Mesh: Compacting Memory Management for C/C++ Applications

Bobby Powers
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA, USA
bpowers@cs.umass.edu

David Tench
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA, USA
dtench@cs.umass.edu

Emery D. Berger
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA, USA
emery@cs.umass.edu

Andrew McGregor
College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, MA, USA
mcgregor@cs.umass.edu

Abstract

Programs written in C/C++ can suffer from serious memory fragmentation, leading to low utilization of memory, degraded performance, and application failure due to memory exhaustion. This paper introduces Mesh, a plug-in replacement for `malloc` that, for the first time, eliminates fragmentation in unmodified C/C++ applications. Mesh combines novel randomized algorithms with widely-supported virtual memory operations to provably reduce fragmentation, breaking the classical Robson bounds with high probability. Mesh generally matches the runtime performance of state-of-the-art memory allocators while reducing memory consumption; in particular, it reduces the memory of consumption of Firefox by 16% and Redis by 39%.

CCS Concepts • **Software and its engineering → Allocation / deallocation strategies**.

Keywords Memory management, runtime systems, unmanaged languages

ACM Reference Format:
Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314582

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314582

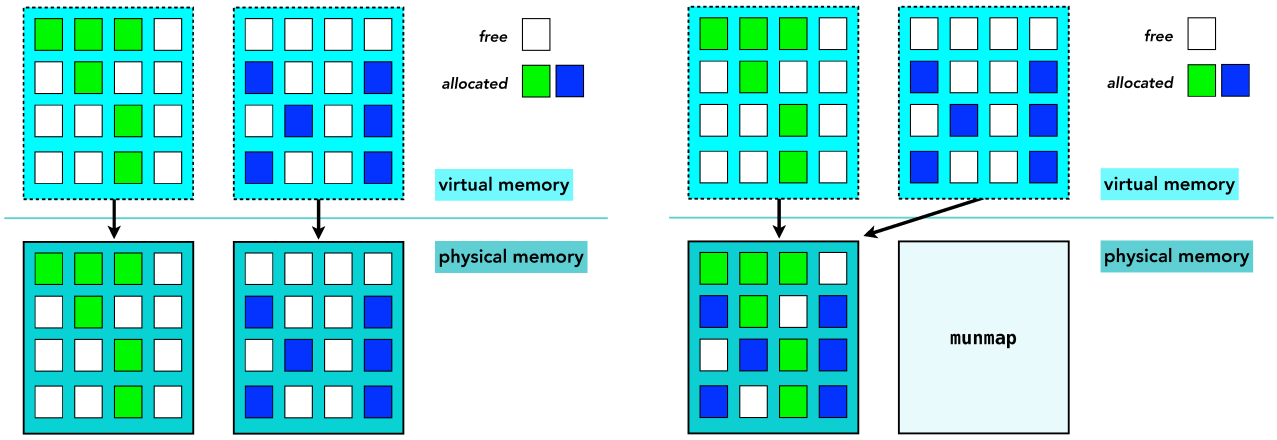
1 Introduction

Memory consumption is a serious concern across the spectrum of modern computing platforms, from mobile to desktop to datacenters. For example, on low-end Android devices, Google reports that more than 99 percent of Chrome crashes are due to running out of memory when attempting to display a web page [19]. On desktops, the Firefox web browser has been the subject of a five-year effort to reduce its memory footprint [33]. In datacenters, developers implement a range of techniques from custom allocators to other *ad hoc* approaches in an effort to increase memory utilization [29, 32].

A key challenge is that, unlike in garbage-collected environments, automatically reducing a C/C++ application's memory footprint via compaction is not possible. Because the addresses of allocated objects are directly exposed to programmers, C/C++ applications can freely modify or hide addresses. For example, a program may stash addresses in integers, store flags in the low bits of aligned addresses, perform arithmetic on addresses and later reference them, or even store addresses to disk and later reload them. This hostile environment makes it impossible to safely relocate objects: if an object is relocated, all pointers to its original location must be updated. However, there is no way to safely update *every* reference when they are ambiguous, much less when they are absent.

Existing memory allocators for C/C++ employ a variety of best-effort heuristics aimed at reducing average fragmentation [21]. However, these approaches are inherently limited. In a classic result, Robson showed that all such allocators can suffer from catastrophic memory fragmentation [30]. This increase in memory consumption can be as high as the log of the ratio between the largest and smallest object sizes allocated. For example, for an application that allocates 16-byte and 128KB objects, it is possible for it to consume 13× more memory than required.

Despite nearly fifty years of conventional wisdom indicating that compaction is impossible in unmanaged languages,



(a) Before: these pages are candidates for "meshing" because their allocated objects do not overlap.

(b) After: both virtual pages now point to the first physical page; the second page is now freed.

Figure 1. Mesh in action. Mesh employs novel randomized algorithms that let it efficiently find and then "mesh" candidate pages within *spans* (contiguous 4K pages) whose contents do not overlap. In this example, it increases memory utilization across these pages from 37.5% to 75%, and returns one physical page to the OS (via `munmap`), reducing the overall memory footprint. Mesh's randomized allocation algorithm ensures meshing's effectiveness with high probability.

this paper shows that it is not only possible but also practical. It introduces Mesh, a memory allocator that effectively and efficiently performs compacting memory management to reduce memory usage in unmodified C and C++ applications.

Crucially and counterintuitively, Mesh performs compaction without relocation; that is, without changing the addresses of objects. This property is vital for compatibility with arbitrary C/C++ applications. To achieve this, Mesh builds on a mechanism which we call *meshing*, first introduced by Novark et al.'s Hound memory leak detector [28]. Hound employed meshing in an effort to avoid catastrophic memory consumption induced by its memory-inefficient allocation scheme, which can only reclaim memory when every object on a page is freed. Hound first searches for pages whose live objects do not overlap. It then copies the contents of one page onto the other, remaps one of the *virtual* pages to point to the single *physical* page now holding the contents of both pages, and finally relinquishes the other physical page to the OS. Figure 1 illustrates meshing in action.

Mesh overcomes two key technical challenges of meshing that previously made it both inefficient and potentially entirely ineffective. First, Hound's search for pages to mesh involves a linear scan of pages on calls to `free`. While this search is more efficient than a naive $O(n^2)$ search of all possible pairs of pages, it remains prohibitively expensive for use in the context of a general-purpose allocator. Second, Hound offers no guarantees that *any* pages would ever be meshable. Consider an application that happens to allocate even one object in the same offset in every page. That layout would preclude meshing altogether, eliminating the possibility of saving any space.

Mesh makes meshing both efficient and provably effective (with high probability) by combining it with two novel randomized algorithms. First, Mesh uses a space-efficient randomized allocation strategy that effectively scatters objects within each virtual page, making the above scenario provably exceedingly unlikely. Second, Mesh incorporates an efficient randomized algorithm that is guaranteed with high probability to quickly find candidate pages that are likely to mesh. These two algorithms work in concert to enable formal guarantees on Mesh's effectiveness. Our analysis shows that Mesh breaks the above-mentioned Robson worst case bounds for fragmentation with high probability [30], as memory reclaimed by meshing is available for use by any size class. This ability to redistribute memory from one size class to another enables Mesh to adapt to changes in an application's allocation behavior in a way other segregated-fit allocators cannot.

We implement Mesh as a library for C/C++ applications running on Linux or Mac OS X. Mesh interposes on memory management operations, making it possible to use it without code changes or recompilation by setting the appropriate environment variable to load the Mesh library (e.g., `export LD_PRELOAD=libmesh.so` on Linux). Our evaluation demonstrates that our implementation of Mesh is both fast and efficient in practice. It generally matches the performance of state-of-the-art allocators while guaranteeing the absence of catastrophic fragmentation with high probability. In addition, it occasionally yields substantial space savings: replacing the standard allocator with Mesh automatically reduces memory consumption by 16% (Firefox) to 39% (Redis).

