from modified objects.
  - Most generational collectors track modifications with a compiler introduced "write barrier".
  - We use the VM system, e.g.
    - Write protect pages (e.g. mprotect for Linux)
    - Catch protection faults (e.g. SIGSEGV)
  - Free if allocation is rare, but otherwise not ideal.

- Mostly concurrent GC:
  - Run concurrent marker once.
  - Run fixup marker zero or more times concurrently, preserving invariant, reducing # dirty objects.
  - Run fixup marker with threads stopped once.
  - Works, reduces pause times, used in other systems.
  - Scheduling tricky, requires threads.
- Incremental GC:
  - Do a little "concurrent" marking during some allocations.
  - Amount of marking proportional to allocation.
  - Same pause time benefit, no throughput benefit.
- Generational GC:
  - Leave mark bits set after "full GC", but track dirty pages.
  - "Fixup GC" is minor GC.

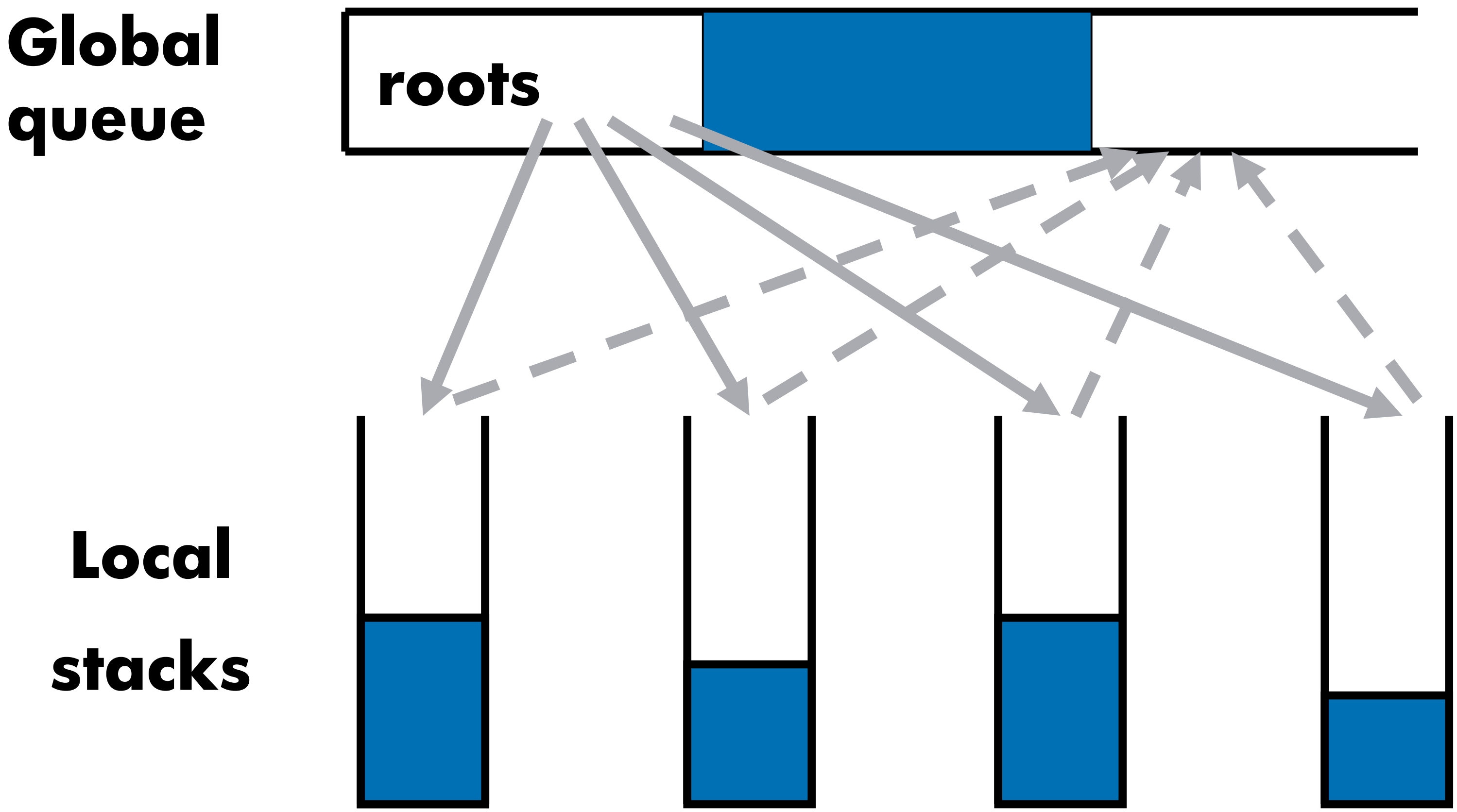
Parallel marking & processor scalability

- As client parallelism increases, eventually we spend all time in sequential part of GC.
- Sweeping is done a page at a time & can be parallelized. What about marking?
- Marking is also quite parallelizable.
- First, and most thoroughly, done by Endo, Taura, and Yonezawa (SC97, 64 processor machine).
- Our distribution contains simpler version …

Parallel marking

- For *n* processors, we have *n*-1 threads waiting to help with next GC.
- Global mark stack becomes queue.
- Each marker thread regularly:
  - Removes a few entries from queue tail.
  - Marks from those using a local mark stack.
- Mark bits are shared between marker threads.
  - Either use mark bytes, or atomic-compare-and-swap.
    - Mark bytes usually win. (1/8 - 1/16 memory overhead.)
  - Work may be duplicated but rarely is.
- Load balance by returning part of local stack to top of queue
  - When local mark stack overflows.
  - When it notices empty global queue.
- Seems to scale adequately, at least for small SMPs.
  - Limit appears to be bus bandwidth.

Parallel marking data structure



Thread-local allocation buffers

- Malloc/free implementations acquire and release a lock twice per object allocation/deallocation:
  - Once per allocation.
  - Once per deallocation.
- Garbage collectors avoid per-deallocation lock.
- We can also avoid per-allocation lock!
- Use per-thread allocation caches.
  - Each thread allocates a “bunch” of memory.
    - Single lock acquisition.
  - Dividing it up doesn’t require a lock.
  - Easy with linear allocation, but also possible here.

Thread-local allocation details

- Each thread has array of small object free-list headers.
- Each header contains either:
  - Count of allocated objects of that size.
  - Pointer to local free list.
- To allocate:
  - For small counts, increment count, allocate from global free list.
  - For count at threshold, or empty free-list, get a page of objects.
  - For nonempty free-list, allocate from local free-list.

