Reconsidering Custom Memory Allocation

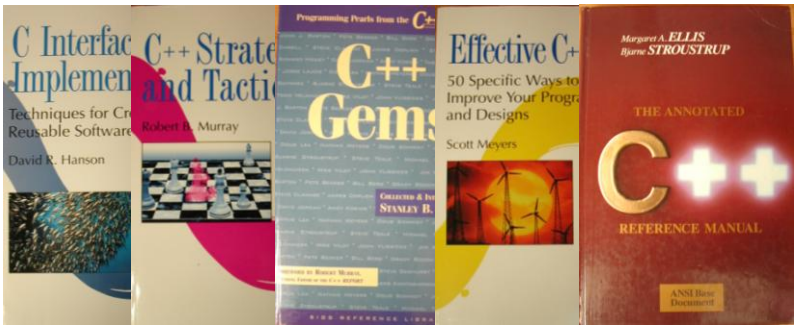
Emery Berger, Ben Zorn, Kathryn McKinley

Microsoft Research

THE UNIVERSITY OF TEXAS AT AUSTIN

Custom Memory Allocation

- Programmers replace `new`/`delete`, bypassing system allocator
  - Reduce runtime – *often*
  - Expand functionality – *sometimes*
  - Reduce space – *rarely*



- **Very** common practice
  - Apache, gcc, lcc, STL, database servers…
  - Language-level support in C++
  - Widely recommended

"Use custom allocators"

Drawbacks of Custom Allocators

- Avoiding system allocator:
  - More code to maintain & debug
  - Can't use memory debuggers
  - Not modular or robust:
    - Mix memory from custom and general-purpose allocators → **crash!**

⇨ **Increased burden on programmers**

Are custom allocators really *a win?*

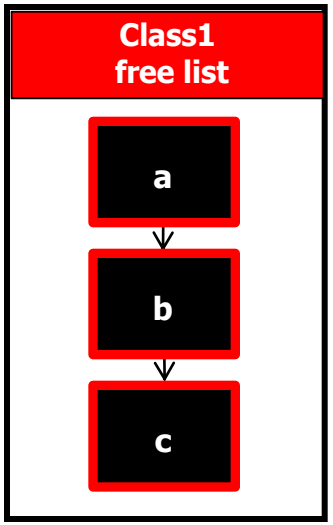
Overview

- Introduction
- Perceived benefits and drawbacks
- Three main kinds of custom allocators
- Comparison with general-purpose allocators
- Advantages and drawbacks of regions
- Reaps – generalization of regions & heaps

(1) Per-Class Allocators

- Recycle freed objects from a free list

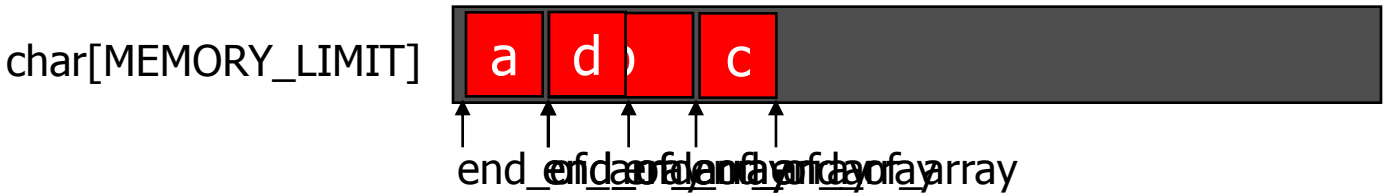
```
a = new Class1;
b = new Class1;
c = new Class1;
delete a;
delete b;
delete c;
a = new Class1;
b = new Class1;
c = new Class1;
```



- \+ *Fast*
  - \+ Linked list operations
- \+ *Simple*
  - \+ Identical semantics
  - \+ C++ language support
- \- *Possibly space-inefficient*

(II) Custom Patterns

- Tailor-made to fit allocation patterns
  - Example: *197.parser* (natural language parser)




```
a = xalloc(8);
b = xalloc(16);
c = xalloc(8);
xfree(b);
xfree(c);
d = xalloc(8);
```

+ *Fast*
  + Pointer-bumping allocation
- *Brittle*
  - Fixed memory size
  - Requires stack-like lifetimes

(III) Regions

- Separate areas, deletion only *en masse*

regioncreate(r)
regionmalloc(r, sz)
regiondelete(r)