1.1 Contributions

This paper makes the following contributions:

- It introduces **Mesh**, a novel memory allocator that acts as a plug-in replacement for `malloc`. Mesh combines remapping of virtual to physical pages (meshing) with randomized allocation and search algorithms to enable safe and effective *compaction without relocation* for C/C++ (§2, §3, §4).
- It presents theoretical results that guarantee Mesh's efficiency and effectiveness with high probability (§5).
- It evaluates Mesh's performance empirically, demonstrating Mesh's ability to reduce space consumption while generally imposing low runtime overhead (§6).

2 Overview

This section provides a high-level overview of how Mesh works and gives some intuition as to how its algorithms and implementation ensure its efficiency and effectiveness, before diving into detailed description of Mesh's algorithms (§3), implementation (§4), and its theoretical analysis (§5).

2.1 Remapping Virtual Pages

Mesh enables compaction without relocating object addresses; it depends only on hardware-level virtual memory support, which is standard on most computing platforms like x86 and ARM64. Mesh works by finding pairs of pages and merging them together *physically* but not *virtually*: this merging lets it relinquish physical pages to the OS.

Meshing is only possible when no objects on the pages occupy the same offsets. A key observation is that as fragmentation increases (that is, as there are more free objects), the likelihood of successfully finding pairs of pages that mesh also increases.

Figure 1 schematically illustrates the meshing process. Mesh manages memory at the granularity of *spans*, which are runs of contiguous 4K pages (for purposes of illustration, the figure shows single-page spans). Each span only contains same-sized objects. The figure shows two spans of memory with low utilization (each is under 40% occupied) and whose allocations are at non-overlapping offsets.

Meshing consolidates allocations from each span onto one physical span. Each object in the resulting meshed span resides at the same offset as it did in its original span; that is, its virtual addresses are preserved, making meshing invisible to the application. Meshing then updates the virtual-to-physical mapping (the page tables) for the process so that both virtual spans point to the same physical span. The second physical span is returned to the OS. When average occupancy is low, meshing can consolidate many pages, offering the potential for considerable space savings.

2.2 Random Allocation

A key threat to meshing is that pages could contain objects at the same offset, preventing them from being meshed. In the worst case, all spans would have only one allocated object, each at the same offset, making them non-meshable. Mesh employs randomized allocation to make this worst-case behavior exceedingly unlikely. It allocates objects uniformly at random across all available offsets in a span. As a result, the probability that all objects will occupy the same offset is $(1/b)^{n-1}$, where b is the number of objects in a span, and n is the number of spans.

In practice, the resulting probability of being unable to mesh many pages is vanishingly small. For example, when meshing 64 spans with one 16-byte object allocated on each (so that the number of objects b in a 4K span is 256), the likelihood of being unable to mesh any of these spans is 10^{-152}. To put this into perspective, there are estimated to be roughly 10^{82} particles in the universe.

We use randomness to guide the design of Mesh's algorithms (§3) and implementation (§4); this randomization lets us prove robust guarantees of its performance (§5), showing that Mesh breaks the Robson bounds with high probability.

2.3 Finding Spans to Mesh

Given a set of spans, our goal is to mesh them in a way that frees as many physical pages as possible. We can think of this task as that of partitioning the spans into subsets such that the spans in each subset mesh. An optimal partition would minimize the number of such subsets.

Unfortunately, as we show, optimal meshing is not feasible (§5). Instead, the algorithms in Section 3 present practical methods for finding high-quality meshes under real-world time constraints. We show that solving a simplified version of the problem (§3) is sufficient to achieve reasonable meshes with high probability (§5).

3 Algorithms

Mesh comprises three main algorithmic components: allocation (§3.1), deallocation (§3.2), and finding spans to mesh (§3.3). Unless otherwise noted and without loss of generality, all algorithms described here are per size class (within spans, all objects are same size).

3.1 Allocation

Allocation in Mesh consists of two steps: (1) finding a span to allocate from, and (2) randomly allocating an object from that span. Mesh always allocates from a thread-local shuffle vector – a randomized version of a freelist (described in detail in §4.2). The shuffle vector contains offsets corresponding to the slots of a single span. We call that span the *attached* span for a given thread.

If the shuffle vector is empty, Mesh relinquishes the current thread's attached span (if one exists) to the *global heap*

(which holds all unattached spans), and asks it to select a new span. If there are no partially full spans, the global heap returns a new, empty span. Otherwise, it selects a partially full span for reuse. To maximize utilization, the global heap groups spans into bins organized by decreasing occupancy (e.g., 75-99% full in one bin, 50-74% in the next). The global heap scans for the first non-empty bin (by decreasing occupancy), and randomly selects a span from that bin.

Once a span has been selected, the allocator adds the offsets corresponding to the free slots in that span to the thread-local shuffle vector (in a random order). Mesh pops the first entry off the shuffle vector and returns it.

3.2 Deallocation

Deallocation behaves differently depending on whether the free is local (the address belongs to the current thread's attached span), remote (the object belongs to another thread's attached span), or if it belongs to the global heap.

For local frees, Mesh adds the object's offset onto the span's shuffle vector in a random position and returns. For remote frees, Mesh atomically resets the bit in the corresponding index in a bitmap associated with each span. Finally, for an object belonging to the global heap, Mesh marks the object as free, updates the span's occupancy bin; this action may additionally trigger meshing.

3.3 Meshing

When meshing, Mesh randomly chooses pairs of spans and attempts to mesh each pair. The meshing algorithm, which we call SplitMesher (Figure 2), is designed both for practical effectiveness and for its theoretical guarantees. The parameter t, which determines the maximum number of times each span is probed (line 3), enables space-time trade-offs. The parameter t can be increased to improve mesh quality and therefore reduce space, or decreased to improve runtime, at the cost of sacrificed meshing opportunities. We empirically found that $t = 64$ balances runtime and meshing effectiveness, and use this value in our implementation.

SplitMesher proceeds by iterating through S_l and checking whether it can mesh each span with another span chosen from S_r (line 6). If so, it removes these spans from their respective lists and meshes them (lines 7–9). SplitMesher repeats until it has checked $t * |S_l|$ pairs of spans; §4.5 describes the implementation of SplitMesher in detail.

SplitMesher(S, t)

```
1  n = length(S)
2  S_l, S_r = S[1 : n/2], S[n/2 + 1 : n]
3  for (i = 0, i < t, i++)
4      len = |S_l|
5      for (j = 0, j < len, j++)
6          if Meshable (S_l(j), S_r(j + i % len))
7              S_l ← S_l \ S_l(j)
8              S_r ← S_r \ S_r(j + i % len)
9              mesh(S_l(j), S_r(j + i % len))
```

Figure 2. Meshing random pairs of spans. SplitMesher splits the randomly ordered span list S into halves, then probes pairs between halves for meshes. Each span is probed up to t times.

4 Implementation

We implement Mesh as a drop-in replacement memory allocator that implements meshing for single or multi-threaded applications written in C/C++. Its current implementation work for 64-bit Linux and Mac OS X binaries. Mesh can be explicitly linked against by passing `-lmesh` to the linker at compile time, or loaded dynamically by setting the `LD_PRELOAD` (Linux) or `DYLD_INSERT_LIBRARIES` (Mac OS X) environment variables to point to the Mesh library. When loaded, Mesh interposes on standard libc functions to replace all memory allocation functions.

Mesh combines traditional allocation strategies with meshing to minimize heap usage. Like most modern memory allocators [2, 3, 14, 16, 27], Mesh is a segregated-fit allocator. Mesh employs fine-grained size classes to reduce internal fragmentation due to rounding up to the nearest size class. Mesh uses the same size classes as those used by jemalloc for objects 1024 bytes and smaller [14], and power-of-two size classes for objects between 1024 and 16K. Allocations are fulfilled from the smallest size class they fit in (e.g., objects of size 33–48 bytes are served from the 48-byte size class); objects larger than 16K are individually fulfilled from the global arena. Small objects are allocated out of *spans* (§2), which are multiples of the page size and contain between 8 and 256 objects of a fixed size. Having at least eight objects per span helps amortize the cost of reserving memory from the global manager for the current thread's allocator.

Objects of 4KB and larger are always page-aligned and span at least one entire page. Mesh does not consider these objects for meshing; instead, the pages are directly freed to the OS.