Finalization

- Finalizable objects are added to a growable hash table.
- After each GC, we walk this hash table two or three times:
  - Mark all objects reachable from objects in the table.
    - But not the objects in the table themselves.
    - Table entries contain the procedures to do this marking to handle variants like Java.
  - Enqueue still unmarked objects in the table for finalization, and possibly mark them.
  - Possibly mark objects reachable from finalizable objects. (Java style finalization only.)
- Process finalizable objects, preferably in separate thread, once allocation lock is released. (See POPL 2003 paper.)
- Weak pointers ("disappearing links") are handled similarly.

Finalization (quick observations)

- Finalization is moderately expensive.
  - Extra space overhead.
  - Tracing cost is significantly higher, even with Java-style finalization (factor of 5?)
- Clients should avoid registering unnecessary finalizers. (JVMs can do this statically.)
- Finalizers do not affect performance of the rest of the GC.
- Finalizers *must* introduce concurrency (even if we had a simple reference counting collector). *There is no such thing as deterministic finalization for heap objects.*
  - Collector runs them in **`GC_malloc`** by default. *This is a bug except in very simple cases.* Use **`GC_finalizer_notifier`**.
  - Concurrency is tricky. Be careful.

Debugging support

- Debug allocators “wrap” each object with extra information:
  - Source file, line number of allocation site.
  - Possibly a stack trace for allocation site.
  - Space for a back pointer. (Should be elsewhere…)
  - Requested object size.
  - Magic (address dependent) numbers before and after object.
- Can mostly tolerate a mixture of wrapped and unwrapped objects.
  - Relies on “magic numbers”.
  - May lead to extra error reports.

Debugging facilities

- GC can check for overwrite errors.
- Various error checks in GC_debug routines.
- Can be configured for leak detection.
- Can tell you whether a single misidentified pointer might result in unbounded space leak (See POPL 2002)
- Can give back-traces of random heap samples (requires different build flags):

```
****Chose address 0x81ac567 in object
0x81ac568 (trace_test.c:13, sz=8, PTRFREE)

Reachable via 0 levels of pointers from offset 4 in object:
0x8192090 (trace_test.c:11, sz=8, NORMAL)

Reachable via 1 levels of pointers from offset 0 in object:
0x81920b8 (trace_test.c:11, sz=8, NORMAL)

Reachable via 2 levels of pointers from root at 0x8055bd4
```

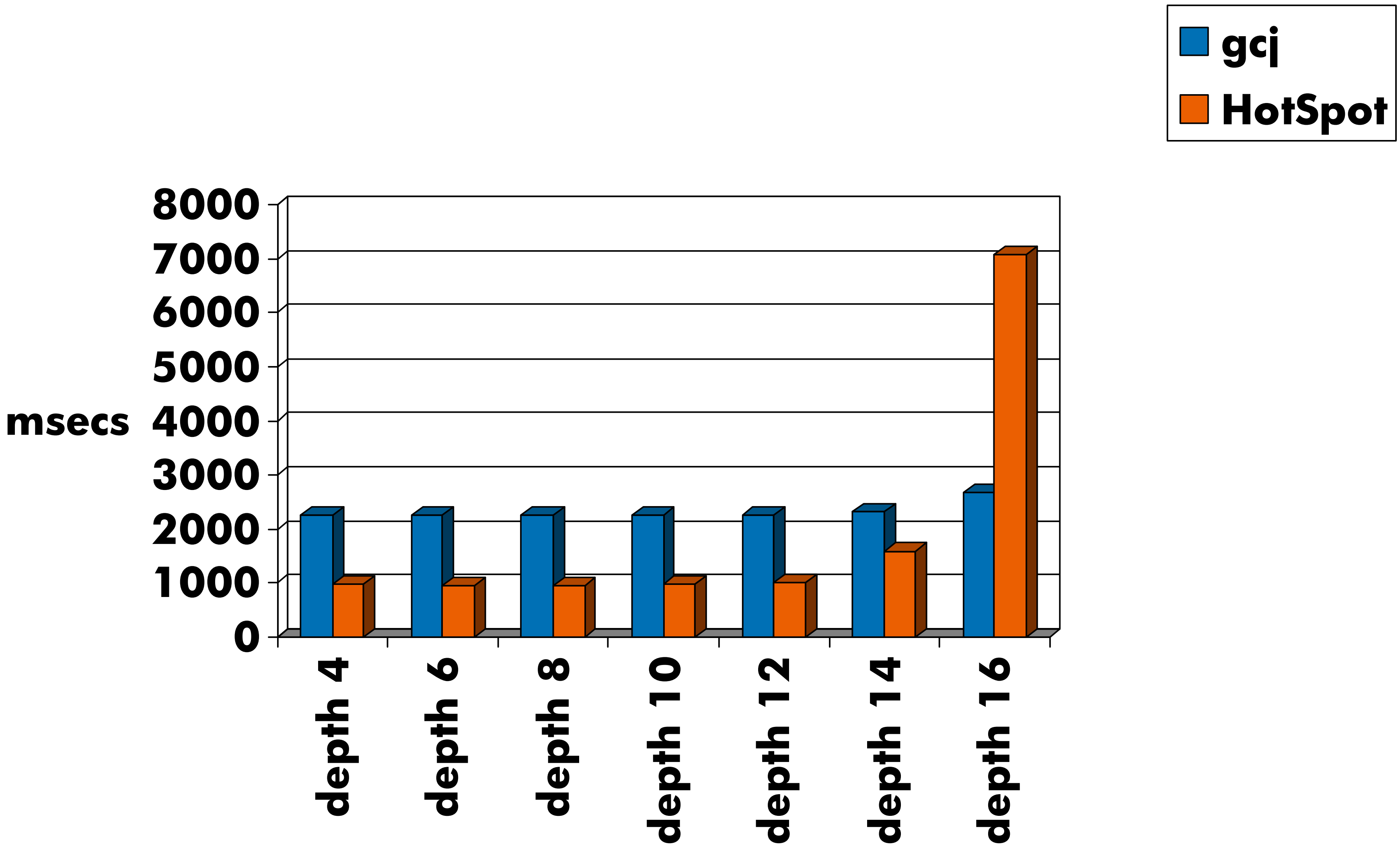
Current state

- Easily available (google "garbage collector")
- Supports Linux, Unix variants, Windows, MacOSX,…
- Used in a variety of C/C++ systems
  - w3m, vesta, …
  - High end Xerox printers.
  - Sometimes as leak detector (e.g Mozilla).
  - Usually with little type information.
- Used in many language runtimes:
  - Gcj (gcc), Mono, Bigloo Scheme
  - Usually with heap type information.
  - Information on static data (e.g. 4.5MB for gcj) would be easy and useful.
- Current version 6.3; 6.4 should appear shortly.
- Stay tuned for 7.0alpha1 (cleaner code base, …)

Performance characteristics

- We use GCBench here.
  - More of a sanity check than a benchmark.
  - Allocates complete binary trees of varying depths.
    - Depth n ➔ 2^n-1nodes of 2 pointers + 2 ints
  - Translated to multiple source languages.
  - Can see effect of object lifetime.
  - About as realistic as any toy benchmark (not very).