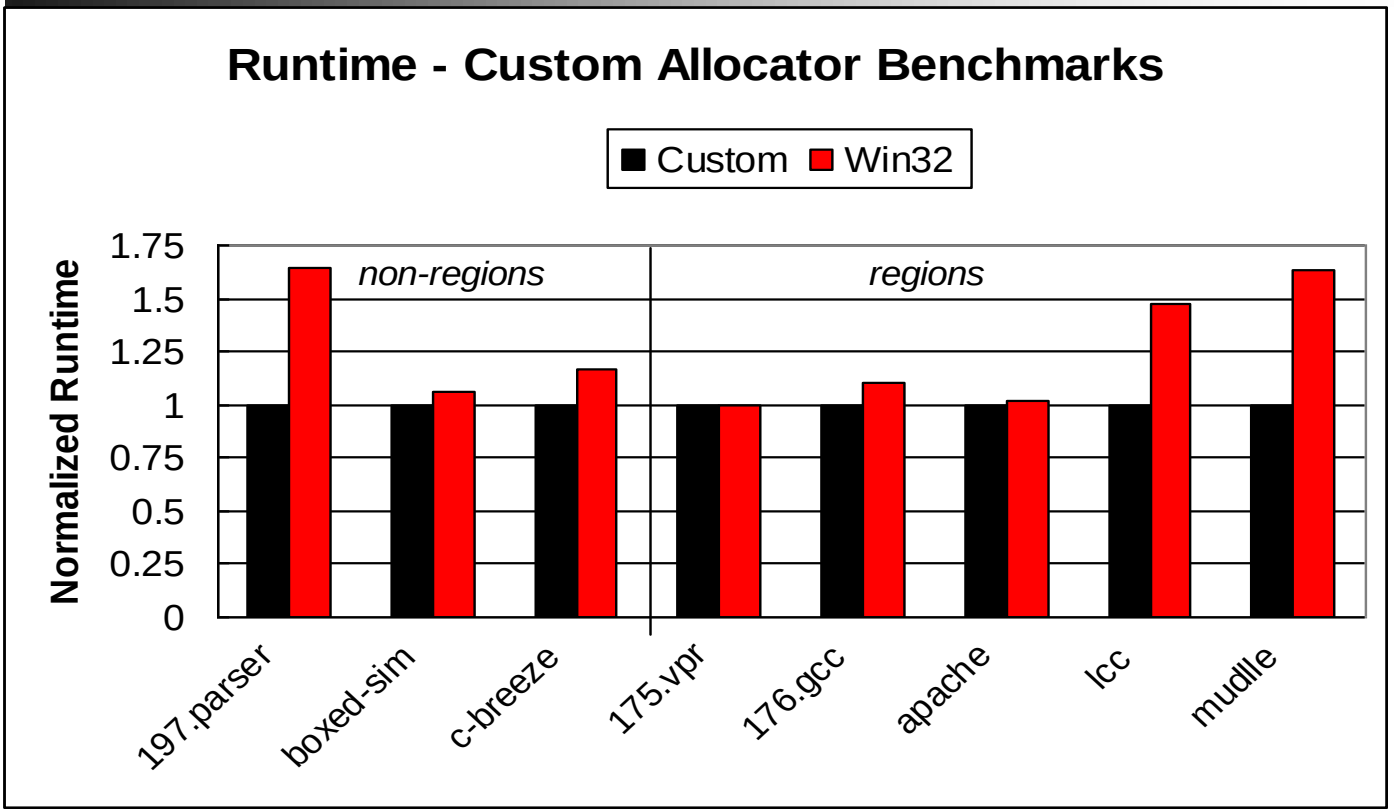
- *Fast*
  - Pointer-bumping allocation
  - Deletion of chunks
- *Convenient*
  - One call frees all memory
- *Risky*
  - Dangling references
  - Too much space

