Compacting the Uncompactable!

MESH  Compact Memory Management
      For C/C++ Applications

Emery Berger
UMass Amherst / MSR

with Bobby Powers, David Tench, & Andrew McGregor
University of Massachusetts Amherst
http://libmesh.org
[PLDI 2019]
Reconquer all of Spain!
MALLOC
of La MANCHA!

Reconquer all of Spain!
MALLOC

La Mancha

Reconquer Malloc all of Spain!
The Impossible Dream (The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC
of
LA MANCHA

(malloc)
The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

La MANCHA

MALLOC

of la MANCHA
The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOCC

of
LA
MÁNCAR
The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

LA MANCHA
MALLOC

of

la

MANCHA

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH
The Impossible Dream (The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH
The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

of

LA

MANCHA
¡Fragmentación!

OS Page ("Spain")

Live objects (malloc'd)
¡Fragmentación!

OS Pages

Live objects
¿Fragmentación!

OS Pages

Live objects

⇒ 1
¡Fragmentación!

OS Pages

$O(\log \frac{1}{x})$ 13x!

[Robson '77]

Live objects
¿Compacción?

OS Pages

$O(\log \frac{1}{13})$

13x!

[Robson ’77]

Live objects
¿Compacción?

OS Pages
0(1)

Live objects
Why not a tracing GC?

- Native interoperability with unmanaged code
- Deterministic destruction provides:
  - No “finalizer problems” like resurrection, threading, etc.
  - Deterministic performance: can test/debug performance stutters
- Performance:
  - GC use ~3-4x more memory than ARC to achieve good performance
  - Memory usage is very important for mobile and cloud apps
  - Incremental/concurrent GCs slow the mutator like ARC does

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management
Matthew Hertz, Emery D. Berger. OOPSLA’05
Native interoperability with unmanaged code

Deterministic destruction provides:
- No “finalizer problems” like resurrection, threading, etc.
- Deterministic performance: can test/debug performance stutters

Performance:
- GC use ~3-4x more memory than ARC to achieve good performance
- Memory usage is very important for mobile and cloud apps
- Incremental/concurrent GCs slow the mutator like ARC does
¡Compacción!
Compaction In Action

OS Page

References

Live objects
$ cc -o yolo main.cc
$ strip yolo
$ cc -o yolo main.cc
$ strip yolo
$ ./yolo

No way to precisely distinguish pointers from integers
$ cc -o yolo main.cc
$ strip yolo
$ ./yolo

No way to precisely distinguish pointers from integers
union tiny
{
    int * ptr;
    uintptr_t flag;
};
tiny x;

// initialize
x.ptr = new int;

// set flag true
x.flag |= 1;
MESH

Compaction without Relocation for C/C++

No code changes
No recompilation
LD_PRELOAD and go
17% heap size reduction
< 1% performance overhead

![Graph showing RSS (MiB) over time with two lines representing default jemalloc and Mesh, indicating a reduction in heap size with minimal performance overhead.](image-url)
redis

Compaction time

5x reduction in time spent compacting
Meshing: compaction without (virtual!) relocation
Pages are **Meshable** when they:

- Hold objects of the same size class
Pages are **Meshable** when they:

- Hold objects of the same size class
- Have non-overlapping object offsets
Meshing

Virtual

Physical
Meshing

Mark virtual page read-only

Virtual

Physical
Meshing

Virtual

Physical

Copy (maintaining offsets)
Meshing

Update page tables with `mmap`, `memfd`, `MAP_SHARED`
Meshing

Mark virtual page read/write

Virtual

Physical
Meshing

No virtual addresses changed!

Virtual

Physical
Worst Case:

low occupancy, non-meshable pages
Worst Case:
many
low occupancy, non-meshable pages
Standard allocators
malloc(256)
malloc(256)
malloc(256)

Standard allocators
Mesh uses **randomization** to ensure live objects are uniformly distributed.
Regular allocation patterns are real.
How to randomize allocation?
Random probing:

```python
while true:
    if rand_off().is_free:
        return rand_off
```
malloc(256)

Random probing:

```python
while true:
    if rand_off().is_free:
        return rand_off
```
`malloc(256)`

Random probing:

```python
while True:
    if rand_off().is_free:
        return rand_off
```
`malloc(256)`

Random probing:

```
while true:
    if rand_off().is_free:
        return rand_off
```

(DieHard [Berger & Zorn 2006])
Random probing fast in expectation *iff* page occupancy is low

but this is at odds with minimizing heap size!
Shuffle Vector:
Fast randomized allocation + full page utilization
Shuffle Vector: Fast randomized allocation

load

Thread-local shuffle vector
Shuffle Vector: Fast randomized allocation

load

Thread-local shuffle vector
All Pages Meshable
Finding pages to Mesh
Problem: Find meshing that releases maximum number of pages

- Run in the `free()` slowpath
- At most once every 100 ms
- Treat each size class independently
Problem: Find meshing that releases the maximum number of pages
MinCliqueCover

(NP-Complete)

BUT! Randomness ensures we can get away with solving simpler graph problem (Matching)
Wrinkle: building this graph would require RAM + time
SplitMesher: approximates Matching without materializing meshing graph

Set of partially full pages → Pairs of meshable pages
Iterate, comparing $a[i]$ to $b[i]$
loop, comparing $a[i]$ to $b[(i+1)\%\text{len}]$
Remove found match
Continue
SplitMesher: approximates Matching without materializing meshing graph

$O(n/q)$ time

($q$ is the global probability of spans meshing)
SplitMesher: approximates Matching without materializing meshing graph

\[ O\left(\frac{n}{q}\right) \text{ time} \]

(\(q\) is the global probability of spans meshing)

\[ 1/2^* \text{ approximation w.h.p.} \]
17% heap size reduction
< 1% performance overhead
Ruby Compaction for Free

![Graph showing RSS (MiB) over time since program start (seconds) for Mesh and Mesh (no rand)]
http://LIBMESH.org

¡Compacción sin Relocación!
(compaction without relocation)