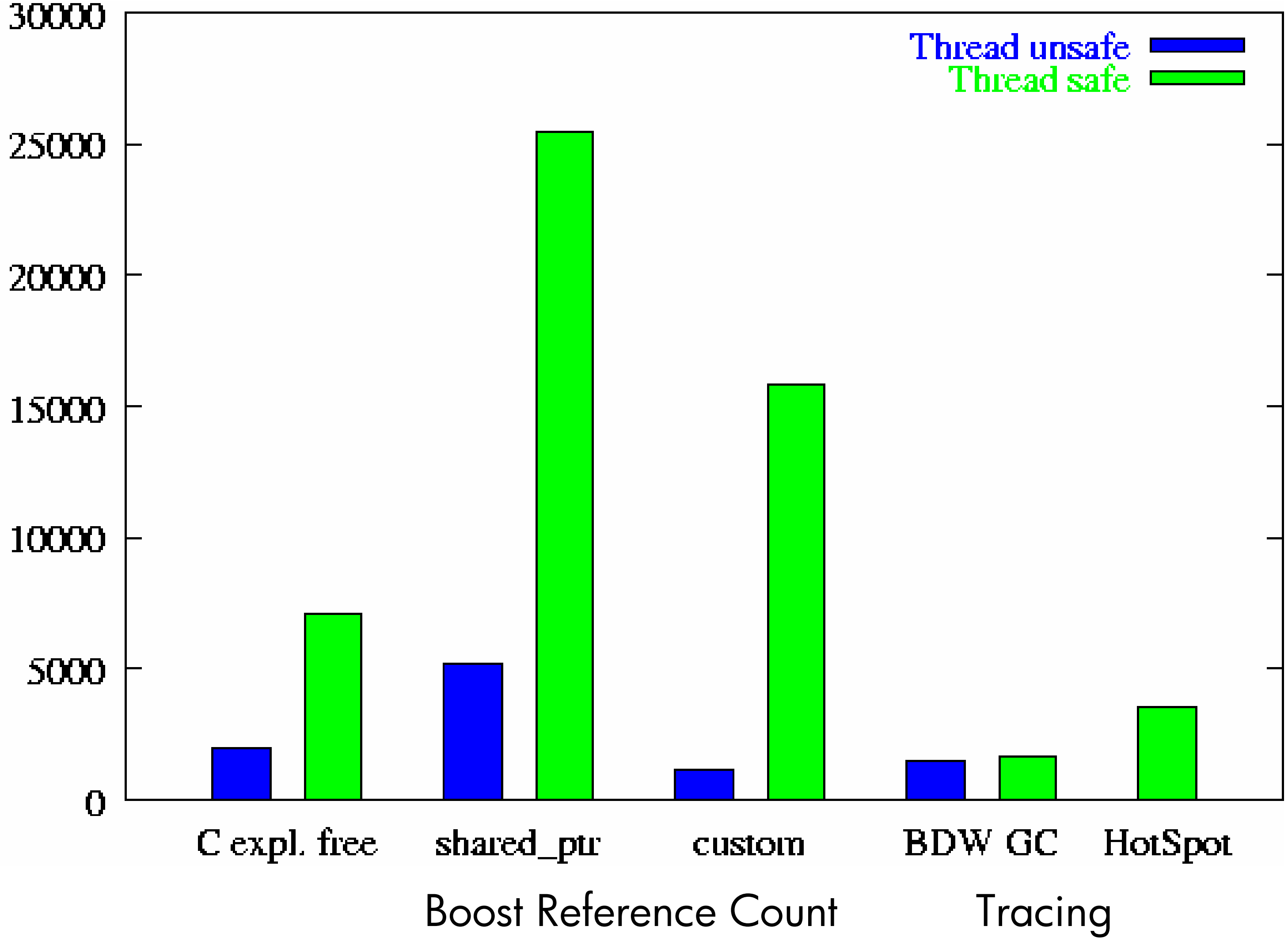
GCBench vs HotSpot 1.4.2 client



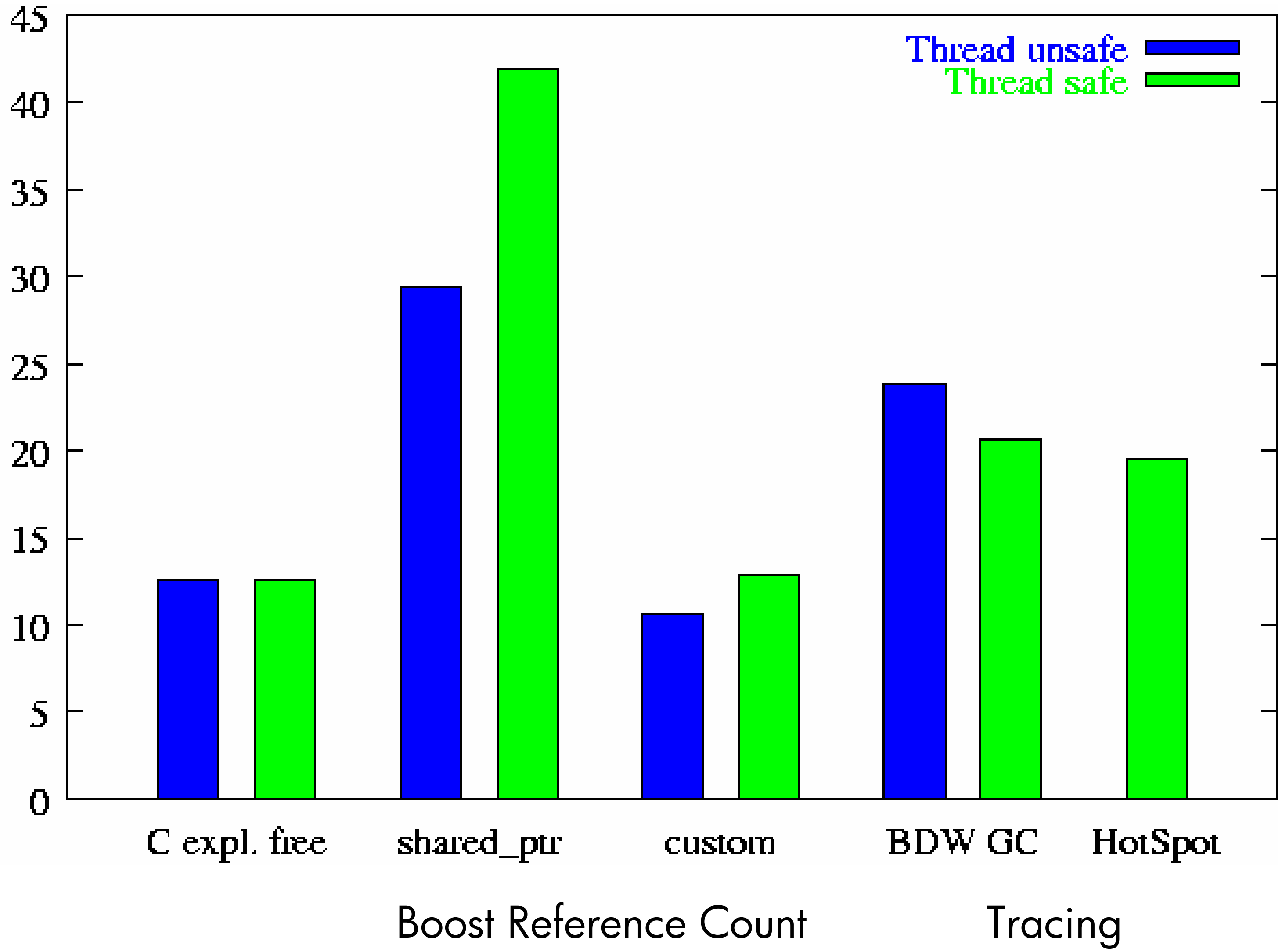
C/C++ GCBench Comparison

- Compare:
  - C with malloc/free
    - "Pause" is tree deallocation time (predictable).
  - Boost classic reference counting (simple and tuned version)