- Increasingly popular custom allocator

Overview

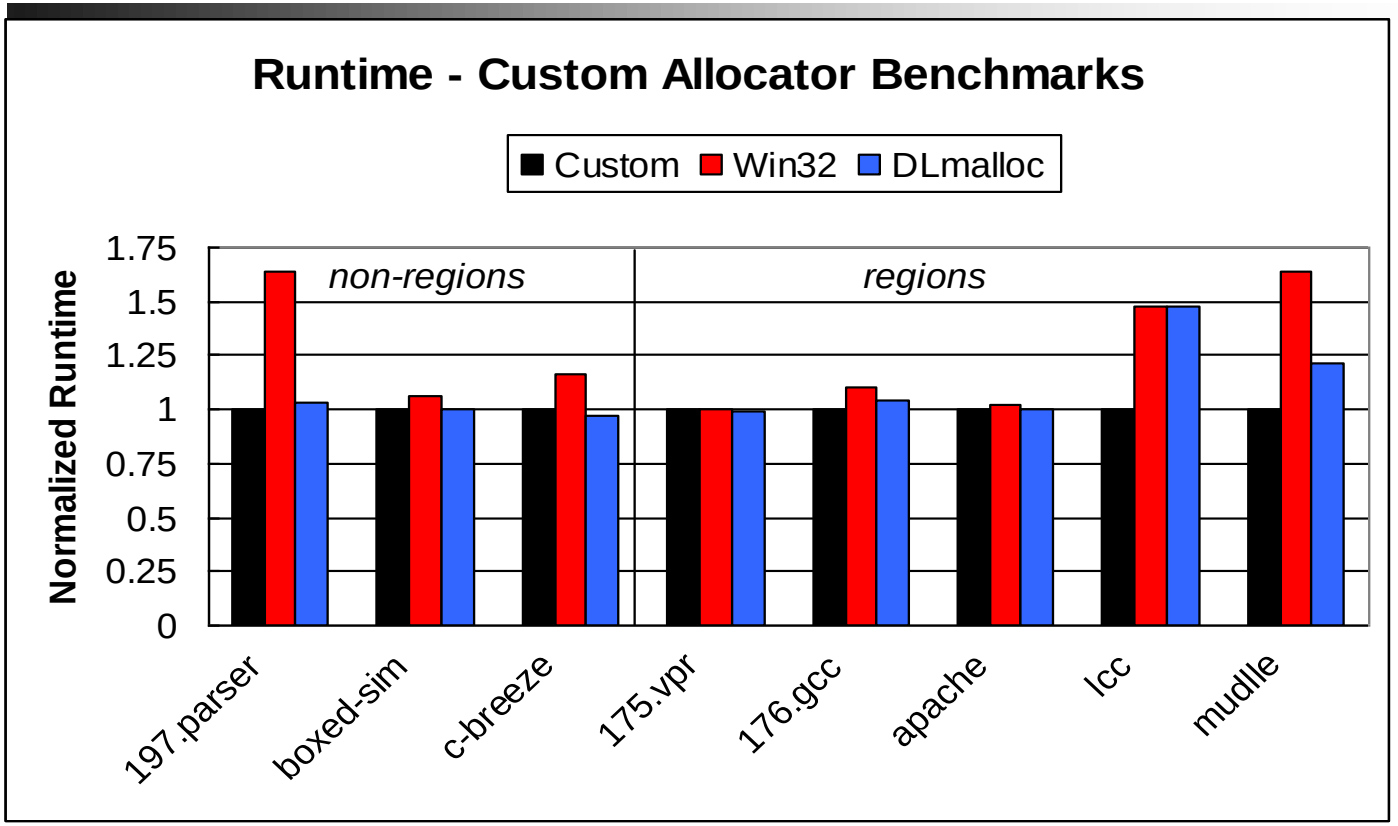
- Introduction
- Perceived benefits and drawbacks
- Three main kinds of custom allocators
- Comparison with general-purpose allocators
- Advantages and drawbacks of regions
- Reaps – generalization of regions & heaps

