Memory Allocation: Either Love It or Hate It (or just think it's okay)

Andrei Alexandrescu

Memory Allocation

- Fundamental part of most systems since 1960
- Frequently at the top of time histograms
- Hard to improve on
- Two views:
  - Solved
  - Insoluble

Overview

Lies, Damn Lies. . .

Custom Allocators

Policy-based Implementation

Conclusions

Lies, Damn Lies. . .

Memory Allocation

- *Need:* randomly allocate and free chunks of various sizes
- *Fact:* most applications have highly regular allocation/deallocation patterns
- *Challenge:* Such patterns notoriously escape standard statistical analysis

Engineering Challenges

- Highly competitive:
  - General-purpose allocators: 100 cycles/alloc
  - Specialized allocators: $<$12 cycles/alloc
- Hard to balance capacity with generalization
  - “Too general:” mediocre for all patterns
  - “Overspecialized:” numerous degenerate situations
- Very hard to modularize/componentize

Capacity and Generalization

- Generalization: Figure out patterns from seen data
  - Too much: “Rectangular piece of paper? That’s a $100 bill”
- Capacity: Ability to allocate new categories for data
  - Too much: “#L26118670? It’s a fake; all $100 bills I’ve seen had other serial numbers”
- They are competitive with one another
- How to strike the right balance?

Odysseus, Beware of Sirens

- *Myth:* application efficiency can be largely increased by hand-writing a custom memory allocator
- *Fact:* on six out of eight popular benchmarks, replacing *back* the custom allocator with a general-purpose allocator generally *improves* speed and memory consumption [1]
- *Fact:* simplest custom allocators tuned for special situations are the only ones providing a measurable gain

The Windows XP Allocator

- Best-fit allocator
- 127 free lists for 8, 16, 24, 32. . . , 1024 bytes
- $>$1024 bytes from sorted linked list
  - Sacrifices speed for size accuracy

Windows XP-Specific Heap

- Very rich interface
  - Multiple heaps
  - Region semantics
  - Individual Object Deletion
- Distinctly unimpressive performance

The Doug Lea Allocator

- Approximate best-fit allocator
- Quicklists for $<$64-bytes allocations
- Coalesce and split blocks on medium-sized allocations ($<$128KB)
- Large blocks allocated with `mmap`

- The basis of the Linux allocator
- Honed throughout millennia

Benchmarks

#	Name	Description
1.	`197.parser`	English parser
2.	`boxed-sim`	Balls-in-box simulator
3.	`C-breeze`	C-to-C optimizing compiler
4.	`175.vpr`	FPGA placement and routing
5.	`176.gcc`	Optimizing C compiler
6.	`apache`	Web server
7.	`lcc`	Retargetable C compiler
8.	`muddle`	MUD compiler/interpreter

Results

- Pretty much everybody beats the Windows XP allocator in speed
  - Space consumed not measurable
- Only `lcc` and `muddle` outperformed Doug Lea's allocator (20-47% in speed, 25% in space)
- Rest: within 10% in speed, but consuming a hefty 33% extra space

Custom Allocators

Temporal Allocation Patterns

- *Ramps:* Slowly chews memory (logs, editors, file processing, debuggers, leaks)
- *Plateaus:* Dynamic equilibrium—the application statistically allocates as much as it deallocates
- *Peaks:* The application suddenly allocates much memory and then deallocates most of it (compilers, linkers, parsers, other staged tools)

Size Distribution

- *Pareto-distributed (80/20):* Most allocations are of a small set of sizes
  - Focus on allocating specific sizes
- *Long-tail:* Distribution does not have a strong bias
  - Focus on allocating random sizes

Useful Custom Allocators

- Mostly there are two effective custom allocators
  - Regions (a.k.a. arenas, pools)
  - Free Lists
- Both only work for particular allocation patterns
- Both may do very, very bad on other allocation patterns
- Precisely because everything that works for *general* allocation patterns has been already incorporated in today's allocators!

Regions

- Allocate memory blocks of arbitrary sizes
- Deallocate all at once
- Useful for batch, transaction-oriented applications
  - Web servers
  - Compilers
  - Databases
- API:

```
void * allocate(size_t size);
void freeAll();
```

How Regions Work

- Initially allocate a large block of memory
- Each allocation is a bump of a pointer
  - Plus checking for overrun
- There's no deallocation
- Terminating the region deallocates the large block

- Risk: pointers into the region from global heap or longer-lived regions

Free Lists

- Useful for Pareto-distributed block sizes
  - Stock data
  - String processing apps
  - Scientific apps
- API:

```
void * allocate(size_t size);
void deallocate(void * p);
size_t allocatedSize(void * p);
```

How Free Lists Work

- Allocate memory blocks of arbitrary sizes
- Remember blocks of specific size in a list
- Subsequent allocations of that size are $O(1)$
- The free list can live inside the freed blocks