    - "Pause" is recursive deallocation time during assignment (unpredictable).
  - Boost versions use C++ benchmark.
  - Expl. free and BDW GC use C version.
  - HotSpot uses Java version.

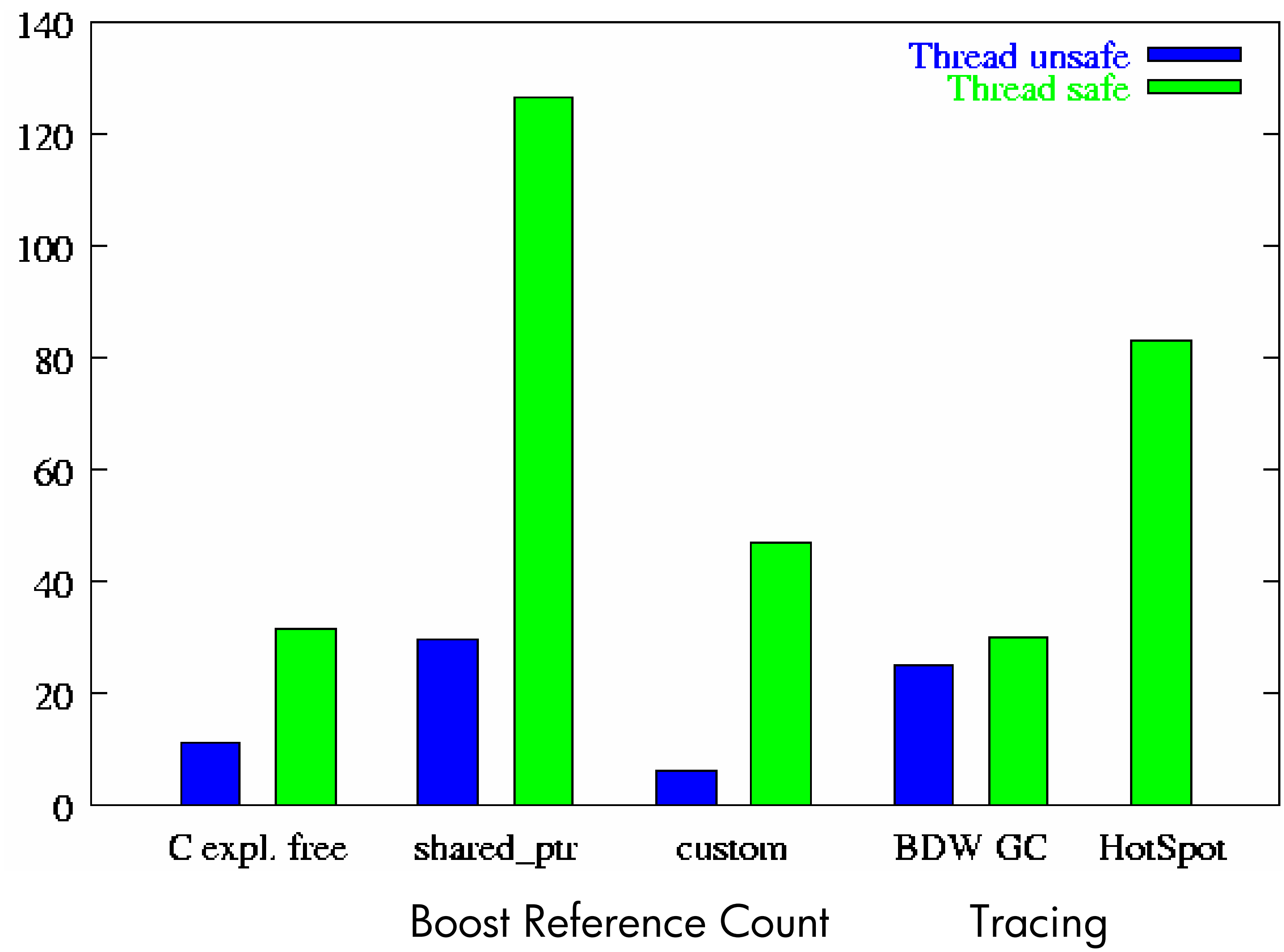
Execution time (msecs, 2GHz Xeon) vs. alternatives