Mesh's heap organization consists of four main components. *MiniHeaps* track occupancy and other metadata for spans (§4.1). *Shuffle vectors* enable efficient, random allocation out of a MiniHeap (§4.2). *Thread local heaps* satisfy small-object allocation requests without the need for locks or atomic operations in the common case (§4.3). Finally, the *global heap* (§4.4) manages runtime state shared by all threads, large object allocation, and coordinates meshing operations (§4.5).

4.1 MiniHeaps

MiniHeaps manage allocated physical spans of memory and are either *attached* or *detached*. An attached MiniHeap is

owned by a specific thread-local heap, while a detached MiniHeap is only referenced through the global heap. New small objects are *only* allocated out of attached MiniHeaps.

Each MiniHeap contains metadata that comprises span length, object size, allocation bitmap, and the start addresses of any virtual spans meshed to a unique physical span. The number of objects that can be allocated from a MiniHeap bitmap is *objectCount* = *spanSize* / *objSize*. The allocation bitmap is initialized to *objectCount* zero bits.

When a MiniHeap is attached to a thread-local *shuffle vector* (§4.2), each offset that is unset in the MiniHeap's bitmap is added to the shuffle vector, with that bit now atomically set to one in the bitmap. This approach is designed to allow multiple threads to free objects which keeping most memory allocation operations local in the common case.

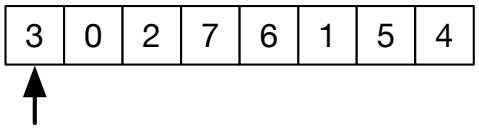
When an object is freed and the free is non-local (§3.2), the bit is reset. When a new MiniHeap is allocated, there is only one virtual span that points to the physical memory it manages. After meshing, there may be multiple virtual spans pointing to the MiniHeap's physical memory.

4.2 Shuffle Vectors

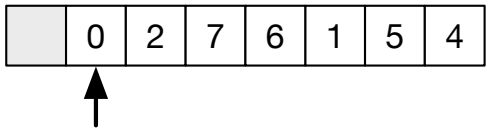
Shuffle vectors are a novel data structure that lets Mesh perform randomized allocation out of a MiniHeap efficiently and with low space overhead.

Previous memory allocators that have employed randomization (for security or reliability) perform randomized allocation by random probing into bitmaps [3, 27]. In these allocators, a memory allocation request chooses a random number in the range [0, *objectCount* − 1]. If the associated bit is zero in the bitmap, the allocator sets it to one and returns the address of the corresponding offset. If the offset is already one, meaning that the object is in use, a new random number is chosen and the process repeated. Random probing allocates objects in $O(1)$ *expected* time but requires overprovisioning memory by a constant factor (e.g., 2× more memory must be allocated than needed). This overprovisioning is at odds with our goal of *reducing* space overhead.

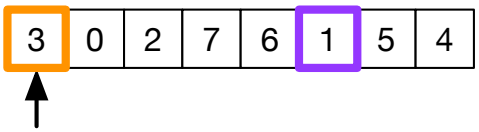
Shuffle vectors solve this problem, combining low space overhead with worst-case $O(1)$ running time for `malloc` and `free`. Each comprises a fixed-size array consisting of all the offsets from a span that are *not* already allocated, and an allocation index representing the head. Each vector is initially randomized with the Knuth-Fischer-Yates shuffle [22], and its allocation index is set to 0. Allocation proceeds by selecting the next available number in the vector, "bumping" the allocation index and returning the corresponding address. Deallocation works by placing the freed object at the front of the vector and performing one iteration of the shuffle algorithm; this operation preserves randomization of the vector. Figure 3 illustrates this process, while Figure 4 has pseudocode listings for initialization, allocation, and deallocation.



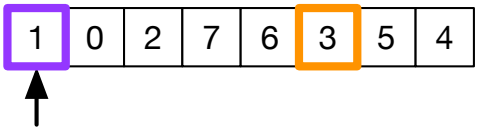
(a) A shuffle vector for a span of size 8, where no objects have yet been allocated.



(b) The shuffle vector after the first object has been allocated.



(c) On `free`, the object's offset is pushed onto the front of the vector, the allocation index is updated, and the offset is swapped with a randomly chosen offset.



(d) Finally, after the swap, new allocations proceed in a bump-pointer like fashion.

Figure 3. Shuffle vectors compactly enable fast random allocation. Indices (one byte each) are maintained in random order; allocation is popping, and deallocation is pushing plus a random swap (§4.2).

Shuffle vectors impose far less space overhead than random probing. First, with a maximum of 256 objects in a span, each offset in the vector can be represented as an unsigned character (a single byte). Second, because Mesh needs only one shuffle vector per attached MiniHeap, the amount of memory required for vectors is $256c$, where c is the number of size classes (24 in the current implementation): roughly 2.8K per thread. Finally, shuffle vectors are only ever accessed from a single thread, and so do not require locks or atomic operations. While bitmaps must be operated on atomically (frees may originate at any time from other threads), shuffle vectors are only accessed from a single thread and do not require synchronization or cache-line flushes.

4.3 Thread Local Heaps

All malloc and free requests from an application start at the thread's local heap. Thread local heaps have shuffle vectors for each size class, a reference to the global heap, and their own thread-local random number generator.

Allocation requests are handled differently depending on the size of the allocation. If an allocation request is larger than 16K, it is forwarded to the global heap for fulfillment (§4.4). Allocation requests 16K and smaller are small object allocations and are handled directly by the shuffle vector for the size class corresponding to the allocation request, as in Figure 4a. If the shuffle vector is empty, it is refilled by requesting an appropriately sized MiniHeap from the global heap. This MiniHeap is a partially-full MiniHeap if one exists, or represents a freshly-allocated span if no partially full ones are available for reuse. Frees, as in Figure 4d, first check if the object is from an attached MiniHeap. If so, it is handled by the appropriate shuffle vector, otherwise it is passed to the global heap to handle.

4.4 Global Heap

The global heap allocates MiniHeaps for thread-local heaps, handles all large object allocations, performs non-local frees for both small and large objects, and coordinates meshing.

4.4.1 The Meshable Arena

The global heap allocates meshable spans and large objects from a single, global meshable arena. This arena contains two sets of bins for same-length spans — one set is for demand zero-ed spans (freshly `mmap`ped), and the other for used spans — and a mapping of page offsets from the start of the arena to their owning MiniHeap pointers. Used pages are not immediately returned to the OS as they are likely to be needed again soon, and reclamation is relatively expensive. Only after 64MB of used pages have accumulated, or whenever meshing is invoked, Mesh returns pages to OS by calling `fallocate` on the heap's file descriptor (§4.5.1) with the `FALLOC_FL_PUNCH_HOLE` flag.

4.4.2 MiniHeap Allocation

Allocating a MiniHeap of size k pages begins with requesting k pages from the meshable arena. The global allocator then allocates and initializes a new MiniHeap instance from an internal allocator that Mesh uses for its own needs. This MiniHeap is kept live so long as the number of allocated objects remains non-zero, and singleton MiniHeaps are used to account for large object allocations. Finally, the global allocator updates the mapping of offsets to MiniHeaps for each of the k pages to point at the address of the new MiniHeap.