- Advantage: superfast for one special size
- Risk: fixed size, no coallescing, no reallocation, extra space consumed

Policy-based Implementation

General Interface

```
template <class BaseAllocator>
struct Allocator : private BaseAllocator {
   Allocator();
   ~Allocator();
   void * allocate(size_t size);
   void deallocate(void * p);
   size_t allocatedSize(void * p);
}
```

- "For adoption" structure $\Rightarrow$ efficient layering
- `allocatedSize` is optional
- `allocatedSize` $\geq$ requested

Top Allocator

```
struct Mallocator {
   static void * allocate(size_t size) {
      return malloc(size);
   }
   void deallocate(void * p) {
      free(p);
   }
```

...continued

```
#if defined(WIN32)
   size_t allocatedSize(void * p) {
      return _msize(p);
   }
#elif defined(GNU)
   size_t allocatedSize(void * p) {
      return malloc_usable_size(p);
   }
#endif
};
```

Supporting the Unsupported

```
template <class B>
struct SizedAlloc : private B {
   void * allocate(size_t s) {
      size_t * pS = static_cast<size_t*>(
         B::allocate(s + sizeof(size_t)));
      return *pS = s, pS + 1;
   }
   void deallocate(void* p) {
      B::deallocate(
         static_cast<size_t*>(p) - 1);
   }
```

. . . continued

```
    ...
    size_t allocatedSize(void * p) {
       return (static_cast<size_t*>(p))[-1];
    }
};
```

- Remark: Add portable size computation over any allocator, not only `malloc`
  - We'll make use of that in a few slides

In-Situ Regions

```
template <size_t S> struct InSituRegion {
    void * allocate(size_t s) {
        s = (s + M) & ~M;
        if (s > S || p > buf + S - s) return 0;
        void * r = p;
        p += s;
        return r;
    }
    ...
    enum { A=2, M=(1<<A)-1}; // align
    private: unsigned char buf[S], *p;
};
```

In-Situ Regions

```
template <size_t S> struct InSituRegion {
    ...
    void freeAll() {
        p = buf;
    }
    ...
};
```

- No `allocatedSize`
- Going out of scope entails an unchecked `freeAll`

Heap Regions

```
template <class B>
struct HeapRegion : private B {
   HeapRegion(size_t s = 1024 * 128) {
      buf = p = enforce(B::allocate(s));
      limit = buf + s;
   }
   ~HeapRegion() {
      B::deallocate(buf);
   }
   ...
   private: unsigned char *buf, *limit, *p;
};
```

Lists of Allocators

- How to make regions no-fail?
- Keep a list of regions
- Allocate from head of the list
- Add a new element when head full

```
template <class A> struct AllocList {
    ...
    private: slist<A> allocs;
};
```

- Works both with in-situ and heap allocators

Policy-Based Design: Yum (I)

```
typedef AllocList<InSituRegion<1024 * 128> >
   ScalableRegion;
```

Free Lists

```
template <size_t S, class B>
struct ExactFreeList : private B {
   void * allocate(size_t s) {
      if (s != S || !list)
         return B::allocate(s);
      void * r = list;
      list = list->next;
      return r;
   }
   ...
```

Free Lists (cont'd)

```
    ...
    void deallocate(void * p) {
        if (allocatedSize(p) != S)
            return B::deallocate(p);
        list * pL = static_cast<List*>(p);
        pL->next_ = list_;
        list_= pL;
    }
    using B::allocatedSize;
private:
    struct List { List * next; } list;
};
```

Aside

- `using`: the enemy of generic programmers

```
struct A {
   ... no declaration of allocatedSize ...
};
template <class B>
struct C : private B {
   using B::allocatedSize;
}
C<A> object; // error!
```

Aside (cont'd)