Custom Allocators Are Faster…



- As good as and sometimes *much faster* than Win32

Not So Fast…

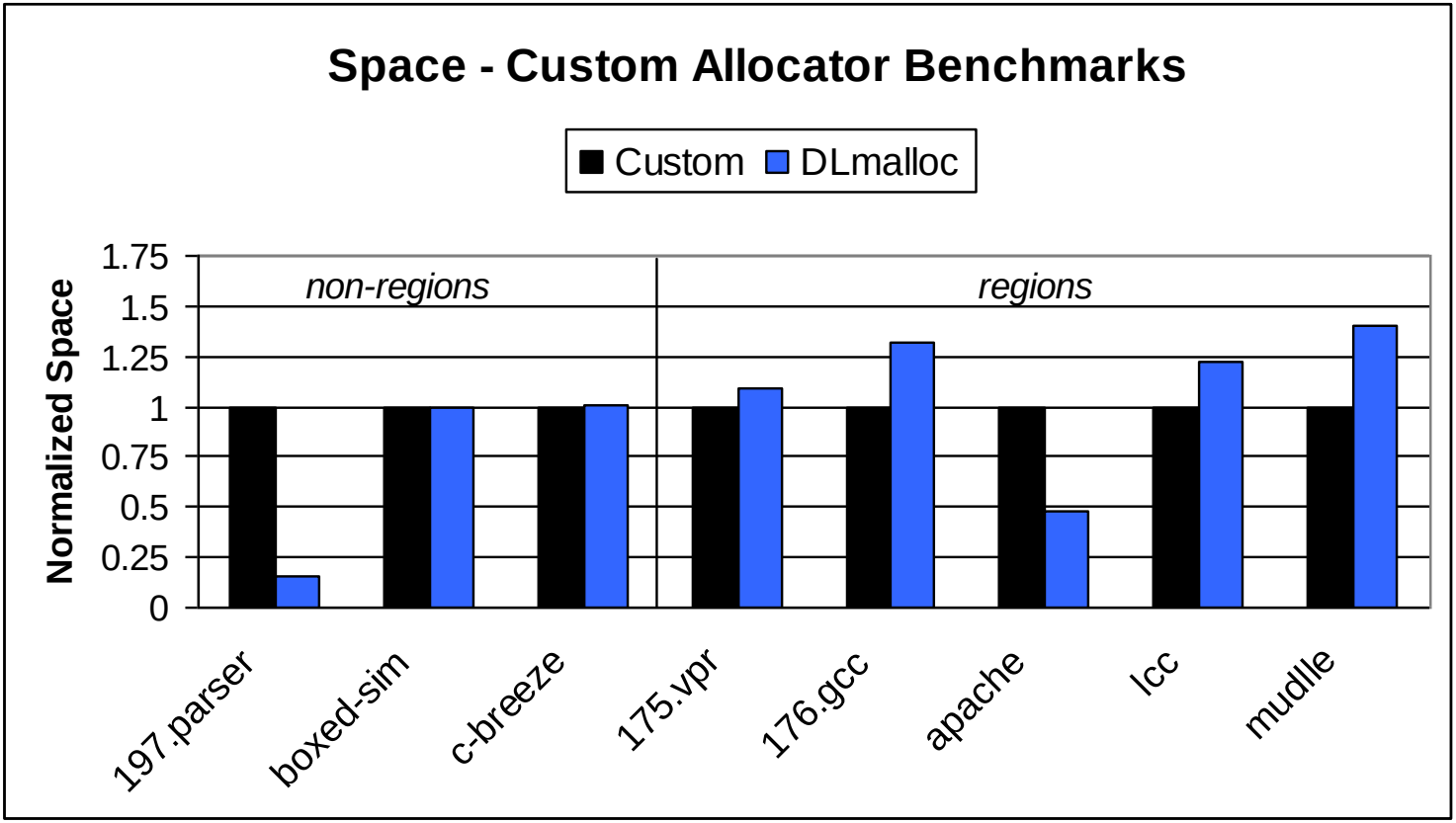


- DLmalloc: as fast or *faster* for most benchmarks

The Lea Allocator (DLmalloc 2.7.0)

- Mature public-domain general-purpose allocator
- Optimized for common allocation patterns
  - Per-size quicklists ≈ per-class allocation
- Deferred coalescing (combining adjacent free objects)
  - ⇨ Highly-optimized fastpath
- Space-efficient

Space Consumption: Mixed Results



Overview

- Introduction
- Perceived benefits and drawbacks
- Three main kinds of custom allocators
- Comparison with general-purpose allocators
- Advantages and drawbacks of regions
- Reaps – generalization of regions & heaps

Regions – Pros and Cons

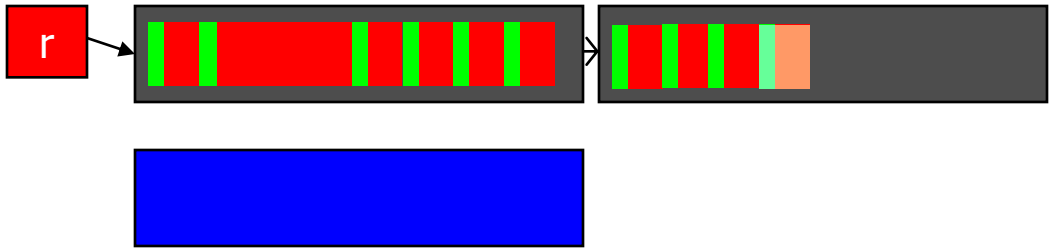
+ Fast, convenient, etc.
+ Avoid resource leaks (e.g., Apache)
  - Tear down memory for terminated connections