4.4.3 Large Objects

All large allocation requests (greater than 16K) are directly handled by the global heap. Large allocation requests are rounded up to the nearest multiple of the hardware page size (4K on x86_64), and a MiniHeap for 1 object of that size is requested, as detailed above. The start of the span tracked by that MiniHeap is returned to the program as the result of the malloc call.

```
void *MeshLocal::malloc(size_t sz) {
  int szClass;
  // forward to global heap if large
  if (!getSizeClass(sz, &szClass))
    return _global->malloc(sz);
  auto shufVec = _shufVecs[szClass];
  if (shufVec.isExhausted()) {
    shufVec.detachMiniheap();
    shufVec.attach(
      _global->allocMiniheap(szClass));}
  return shufVec.malloc();
}

void ShuffleVector::attach(MiniHeap *mh){
  _mh = mh;
  _off = maxCount();
  for (auto i = 0; i < maxCount(); i++){
    // true if atomically set (0 -> 1)
    if (bitmap.tryToSet(i)) {
      _list[_off--] = i;
    } }
  shuffle(_list[_off],
          _list[maxCount()]);
}

void *ShuffleVector::malloc() {
  const auto off = _list[_off++];
  return _spanStart + off * _objSize;
}

void MeshLocal::free(void *ptr) {
  // check if in attached MiniHeap
  for (auto i=0; i<SizeClassCount; i++){
    const auto curr = _shufVecs[i];
    if (curr->contains(ptr)) {
      curr->free(ptr);
      return; } }
  _global->free(ptr); // general case
}

void ShuffleVector::free(void *ptr) {
  const auto freedOff = getOff(ptr);
  _list[--_off] = freedOff;
  // place newly freed address
  // randomly in the shuffle vector
  auto swapOff =
    _rng.inRange(_off, maxCount() - 1);
  swap(_list[_off], _list[swapOff]);
}
```

Figure 4. Pseudocode for Mesh's core allocation and deallocation routines.

4.4.4 Non-local Frees

If `free` is called on a pointer that is not contained in an attached MiniHeap for that thread, the free is handled by the global heap. Non-local frees occur when the thread that

frees the object is different from the thread that allocated it, or if there have been sufficient allocations on the current thread that the original MiniHeap was exhaused and a new MiniHeap for that size class was attached.

Looking up the owning MiniHeap for a pointer is a constant time operation. The pointer is checked to ensure it falls within the arena, the arena start address is subtracted from it, and the result is divided by the page size. The resulting offset is then used to index into a table of MiniHeap pointers. If the result is zero, the pointer is invalid; otherwise, it points to a live MiniHeap. This enables us to catch certain application-level memory management errors, like non-local double frees when a new object hasn't been allocated at the same address.

Once the owning MiniHeap has been found, that MiniHeap's bitmap is updated atomically in a compare-and-set loop. If a free occurs for an object where the owning MiniHeap is attached to a different thread, the free atomically updates that MiniHeap's bitmap, but does not update the other thread's corresponding shuffle vector.

4.5 Meshing

Mesh's implementation of meshing is guided by theoretical results (described in detail in Section 5) that enable it to efficiently find a number of spans that can be meshed.

Meshing is rate limited by a configurable parameter, settable at program startup and during runtime by the application through the semi-standard `mallctl` API. The default rate meshes at most once every tenth of a second. If the last meshing freed less than one MB of heap space, the timer is not restarted until a subsequent allocation is freed through the global heap. This approach ensures that Mesh does not waste time searching for meshes when the application and heap are in a steady state.

We implement the SplitMesher algorithm from Section 3 in C++ to find meshes. Meshing proceeds one size class at a time. Pairs of mesh candidates found by SplitMesher are recorded in a list, and after SplitMesher returns candidate pairs are meshed together *en masse*.

Meshing spans together is a two step process. First, Mesh consolidates objects onto a single physical span. This consolidation is straightforward: Mesh copies objects from one span into the free space of the other span, and updates MiniHeap metadata (like the allocation bitmap). Importantly, as Mesh copies data at the physical span layer, even though objects are moving in memory, no pointers or data internal to moved objects or external references need to be updated. Finally, Mesh updates the process's virtual-to-physical mappings to point all meshed virtual spans at the consolidated physical span.

Physical memory reclaimed from meshing in one size class is able to be used to satisfy future allocations in other size classes.

4.5.1 Page Table Updates

Mesh updates the process's page tables via calls to `mmap`. We exploit the fact that `mmap` lets the same offset in a file (corresponding to a physical span) be mapped to multiple addresses. Mesh's arena, rather than being an anonymous mapping, as in traditional `malloc` implementations, is instead a shared mapping backed by a temporary file. This temporary file is obtained via the `memfd_create` system call and only exists in memory or on swap.

4.5.2 Concurrent Meshing

Meshing takes place concurrently with the normal execution of other program threads with *no* stop-the-world phase required. This is similar to how concurrent relocation is implemented in low-latency garbage collector algorithms like Pauseless and C4 [5, 34], as described below. Mesh maintains two invariants throughout the meshing process: reads of objects being relocated are always correct and available to concurrently executing threads, and objects are never written to while being relocated between physical spans. The first invariant is maintained through the atomic semantics of `mmap`, the second through a write barrier.

Mesh's write barrier is implemented with page protections and a segfault trap handler. Before relocating objects, Mesh calls `mprotect` to mark the virtual page where objects are being copied from as read-only. Concurrent reads succeed as normal. If a concurrent thread tries to write to an object being relocated, a Mesh-controlled segfault signal handler is invoked by a combination of the hardware and operating system. This handler waits on a lock for the current meshing operation to complete, the last step of which is remapping the source virtual span as read/write. Once meshing is done the handler checks if the address that triggered the segfault was involved in a meshing operation; if so, the handler exits and the instruction causing the write is re-executed by the CPU as normal against the fully relocated object.

4.5.3 Concurrent Allocation

All thread-local allocation (on threads other than the one running SplitMesher) can proceed concurrently and independently with meshing, until and unless a thread needs a fresh span to allocate from. Allocation only is performed from spans owned by a thread, and only spans owned by the global manager are considered for meshing; spans have a single owner. The thread running SplitMesher holds the global heap's lock while meshing. This lock also synchronizes transferring ownership of a span from the global heap to a thread-local heap (or vice-versa). If another thread requires a new span to fulfill an allocation request, the thread waits until the global manager finishes meshing and releases the lock.

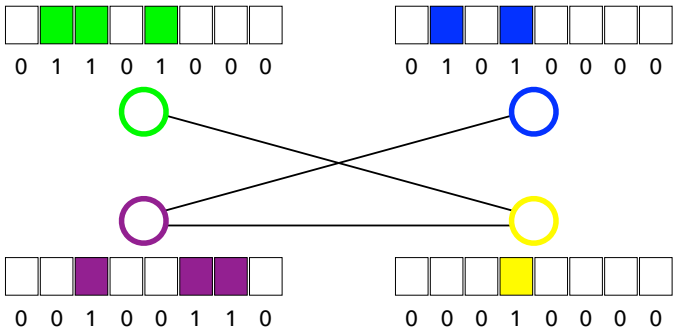


Figure 5. An example meshing graph. Nodes correspond to the spans represented by the strings `01101000`, `01010000`, `00100110`, and `00010000`. Edges connect meshable strings (corresponding to non-overlapping spans).

4.6 Huge Pages

Mesh's heap is not designed to be used in conjunction with transparent huge pages, where the page table size used by the kernel and hardware is 2MB rather than 4KB and the kernel runs a garbage collection-like daemon to coalesce 4KB pages into 2MB pages. Huge pages reduce TLB pressure, but necessarily increase the granularity at which the kernel manages physical memory on behalf of the process. This coarse granularity is fundamentally at odds with Mesh's focus on minimizing heap size. Additionally, the `madvise` mechanism that Mesh and other allocators like jemalloc use to return memory to the OS interacts poorly with transparent huge pages on Linux (causing 2MB pages to be split into 4KB pages), to the extent that major software vendors and operators recommend disabling transparent huge pages altogether [4, 6, 12, 25, 31]. Applications that need to back datasets or data structures with huge pages can still directly allocate (non-Mesh-managed) memory from Linux using one of several interfaces [17].

5 Analysis

This section shows that the SplitMesher procedure described in §3.3 comes with strong formal guarantees on the *quality* of the meshing found along with bounds on its *runtime*. In situations where significant meshing opportunities exist (that is, when compaction is most desirable), SplitMesher finds with high probability an approximation arbitrarily close to $1/2$ of the best possible meshing in $O(n/q)$ time, where n is the number of spans and q is the global probability of two spans meshing.

To formally establish these bounds on quality and runtime, we show that meshing can be interpreted as a graph problem, analyze its complexity (§5.1), show that we can do nearly as well by solving an easier graph problem instead (§5.2), and prove that SplitMesher approximates this problem with high probability (§5.3).