Max. space usage (MB) vs. others



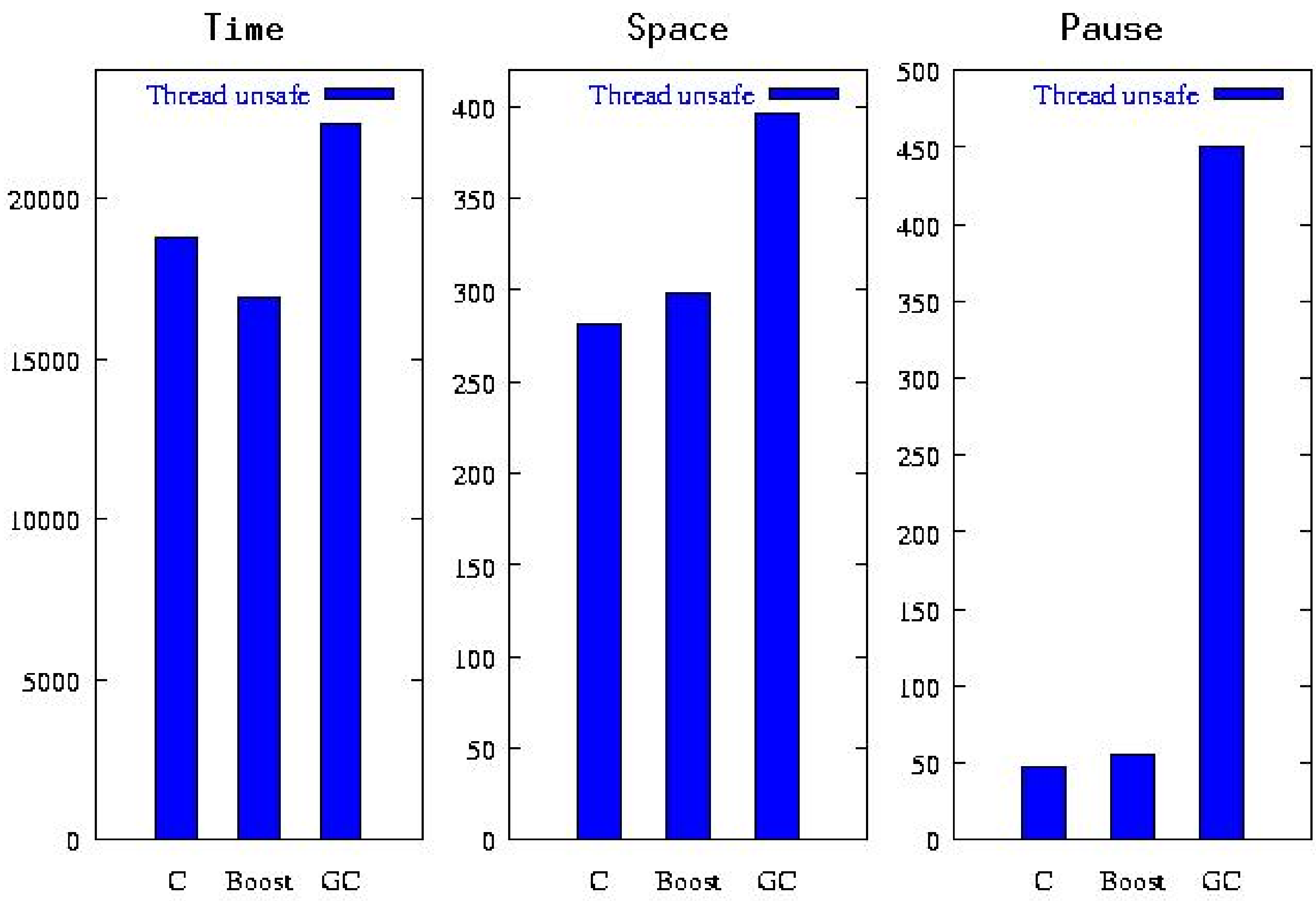
Max pause time (msecs) vs. others



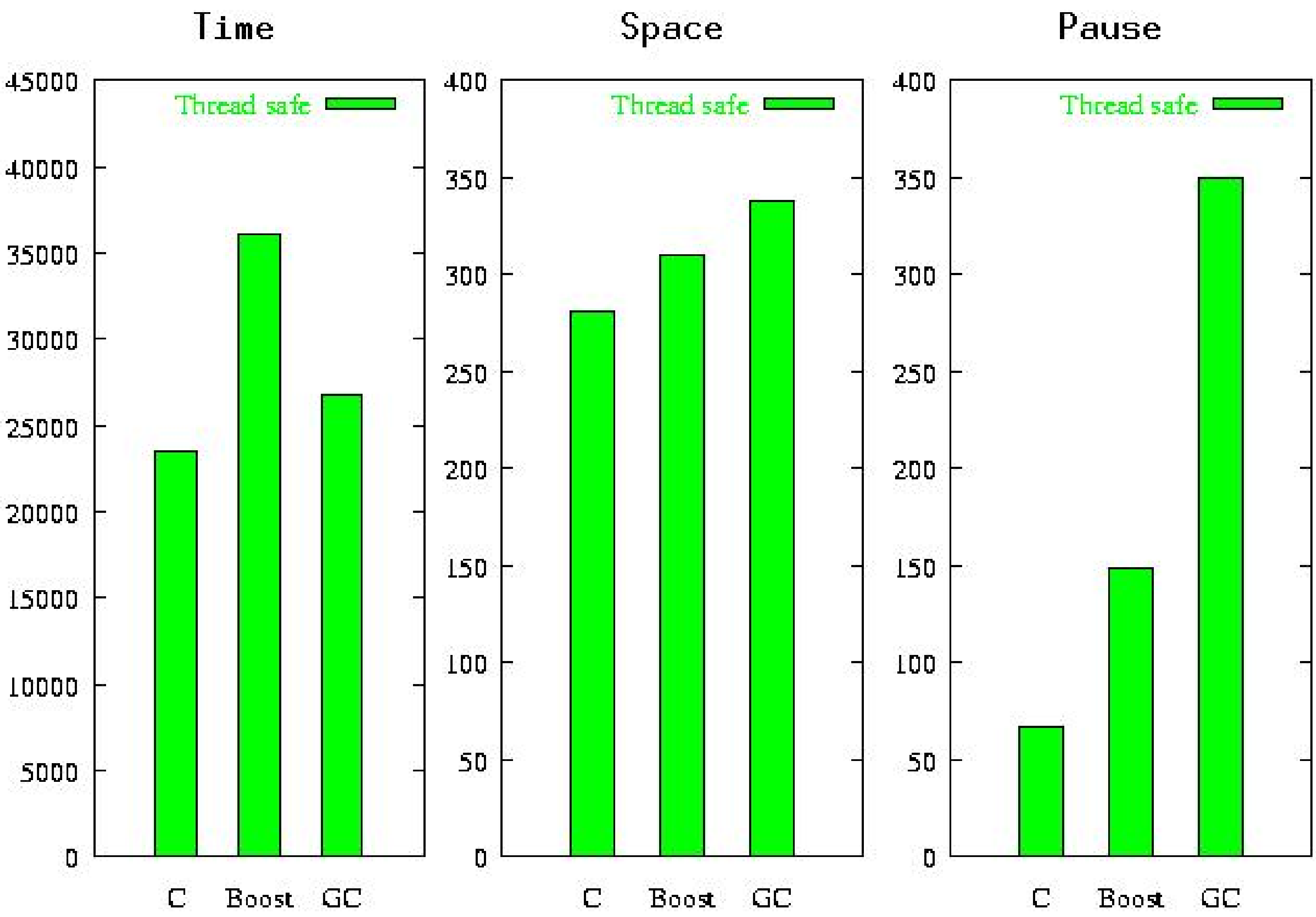
But:

- GCBench uses small objects.
- Allocation + GC cost is proportional to object size.
- Redo experiment with 128 extra null pointer per node.

Large Objects (msecs, MB, 2GHz Xeon)



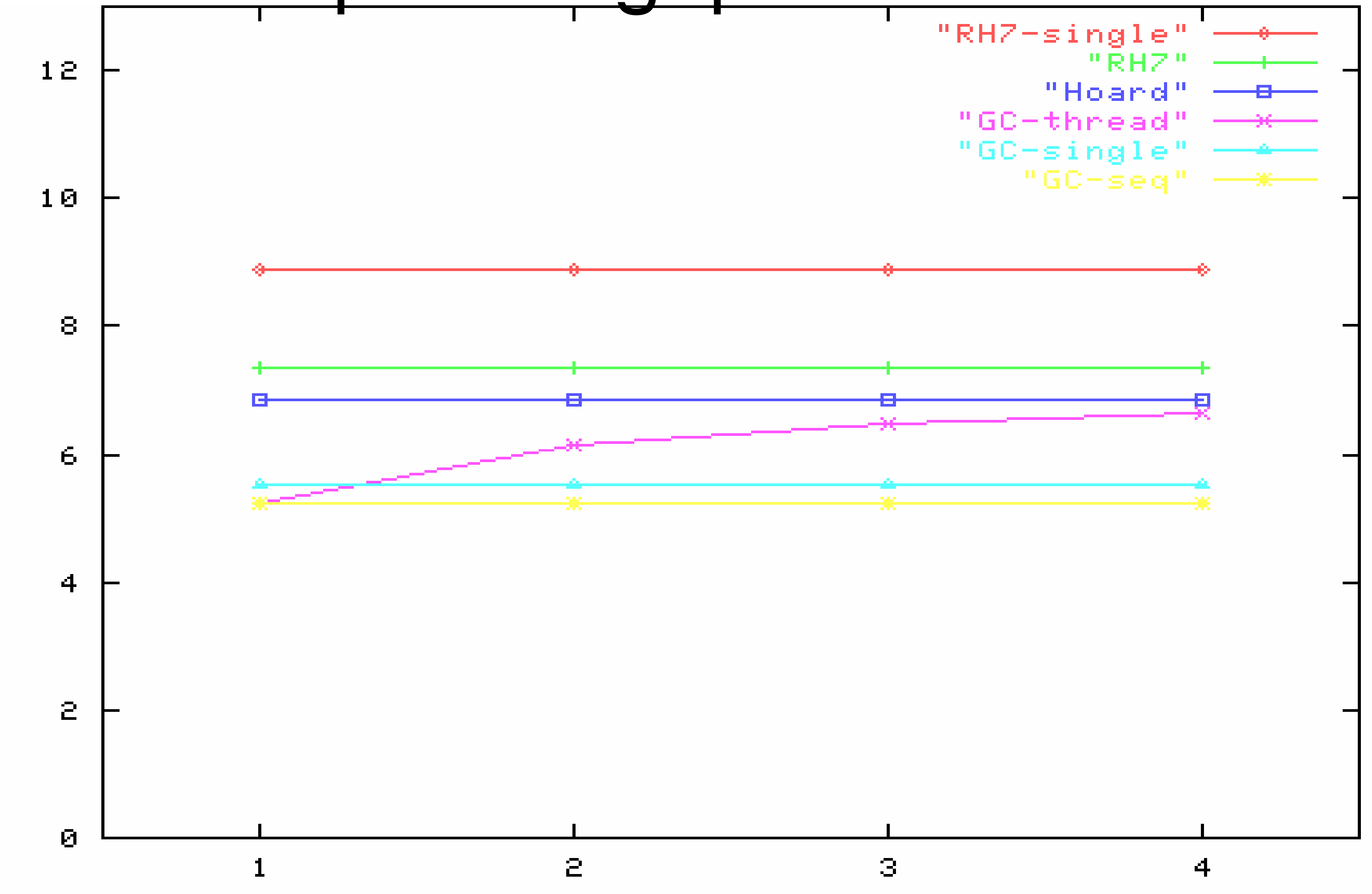
Large Objects (thread-safe)