- No individual object deletion
  - ⇨ **Unbounded memory consumption**
    (producer-consumer, long-running computations, off-the-shelf programs)
  - Apache: vulnerable to DoS, memory leaks

Reap Hybrid Allocator

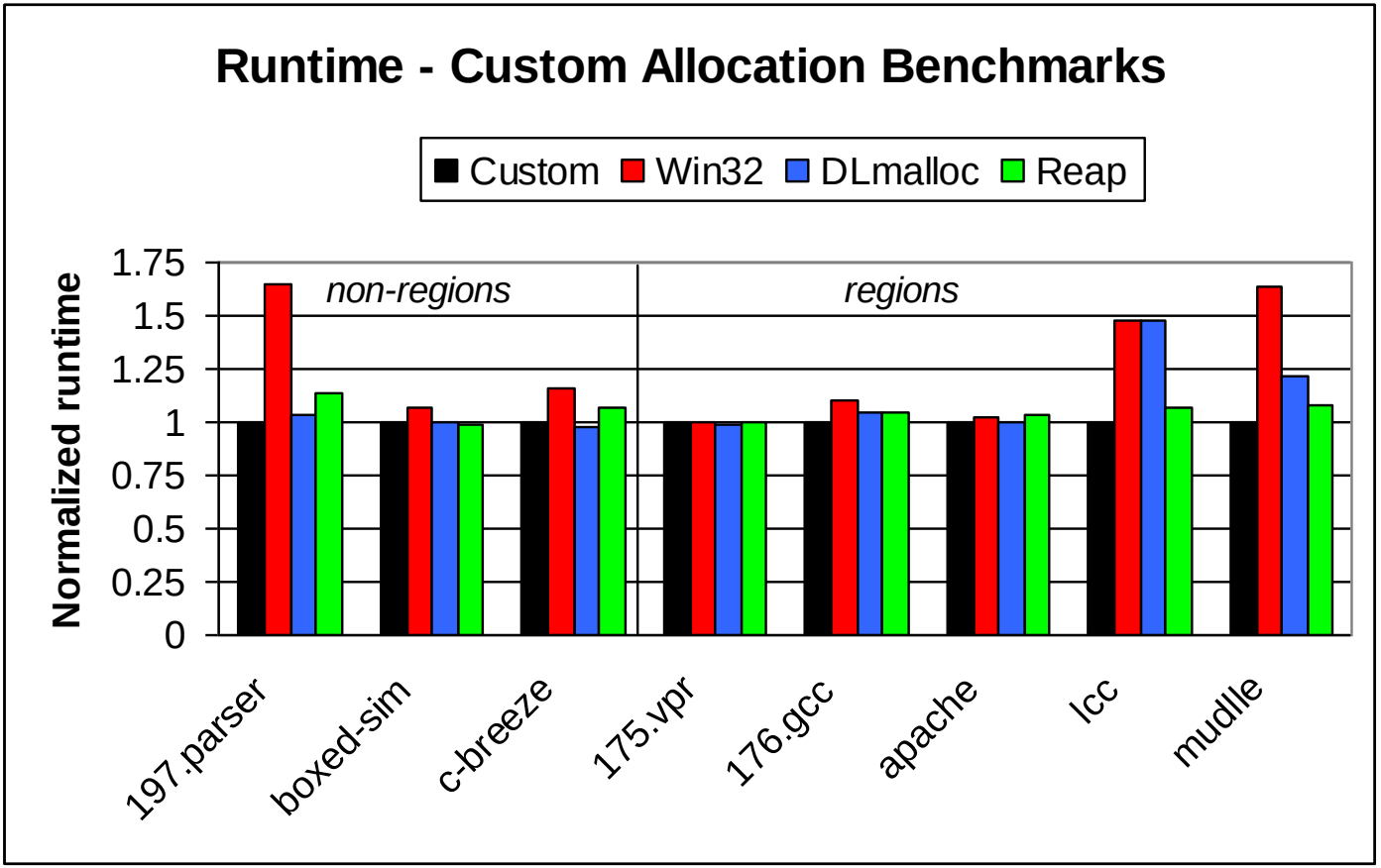
- Reap = *region* + *heap*
  - Adds individual object deletion & heap

reapcreate(r)
reapmalloc(r, sz)
reapfree(r,p)
reapdelete(r)