5.1 Formal Problem Definitions

Since Mesh segregates objects based on size, we can limit our analysis to compaction within a single size class without loss of generality. For our analysis, we represent spans as binary strings of length b, the maximum number of objects that the span can store. Each bit represents the allocation state of a single object. We represent each span π with string s such that $s(i) = 1$ if π has an object at offset i, and 0 otherwise.

Definition 5.1. We say two strings s_1, s_2 *mesh iff* $\sum_i s_1(i) \cdot s_2(i) = 0$. *More generally, a set of binary strings are said to mesh if every pair of strings in this set mesh.*

When we mesh k spans together, the objects scattered across those k spans are moved to a single span while retaining their offset from the start of the span. The remaining $k - 1$ spans are no longer needed and are released to the operating system. We say that we "release" $k - 1$ *strings* when we mesh k strings together. Since our goal is to empty as many physical spans as possible, we can characterize our theoretical problem as follows:

Problem 1. *Given a multi-set of n binary strings of length b, find a meshing that releases the maximum number of strings.*

A Formulation via Graphs: We observe that an instance of the meshing problem, a string multi-set S, can naturally be expressed via a graph $G(S)$ where there is a node for every string in S and an edge between two nodes iff the relevant strings can be meshed. Figure 5 illustrates this representation via an example.

If a set of strings are meshable, then there is an edge between every pair of the corresponding nodes: the set of corresponding nodes is a *clique*. We can therefore decompose the graph into k disjoint cliques iff we can free $n - k$ strings in the meshing problem. Unfortunately, the problem of decomposing a graph into the minimum number of disjoint cliques (MinCliqueCover) is in general NP-hard. Worse, it cannot even be approximated up to a factor $m^{1-\epsilon}$ unless $P = NP$ [35].

While the meshing problem is reducible to MinCliqueCover, we have not shown that the meshing problem is NP-Hard. The meshing problem is indeed NP-hard for strings of arbitrary length, but in practice string length is proportional to span size, which is constant.

Theorem 5.2. *The meshing problem for S, a multi-set of strings of constant length, is in P.*

Proof. We assume without loss of generality that S does not contain the all-zero string s_0; if it does, since s_0 can be meshed with any other string and so can always be released, we can solve the meshing problem for $S \setminus s_0$ and then mesh each instance of s_0 arbitrarily.

Rather than reason about MinCliqueCover on a meshing graph G, we consider the equivalent problem of coloring the complement graph $\bar{G}$ in which there is an edge between

every pair of two nodes whose strings do not mesh. The nodes of $\bar{G}$ can be partitioned into at most $2^b - 1$ subsets $N_1 \ldots N_{2^b-1}$ such that all nodes in each N_i represent the same string s_i. The induced subgraph of N_i in $\bar{G}$ is a clique since all its nodes have a 1 in the same position and so cannot be pairwise meshed. Further, all nodes in N_i have the same set of neighbors.

Since N_i is a clique, at most one node in N_i may be colored with any color. Fix some coloring on $\bar{G}$. Swapping the colors of two nodes in N_i does not change the validity of the coloring since these nodes have the same neighbor set. We can therefore unambiguously represent a valid coloring of $\bar{G}$ merely by indicating in which cliques each color appears.

With 2^b cliques and a maximum of n colors, there are at most $(n+1)^c$ such colorings on the graph where $c = 2^{2^b}$. This follows because each color used can be associated with a subset of $\{1, \ldots, 2^b\}$ corresponding to which of the cliques have node with this color; we call this subset a *signature* and note there are c possible signatures. A coloring can be therefore be associated with a multi-set of possible signatures where each signature has multiplicity between 0 and n; there are $(n+1)^c$ such multi-sets. This is polynomial in n since b is constant and hence c is also constant. So we can simply check each coloring for validity (a coloring is valid iff no color appears in two cliques whose string representations mesh). The algorithm returns a valid coloring with the lowest number of colors from all valid colorings discovered. □

Note that the runtime of the above algorithm is at least exponential in the string length. While technically polynomial for constant string length, the running time of the above algorithm would obviously be prohibitive in practice and so we never employ it in Mesh. Fortunately, as we show next, we can exploit the randomness in the strings to design a much faster algorithm.

5.2 Simplifying the Problem: From MinCliqueCover to Matching

We leverage Mesh's random allocation to simplify meshing; this random allocation implies a distribution over the graphs that exhibits useful structural properties. We first make the following important observation:

Observation 1. *Conditioned on the occupancies of the strings, edges in the meshing graph are not three-wise independent.*

To see that edges are not three-wise independent consider three random strings s_1, s_2, s_3 of length 4, each with exactly 2 ones. It is impossible for these strings to all mesh mutually since if we know that s_1 and s_2 mesh, and that s_2 and s_3 mesh, we know for certain that s_1 and s_3 cannot mesh. More generally, conditioning on s_1 and s_2 meshing and s_1 and s_3 meshing decreases the probability that s_1 and s_3 mesh. Below, we quantify this effect to argue that we can mesh near-optimally by solving the much easier Matching problem on the meshing graph (i.e., restricting our attention to finding cliques of size 2) instead of MinCliqueCover. Another consequence of the above observation is that we will not be able to appeal to theoretical results on the standard model of random graphs, *Erdős-Renyi graphs*, in which each possible edge is present with some fixed probability and the edges are fully independent. Instead we will need new algorithms and proofs that only require independence of acyclic collections of edges.

Triangles and Larger Cliques are Uncommon. Because of the dependencies across the edges present in a meshing graph, we can argue that *triangles* (and hence also larger cliques) are relatively infrequent in the graph and certainly less frequent than one would expect were all edges independent. For example, consider three strings $s_1, s_2, s_3 \in \{0,1\}^b$ with occupancies r_1, r_2, and r_3, respectively. The probability they mesh is

$$\binom{b-r_1}{r_2} \Big/ \binom{b}{r_2} \times \binom{b-r_1-r_2}{r_3} \Big/ \binom{b}{r_3}.$$

This value is significantly less than would have been the case if the events corresponding to pairs of strings being meshable were independent. For instance, if $b = 32, r_1 = r_2 = r_3 = 10$, this probability is so low that even if there were 1000 strings, the expected number of triangles would be less than 2. In contrast, had all meshes been independent, with the same parameters, there would have been 167 triangles.

The above analysis suggests that we can focus on finding only cliques of size 2, thereby solving Matching instead of MinCliqueCover. The evaluation in Section 6 vindicates this approach, and we show a strong accuracy guarantee for Matching below.

5.3 Theoretical Guarantees

Since we need to perform meshing at runtime, it is essential that our algorithm for finding strings to mesh be as efficient as possible. It would be far too costly in both time and memory overhead to actually construct the meshing graph and run an existing matching algorithm on it. Instead, the SplitMesher algorithm (shown in Figure 2) performs meshing without the need for explicitly constructing the meshing graph.

For further efficiency, we need to constrain the value of the parameter t, which controls Mesh's space-time tradeoff. If t were set as large as n, then SplitMesher could, in the worst case, exhaustively search all pairs of spans between the left and right sets: a total of $n^2/4$ probes. In practice, we want to choose a significantly smaller value for t so that Mesh can always complete the meshing process quickly without the need to search all possible pairs of strings.

Lemma 5.3. *Let $t = k/q$ where $k > 1$ is some user defined parameter and q is the global probability of two spans meshing.*

SplitMesher *finds a matching of size at least* $n(1-e^{-2k})/4$ *between the left and right span sets with probability approaching 1 as* $n \geq 2k/q$ *grows.*

Proof. Let $S_l = \{v_1, v_2, \ldots v_{n/2}\}$ and $S_r = \{u_1, u_2, \ldots u_{n/2}\}$. Let $t = k/q$ where $k > 1$ is some arbitrary constant. For $u_i \in S_l$ and $i \leq j \leq j + t$, we say (u_i, v_j) is a *good match* if all the following properties hold: (1) there is an edge between u_i and v_j, (2) there are no edges between u_i and $v_{j'}$ for $i \leq j' < j$, and (3) there are no edges between $u_{i'}$ and v_j for $i < i' \leq j$.