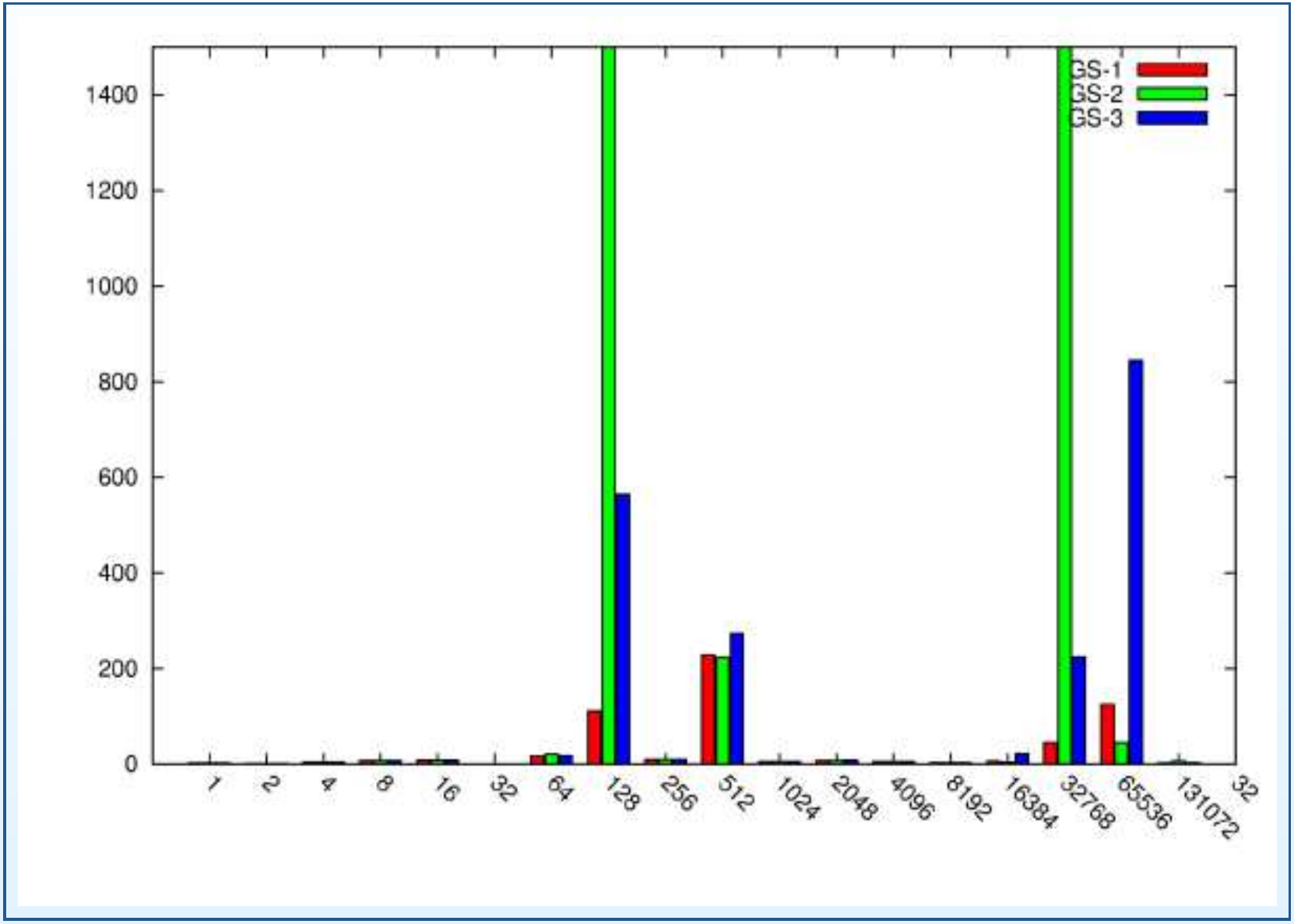
- However, this does work:

```
struct A {
   ... no declaration of allocatedSize ...
};
template <class B>
struct C : private B {
   size_t allocatedSize() {
      return B::allocatedSize();
   }
}
C<A> object; // fine
```

Rounded Free Lists

```
template <class B>
struct RoundedFreeList : private B {
   RoundedFreeList(size_t preferred) {
      list = B::allocate(preferred);
      list->next = 0;
      S = allocSize(list);
   }
   ...
   private: size_t S;
};
```

Example: Ghostscript [3]



Policy-Based Design: Yum (II)

```
typedef ExactFreeList<65536,
   ExactFreeList<128,
      ExactFreeList<512, Mallocator>
   >
>
GhostscriptAllocator;
```

Reaps

- Regions can't deallocate (`deallocate` is a no-op)
- Free lists intercept `deallocate` calls to remember deallocated blocks of a specific size

- . . . Wait a **minute**!

Policy-Based Design: Yum (III)

```
typedef ExactFreeList<128,
   ExactFreeList<512,
      SizedAlloc<InSituRegion<128 * 1024> >
   >
>
GhostscriptRegionAllocator;
```

- Allocate from the region (fast)
- Part of what's deallocated gets under free list control (fast)
- Combine benefits of regions and free lists!

General Reaps

- Berger's insight [1]:
  - Any pointer freed from a reap can be passed to another memory manager
  - Best-fit, free lists, quick lists. . .
  - Complication: some of these might need to add metadata to each allocation
- Modular implementation in HeapLayers [2]
- Yet competitive with hand-written special allocators

Conclusions

To Conclude

- To speed up applications:

```
if (profilerClearlyShowsAllocBottleneck()) {
   if (needToUseRegions()) {
      useReaps();
   } else {
      assert(!useWindowsAllocator());
      useDlMalloc();
   }
}
```

References

[1] E. Belkin, B. Zorn, and K. McKinley. Reconsidering Custom Memory Allocation. *OOPSLA*, 2002.

[2] Emery D. Berger et al. Composing High-Performance Memory Allocators. In *SIGPLAN*, pages 114–124, 2001.

[3] Miguel Masmano. Histograms Library. `http://tinyurl.com/2n5nds`.