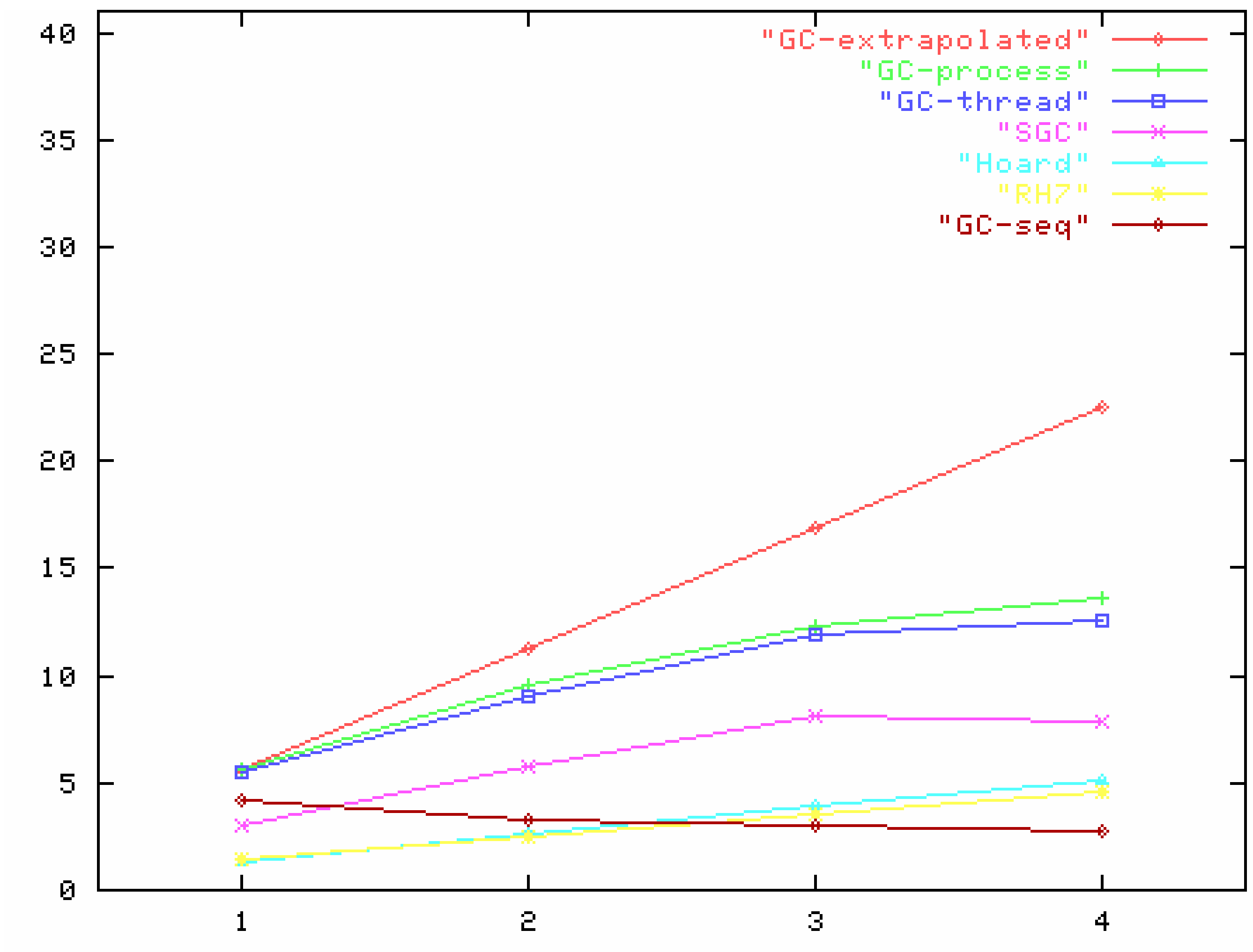
Some older measurements on malloc benchmarks

- These are a bit obsolete, things have probably improved, but …
- Measured on 4xPPro (which was obsolete then).