We observe that SplitMesher finds any good match, although it may also find additional matches. It therefore suffices to consider only the number of good matches. The probability (u_i, v_j) is a good match is $q(1-q)^{2(j-i)}$ by appealing to the fact that the collection of edges under consideration is acyclic. Hence, $\Pr(u_i \text{ has a good match})$ is

$$r := q \sum_{i=0}^{k/q-1} (1-q)^{2i} = q\frac{1-(1-q)^{2k/q}}{1-(1-q)^2} > \frac{1-e^{-2k}}{2} .$$

To analyze the number of good matches, define $X_i = 1$ iff u_i has a good match. Then, $\sum_i X_i$ is the number of good matches. By linearity of expectation, the expected number of good matches is $rn/2$. We decompose $\sum_i X_i$ into

$$Z_0 + Z_1 + \ldots + Z_{t-1} \quad \text{where} \quad Z_j = \sum_{i \equiv j \bmod t} X_i .$$

Since each Z_j is a sum of $n/(2t)$ independent variables, by the Chernoff bound, $P\left(Z_j < (1-\epsilon)\,E[Z_j]\right) \leq \exp\left(-\epsilon^2 rn/(4t)\right)$. By the union bound,

$$P\left(X < (1-\epsilon)\,rn/2\right) \leq t \exp\left(-\epsilon^2 rn/(4t)\right)$$

and this becomes arbitrarily small as n grows. □

In the worst case, the algorithm checks $nk/2q$ pairs. For our implementation of Mesh, we use a static value of $t = 64$; this value enables the guarantees of Lemma 5.1 in cases where significant meshing is possible. As Section 6 shows, this value for t results in effective memory compaction with modest performance overhead.

5.4 Summary of Analytical Results

We show the problem of meshing is reducible to a graph problem, MinCliqueCover. While solving this problem is infeasible, we show that probabilistically, we can do nearly as well by finding the maximum Matching, a much easier graph problem. We analyze our meshing algorithm as an approximation to the maximum matching on a random meshing graph, and argue that it succeeds with high probability. As a corollary of these results, Mesh breaks the Robson bounds with high probability.

6 Evaluation

Our evaluation answers the following questions: Does Mesh reduce overall memory usage with reasonable performance overhead? (§6.2) Does randomization provide empirical benefits beyond its analytical guarantees? (§6.3)

6.1 Experimental Setup

We perform all experiments on a MacBook Pro with 16 GiB of RAM and an Intel i7-5600U, running Linux 4.18 and Ubuntu Bionic. We use glibc 2.26 and jemalloc 3.6.0 for SPEC2006, Redis 4.0.2, and Ruby 2.5.1. Two builds of Firefox 57.0.4 were compiled as release builds, one with its internal allocator disabled to allow the use of alternate allocators via `LD_PRELOAD`. SPEC was compiled with clang version 4.0 at the `-O2` optimization level, and Mesh was compiled with gcc 8 at the `-O3` optimization level and with link-time optimization (`-flto`). We primarily compare Mesh results to the default allocator for each test (jemalloc for Firefox and Redis, glibc for SPEC and Ruby). Where appropriate, we also include results from tcmalloc (gperftools 2.5) and Hoard (git commit `a0e46aa1`); when omitted, it is because their performance is virtually identical to jemalloc and glibc.

Measuring memory usage: To accurately measure the memory usage of an application over time, we developed a Linux-based utility, `mstat`[1], that runs a program in a new memory control group [26]. `mstat` polls the resident-set size (RSS) and kernel memory usage statistics for all processes in the control group at a constant frequency. This enables us to account for the memory required for larger page tables (due to meshing) in our evaluation. We have verified that `mstat` does not perturb performance results.

6.2 Memory Savings and Performance Overhead

We evaluate Mesh's impact on memory consumption and runtime across the Firefox web browser, the Redis data structure store, and the SPECint2006 benchmark suite.

6.2.1 Firefox

Firefox is an especially challenging application for memory reduction since it has been the subject of a five year effort to reduce its memory footprint [33]. To evaluate Mesh's impact on Firefox's memory consumption under realistic conditions, we measure Firefox's RSS while running the Speedometer 2.0 benchmark. Speedometer was constructed by engineers working on the Google Chrome and Apple Safari web browsers to simulate the patterns in use on websites today, stressing a number of browser subsystems like DOM APIs, layout, CSS resolution and the JavaScript engine. In Firefox, most of these subsystems are multi-threaded, even for a single page [13]. The benchmark comprises a number of small "todo" apps written in a number of different languages and styles, with a final score computed as the geometric mean of the time taken by the executed tests.

We test Firefox in single-process mode (disabling content sandboxing, which spawns multiple processes) under the

[1] `mstat` is open source, and available at https://github.com/bpowers/mstat

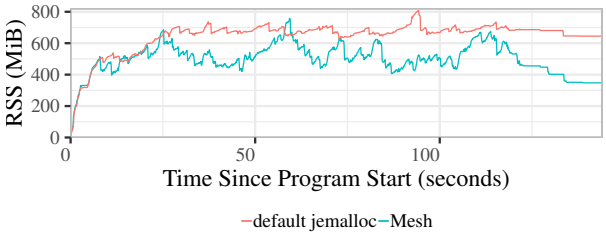


Figure 6. Firefox: Mesh decreases mean heap size by 16% over the course of the Speedometer 2.0 benchmark compared with the version of jemalloc bundled with Firefox, with less than a 1% change in the reported Speedometer score (§6.2.1).

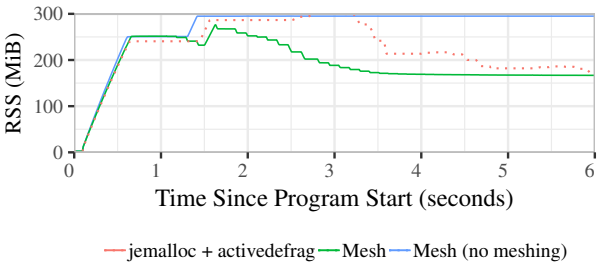


Figure 7. Redis: Mesh automatically achieves significant memory savings (39%), obviating the need for its custom, application-specific "defragmentation" routine (§6.2.2).

`mstat` tool to record memory usage over time. Our test opens a tab, loads the Speedometer page from a local server, waits 2 seconds, and then automatically executes the test. We record the reported score at the end of the benchmark run and calculate average memory usage recorded by `mstat`. We tested both a standard release build of Firefox, along with a release build that did not bundle Mozilla's fork of jemalloc (hereafter referred to as `mozjemalloc`) and instead directly called `malloc`-related functions, with Mesh included via `LD_PRELOAD`. We report the average resident set size over the course of the benchmark and a 15 second cooldown period afterward, collecting three runs per allocator.

Mesh reduces the memory consumption of Firefox by 16% compared to Firefox's bundled jemalloc allocator. Mesh requires 530 MB (σ = 22.4 MB) to complete the benchmark, while the Mozilla allocator needs 632 MB (σ = 25.3 MB). Mesh spent a total of 135 ms meshing over the course of the benchmark, with a maximum meshing latency of 7.5 ms and average meshing latency of 0.2 ms. This result shows that Mesh can effectively reduce memory overhead even in widely used and heavily optimized applications. Mesh achieves this savings with less than a 1% reduction in performance (measured as the score reported by Speedometer). Hoard and tcmalloc improved Speedometer performance relative to jemalloc by 5.4 and 6.0% respectively, while increasing average heap size by 48.0% and 8.6%.

Figure 6 shows memory usage over the course of a Speedometer benchmark run under Mesh and the default jemalloc allocator. While memory usage under both peaks to similar levels, Mesh is able to keep heap size consistently lower.

6.2.2 Redis

Redis is a widely-used in-memory data structure server. Redis 4.0 introduced a feature called "active defragmentation" [29, 32]. Redis calculates a fragmentation ratio (RSS over sum of active allocations) once a second. If this ratio is too high, it triggers a round of active defragmentation. This involves making a fresh copy of Redis's internal data structures and freeing the old ones. Active defragmentation relies on allocator-specific APIs in jemalloc both for gathering statistics and for its ability to perform allocations that bypass thread-local caches, increasing the likelihood they will be contiguous in memory.