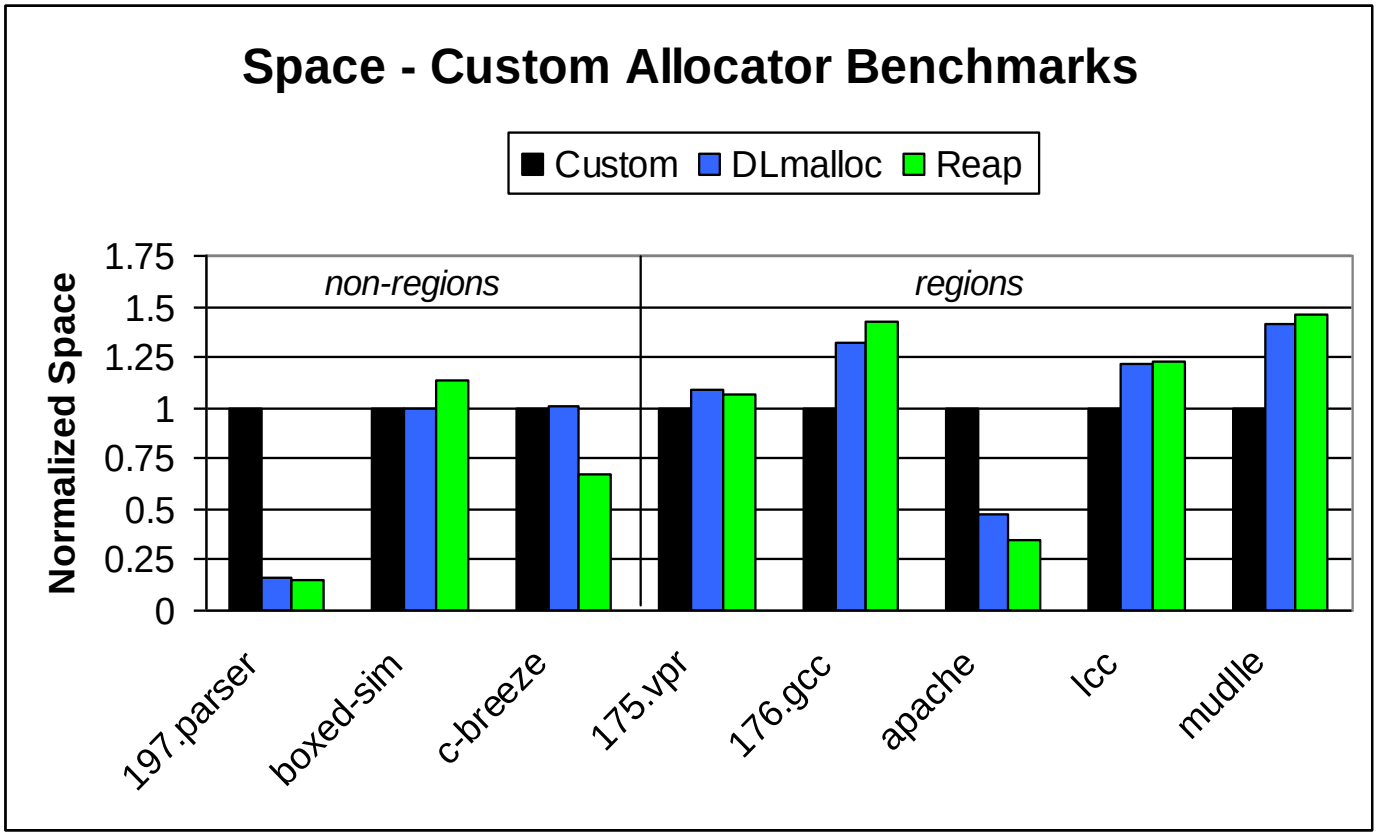


- \+ Can reduce memory consumption
- \+ Fast
  - \+ Adapts to use (region or heap style)
  - \+ Cheap deletion

Reap Runtime



Reap Space


Space - Custom Allocator Benchmarks
Custom
DLmalloc
Reap
Normalized Space
1.75
1.5
1.25
1
0.75
0.5
0.25
0
non-regions
regions
197.parser
boxed-sim
c-breeze
175.vpr
176.gcc
apache
lcc
mudlle