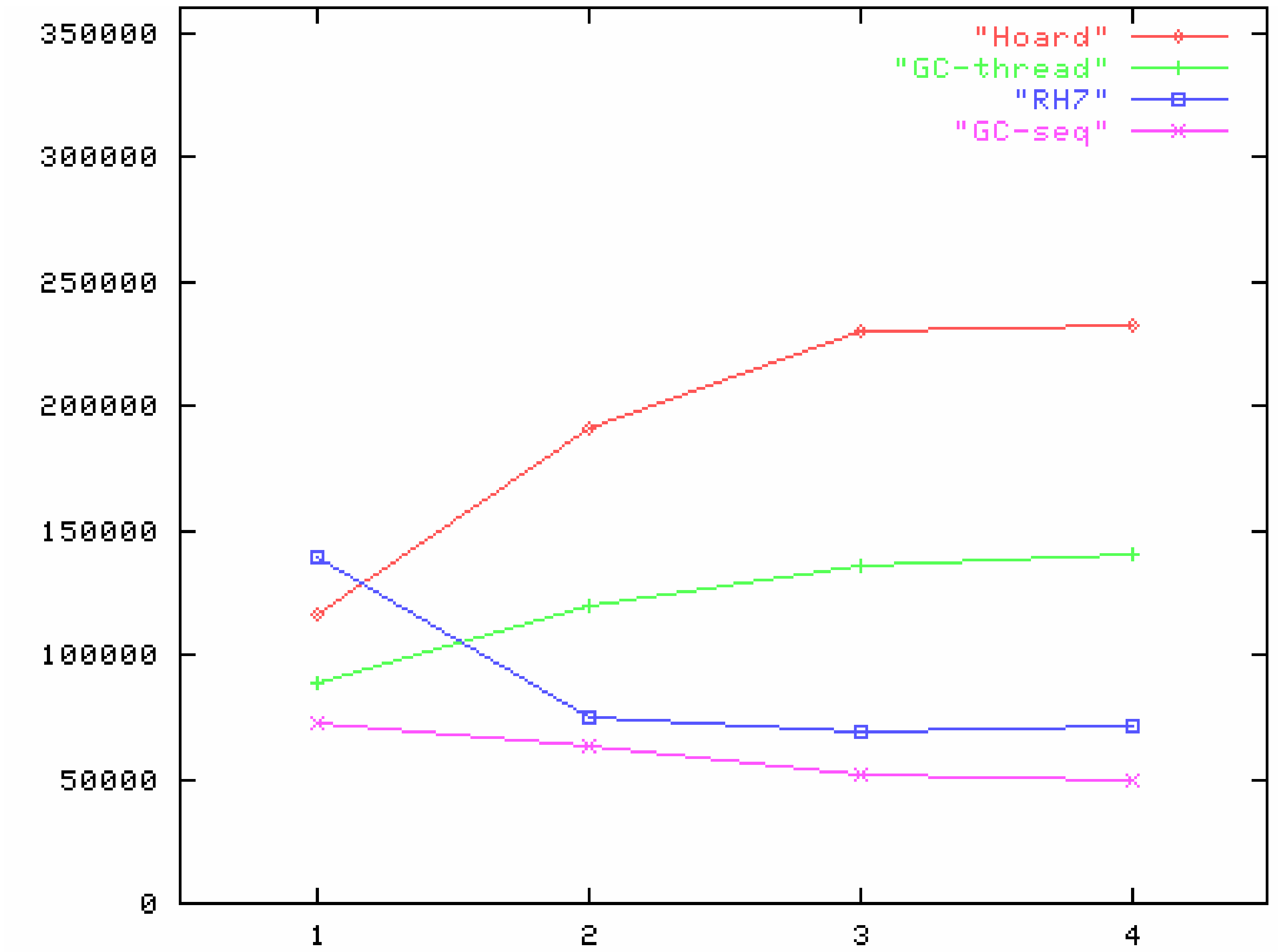
Ghostscript throughput



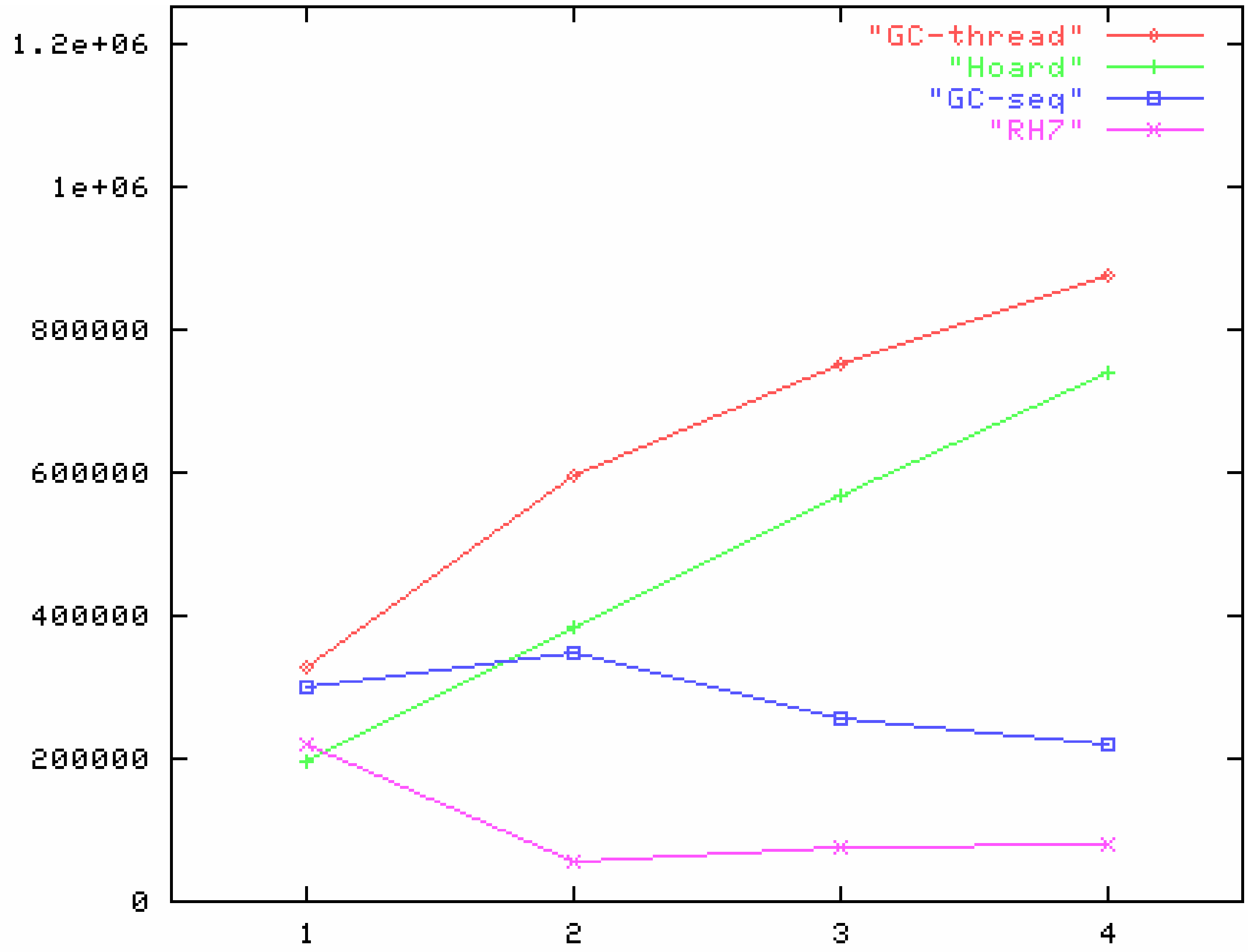
MT_GCBench2 throughput



Larson (slightly mod.) benchmark throughput



Larson-small throughput



Other experiences

- Generally works quite well for small (< 100MB live data) clients or on 64-bit machines.
  - Sometimes needs a bit pointer location information for frequently occurring heap objects. Usually GC_MALLOC_ATOMIC is sufficient for C code.
- Some successful uses with much larger heaps.
- Some problems with 500MB heaps on 32-bit machines.
- Large arrays (> about 1MB) sometimes problematic.
- Fragmentation cost (for heaps > a few MB) is typically less than a factor of 2.
  - Fragmentation essentially never an issue for small objects.
  - Whole block coalescing is important.
- I haven't seen much of a problem with long running apps. (Vesta, Xerox printers).
- Stationary objects allow one word object headers in gcj.

Space overhead of conservative GC

- Clever empirical study:
  - Hirzel, Diwan, Henkel, “On the Usefulness of Type and Liveness Accuracy for Garbage Collection”, TOPLAS 24, 6, November 2002.
    - Liveness information is usually more important than type information, especially on 64-bit platforms.
    - Up to 62% space overhead.
- More theoretical study:
  - Boehm, “Bounding Space Usage of Conservative Garbage Collectors”, POPL 2002.

Conclusions

- Collector is still a useful tool for
  - Avoiding manual memory management issues in C/C++.
  - Quickly building language runtimes, especially, but not only, for research systems.
  - Some GC research. (One underlying algorithm, mult. languages.)
- Performance is competitive with malloc/free.
  - Usually wins for threads + small objects.
- Tracing performance is very close to best commercial JVMs.
  - See also Smith and Morrisett, ISMM 98.
  - Currently does less well when there is a large benefit from generational GC. (But see OOPSLA 2003 paper by Barabash et al.)
- There may be a cache cost to free list allocation.
  - See work by Blackburn, Cheng, and McKinley.
  - But I don't think we fully understand this yet …