We adapt a benchmark from the official Redis test suite to measure how Mesh's automatic compaction compares with Redis's active defragmentation, as well as against the standard glibc allocator. This benchmark runs for a total of 7.5 seconds, regardless of allocator. It configures Redis to act as an LRU cache with a maximum of 100 MB of objects (keys and values). The test then allocates 700,000 random keys and values, where the values have a length of 240 bytes. Finally, the test inserts 170,000 new keys with values of length 492. Our only change from the original Redis test is to increase the value sizes in order to place all allocators on a level playing field with respect to *internal* fragmentation; the chosen values of 240 and 492 bytes ensure that tested allocators use similar size classes for their allocations. We test Mesh with Redis in two configurations: with meshing always on and with meshing disabled, both without any input or coordination from the redis-server application.

Figure 7 shows memory usage over time for Redis under Mesh, as well as under jemalloc with Redis's "activedefrag" enabled, as measured by `mstat` (§6.1). The "activedefrag" configuration enables active defragmentation after all objects have been added to the cache.

Using Mesh automatically and portably achieves the same heap size reduction (39%) as Redis's active defragmentation. During most of the 7.5s of this test Redis is idle; Redis only triggers active defragmentation during idle periods. With Mesh, insertion takes 1.76s, while with Redis's default of jemalloc, insertion takes 1.72s. Redis under Hoard and tcmalloc has the same average heap size as Mesh with meshing disabled (under 2% difference), and both allocators are similarly within 2% of the insertion speed of jemalloc. Mesh's compaction is additionally *significantly* faster than Redis's active defragmentation. During execution with Mesh, a total of 0.23s are spent meshing (the longest pause is 22 ms), while active defragmentation accounts for 1.49s (5.5× slower). This high latency may explain why Redis disables "activedefrag" by default.

6.2.3 SPEC Benchmarks

Most of the SPEC benchmarks are not particularly compelling targets for Mesh because they have small overall footprints and do not exercise the memory allocator. For example, while the `401.bzip2` benchmark has one of the higher heap size averages at 665 MB, the difference in both runtime and average heap size between the fastest + slowest allocators is under 1%.

Across the entire SPECint 2006 benchmark suite, Mesh modestly decreases average memory consumption (geomean: −2.4%) vs. glibc, while imposing minimal execution time overhead (geomean: 0.7%).

However, for applications that are both allocation-intensive (many calls to `malloc` and `free`) and which have large footprints, Mesh is able to substantially reduce peak memory consumption. The most allocator-sensitive benchmark is `400.perlbench`, a Perl benchmark that performs a number of e-mail related tasks including spam detection (SpamAssassin). Peak RSS with glibc, jemalloc, and Hoard is 664 MB, 614 MB, and 732 MB respectively. Mesh reduces peak RSS to 564 MB (a 15% reduction relative to glibc) while increasing runtime overhead by only 3.9%.

6.3 Empirical Value of Randomization

Randomization is key to Mesh's analytic guarantees; we evaluate whether it also can have an observable empirical impact on its ability to reclaim space. To do this, we test three configurations of Mesh: (1) meshing disabled, (2) meshing enabled but randomization disabled, and (3) Mesh with both meshing and randomization enabled (the default).

We tested these configurations with Firefox and Redis, and found no significant differences when randomization was disabled; we believe that this is due to the highly irregular (effectively random) allocation patterns that these applications exhibit. We hypothesized that a more regular allocation pattern would be more challenging for a non-randomized baseline. To test this hypothesis, we wrote a synthetic microbenchmark with a regular allocation pattern in Ruby. Ruby is an interpreted programming language popular for implementing web services, including GitHub, AirBnB, and the original version of Twitter. Ruby makes heavy use of object-oriented and functional programming paradigms, making it allocation-intensive. Ruby is garbage collected, and while the standard MRI Ruby implementation (written in C) has a custom GC arena for small objects, large objects (like strings) are allocated directly with `malloc`.

Our Ruby microbenchmark repeatedly performs a sequence of string allocations and deallocations, simulating the effect of accumulating results from an API and periodically filtering some out. It allocates a number of strings of a fixed size, then retaining references 25% of the strings while dropping references to the rest. Each iteration the length of the strings is doubled. The test requires only a fixed 128 MB to hold the string contents.

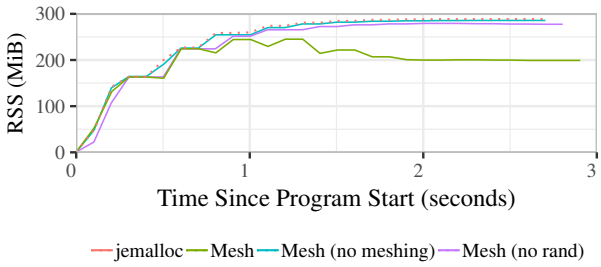


Figure 8. Ruby benchmark: Mesh is able to decrease mean heap size by 18% compared to Mesh with randomization disabled and non-compacting allocators (§6.3).

Figure 8 presents the results of running this application with the three variants of Mesh and jemalloc; for this benchmark, jemalloc and glibc are essentially indistinguishable. With meshing disabled, Mesh exhibits similar runtime and heap size to jemalloc. With meshing enabled but randomization disabled, Mesh imposes a 4% runtime overhead and yields only a modest 3% reduction in heap size.

Enabling randomization in Mesh increases the time overhead to 10.7% compared to jemalloc, but the use of randomization lets it significantly reduce the mean heap size over the execution time of the microbenchmark (a 19% reduction). The additional runtime overhead is due to the additional system calls and memory copies induced by the meshing process. This result demonstrates that randomization is not just useful for providing analytical guarantees but can also be essential for meshing to be effective in practice.

6.4 Summary of Empirical Results

For a number of memory-intensive applications, including aggressively space-optimized applications like Firefox, Mesh can substantially reduce memory consumption (by 16% to 39%) while imposing a modest impact on runtime performance (e.g., around 1% for Firefox and SPECint 2006). We find that Mesh's randomization can enable substantial space reduction in the face of a regular allocation pattern.

7 Related Work

Hound: Hound is a memory leak detector for C/C++ applications that introduced meshing (a.k.a. "virtual compaction"), a mechanism that Mesh leverages [28]. Hound combines an age-segregated heap with data sampling to precisely identify leaks. Because Hound cannot reclaim memory until every object on a page is freed, it relies on a heuristic version of meshing to prevent catastrophic memory consumption. Hound is unsuitable as a replacement general-purpose allocator; it lacks both Mesh's theoretical guarantees and space and runtime efficiency (Hound's repository is missing files and it does not build, precluding a direct empirical comparison here). The Hound paper reports a geometric mean

slowdown of ≈ 30% for SPECint2006 (compared to Mesh's 0.7%), slowing one benchmark (`xalancbmk`) by almost 10×. Hound also generally *increases* memory consumption, while Mesh often substantially decreases it.

Compaction for C/C++: Previous work has described a variety of manual and compiler-based approaches to support compaction for C++. Detlefs shows that if developers use annotations in the form of smart pointers, C++ code can also be managed with a relocating garbage collector [10]. Edelson introduced GC support through a combination of automatically generated smart pointer classes and compiler transformations that support relocating GC [11]. Google's Chrome uses an application-specific compacting GC for C++ objects called Oilpan that depends on the presence of a single event loop [1]. Developers must use a variety of smart pointer classes instead of raw pointers to enable GC and relocation. This effort took years. Unlike these approaches, Mesh is fully general, works for unmodified C and C++ binaries, and does not require programmer or compiler support; its compaction approach is orthogonal to GC.