Reap: Best of Both Worlds

- Allows mixing of regions and new/delete
- Case study:
  - New Apache module *"mod_bc"*
    - *bc*: C-based arbitrary-precision calculator
    - Changed 20 lines out of 8000
  - Benchmark: compute 1000th prime
    - With Reap: 240K
    - Without Reap: 7.4MB

Conclusions and Future Work

- Empirical study of custom allocators
  - Lea allocator often as fast or *faster*
  - **Non-region custom allocation ineffective**
- Reap: region performance without drawbacks
- Future work:
  - Reduce space with per-page bitmaps
  - Combine with scalable general-purpose allocator (e.g., Hoard)

Software

http://www.cs.umass.edu/~emery

(Reap: part of Heap Layers distribution)

http://g.oswego.edu

(DLmalloc 2.7.0)

If You Can Read This, I Went Too Far

Backup Slides

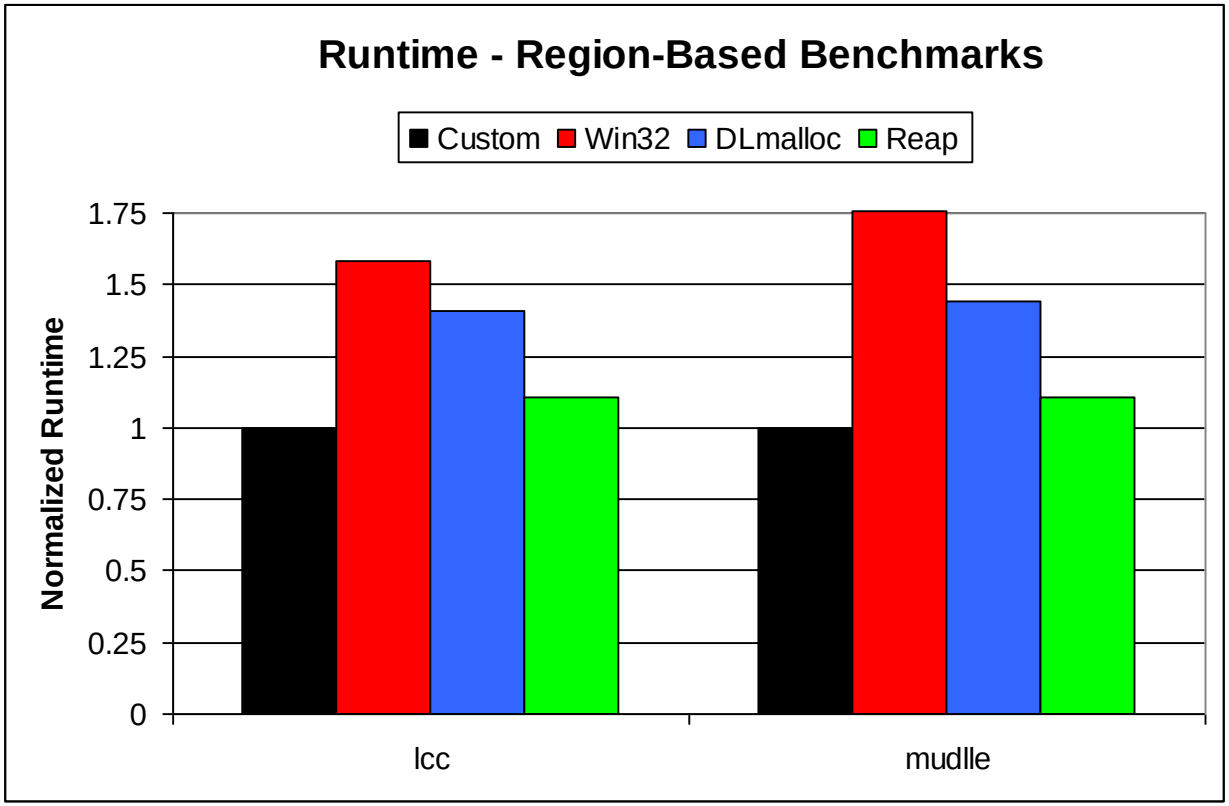
Experimental Methodology

- Comparing to general-purpose allocators
  - Same semantics: no problem
    - E.g., disable per-class allocators
  - Different semantics: use *emulator*
    - Uses general-purpose allocator
    - Adds bookkeeping to support region semantics

Why Did They Do That?

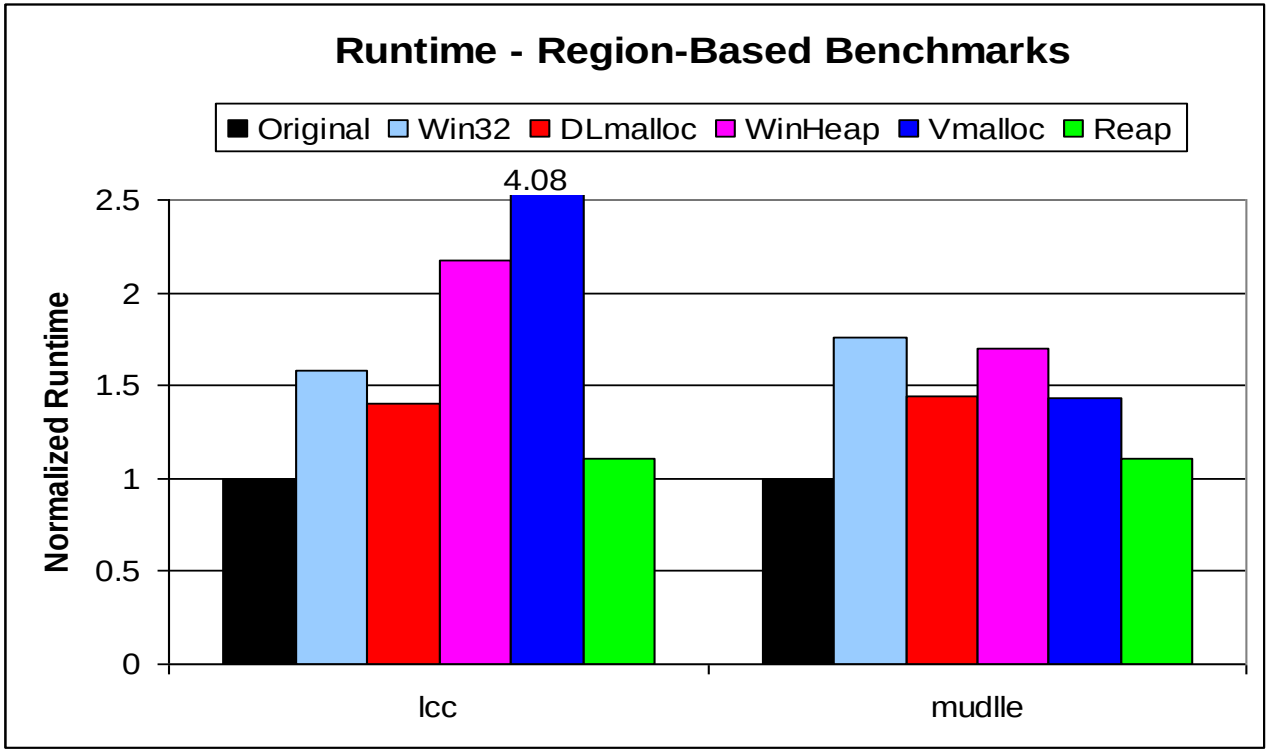
- Recommended practice
- Premature optimization
  - Microbenchmarks vs. actual performance
- Drift
  - Not bottleneck anymore
- Improved competition
  - Modern allocators are better

Reaps as Regions: Runtime



- Reap performance nearly matches regions

Using Reap as Regions



Reap performance nearly matches regions

Drawbacks of Regions

- Can't reclaim memory within regions
  - Bad for long-running computations, producer-consumer patterns, "malloc/free" programs
  - ⇨ **unbounded memory consumption**

- Current situation for Apache:
  - vulnerable to denial-of-service
  - limits runtime of connections
  - limits module programming

Use Custom Allocators?

- Strongly recommended by practitioners
- Little hard data on performance/space improvements
  - Only one previous study [Zorn 1992]
  - Focused on just one type of allocator
  - Custom allocators: waste of time
    - Small gains, bad allocators
- Different allocators better? Trade-offs?

Kinds of Custom Allocators

- Three basic types of custom allocators
  - Per-class
    - Fast
  - Custom patterns
    - Fast, but very special-purpose
  - Regions
    - Fast, possibly more space-efficient
    - Convenient
    - Variants: *nested*, *obstacks*

Optimization Opportunity