CouchDB and Redis implement *ad hoc* best-effort compaction, which they call "defragmentation". These work by iterating through program data structures like hash tables, copying each object's contents into freshly-allocated blocks (in the hope they will be contiguous), updating pointers, and then freeing the old objects [29, 32]. This application-specific approach is not only inefficient (because it may copy objects that are already densely packed) and brittle (because it relies on internal allocator behavior that may change in new releases), but it may also be ineffective, since the allocator cannot ensure that these objects are actually contiguous in memory. Unlike these approaches, Mesh performs compaction efficiently and its effectiveness is guaranteed.

Compacting garbage collection in managed languages: Compacting garbage collection has long been a feature of languages like LISP and Java [15, 18]. Contemporary runtimes like the Hotspot JVM [24], the .NET VM [23], and the SpiderMonkey JavaScript VM [9] all implement compaction as part of their garbage collection algorithms. Mesh brings the benefits of compaction to C/C++; in principle, it could also be used to automatically enable compaction for language implementations that rely on non-compacting collectors.

Bounds on Partial Compaction: Cohen and Petrank prove upper and lower bounds on defragmentation via partial compaction [7, 8]. In their setting, corresponding to managed environments, *every* object *may* be relocated to any free memory location; they ask what space savings can be achieved if the memory manager is only allowed to relocate a bounded number of objects. By contrast, Mesh is designed for unmanaged languages where objects *cannot* be arbitrarily relocated.

PCM fault mitigation: Ipek *et al.* use a technique similar to meshing to address the degradation of phase-change memory (PCM) over the lifetime of a device [20]. The authors introduce dynamically replicated memory (DRM), which uses pairs of PCM pages with non-overlapping bit failures to act as a single page of (non-faulty) storage. When the memory controller reports a page with new bit failures, the OS attempts to pair it with a complementary page. A random graph analysis is used to justify this greedy algorithm.

DRM operates in a qualitatively different domain than Mesh. In DRM, the OS occasionally attempts to pair newly faulty pages against a list of pages with static bit failures. This process is incremental and local. In Mesh, the occupancy of spans in the heap is more dynamic and much less local. Mesh solves a full, non-incremental version of the meshing problem each cycle. Additionally, in DRM, the random graph describes an error model rather than a design decision; additionally, the paper's analysis is flawed. The paper erroneously claims that the resulting graph is a simple random graph; in fact, its edges are not independent (as we show in §5.2). This invalidates the claimed performance guarantees, which depend on properties of simple random graphs. In contrast, we prove the efficacy of our original SplitMesher algorithm for Mesh using a careful random graph analysis.

8 Conclusion

This paper introduces Mesh, a memory allocator that efficiently performs *compaction without relocation* to save memory for unmanaged languages. We show analytically that Mesh provably avoids catastrophic memory fragmentation with high probability, and empirically show that Mesh can substantially reduce memory fragmentation for memory-intensive applications written in C/C++ with low runtime overhead. In future work, we plan to explore integrating Mesh into language runtimes that do not currently support compaction, such as Go and Rust.

We have released Mesh as an open source project; it can be used with arbitrary C and C++ Linux and Mac OS X binaries and can be downloaded at http://libmesh.org.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1637536. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Mads Adger. 2013. *[Blink-dev] Oilpan - experimenting with a garbage collected heap for the DOM.* http://bit.ly/2pwDhwk.

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international*

conference on Architectural support for programming languages and operating systems. ACM Press, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[3] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*. ACM Press, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

[4] Michele Casey. 2013. Performance Issues with Transparent Huge Pages (THP). https://blogs.oracle.com/linux/performance-issues-with-transparent-huge-pages-thp.

[5] Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/1064979.1064988

[6] Cloudera. 2019. Optimizing Performance in CDH. https://www.cloudera.com/documentation/enterprise/latest/topics/cdh_admin_performance.html.

[7] Nachshon Cohen and Erez Petrank. 2013. Limitations of Partial Compaction: Towards Practical Bounds. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 309–320. https://doi.org/10.1145/2491956.2491973

[8] Nachshon Cohen and Erez Petrank. 2017. Limitations of Partial Compaction: Towards Practical Bounds. *ACM Trans. Program. Lang. Syst.* 39, 1, Article 2 (March 2017), 44 pages. https://doi.org/10.1145/2994597

[9] Jon Coppeard. 2015. *Compacting Garbage Collection in SpiderMonkey*. https://mzl.la/2rntQlY.

[10] David Detlefs. 1992. Garbage Collection and Run-time Typing as a C++ Library. In *Proceedings of the 1992 USENIX C++ Conference*. USENIX Association, 37–56.

[11] Daniel R. Edelson. 1992. Precompiling C++ for Garbage Collection. In *Proceedings of the International Workshop on Memory Management (IWMM '92)*. Springer-Verlag, London, UK, UK, 299–314. http://dl.acm.org/citation.cfm?id=645648.664820

[12] Nelson Elhage. 2017. Disable Transparent Hugepages. https://blog.nelhage.com/post/transparent-hugepages/.

[13] Wikipedia: The Free Encyclopedia. 2018. Quantum (Mozilla). https://en.wikipedia.org/wiki/Quantum_(Mozilla).

[14] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[15] Robert R. Fenichel and Jerome C. Yochelson. 1969. A LISP Garbage-collector for Virtual-memory Computer Systems. *Commun. ACM* 12, 11 (Nov. 1969), 611–612. https://doi.org/10.1145/363269.363280

[16] Sanjay Ghemawat and Paul Menage. 2007. TCMalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[17] Mel Gorman. 2010. Huge pages part 2: Interfaces. https://lwn.net/Articles/375096/.

[18] Wilfred J. Hansen. 1969. Compact List Representation: Definition, Garbage Collection, and System Implementation. *Commun. ACM* 12, 9 (Sept. 1969), 499–507. https://doi.org/10.1145/363219.363225

[19] Kentaro Hara. 2017. *State of Blink's Speed*. https://docs.google.com/presentation/d/1Az-F3CamBq6hZ5QqQt-ynQEMWEhHY1VTvlRwL7b_6TU.

[20] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. 2010. Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories. *SIGPLAN Not.* 45, 3 (March 2010), 3–14. https://doi.org/10.1145/1735971.1736023

[21] Mark S. Johnstone and Paul R. Wilson. 1998. The Memory Fragmentation Problem: Solved?. In *Proceedings of the 1st International Symposium on Memory Management (ISMM '98)*. ACM, New York, NY, USA, 26–36. https://doi.org/10.1145/286860.286864

[22] Donald E Knuth. 1981. The Art of Programming, vol. 2, Semi-Numerical Algorithms.

[23] Rico Mariani. 2003. *Garbage Collector Basics and Performance Hints*. https://msdn.microsoft.com/en-us/library/ms973837.aspx.

[24] Sun Microsystems. 2006. Memory Management in the Java HotSpot™ Virtual Machine.

[25] Inc MongoDB. 2018. Disable Transparent Huge Pages (THP). https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/.

[26] Milan Navrátil, Marie Doleželová, Peter Ondrejka, Eva Majoršinová, Martin Prpič, Rüdiger Landmann, and Douglas Silas. 2017. *Introduction To Control Groups, Resource Management Guide*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[27] Gene Novark and Emery D. Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*. ACM, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

[28] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2009. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2009)*. ACM, New York, NY, USA, 397–407. https://doi.org/10.1145/1542476.1542521

[29] Dave Rigby. 2017. *[jemalloc] expose hints that will allow applications to perform runtime active defragmentation Bug 566*. https://github.com/jemalloc/jemalloc/issues/566.

[30] J. M. Robson. 1977. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J.* 20, 3 (1977), 242. https://doi.org/10.1093/comjnl/20.3.242

[31] Salvatore Sanfilippo. 2014. Redis latency spikes and the Linux kernel: a few more details. http://antirez.com/news/84.

[32] Salvatore Sanfilippo. 2017. Redis 4.0 GA is out! https://groups.google.com/forum/#!msg/redis-db/5Kh3viziYGQ/58TKLwX0AAAJ.

[33] John Schoenick. 2017. ARE WE SLIM YET? https://areweslimyet.com/faq.htm.

[34] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

[35] David Zuckerman. 2007. Linear Degree Extractors and the Inapproximability of Max Clique and Chromatic Number. *Theory of Computing* 3, 6 (2007), 103–128.