TCMalloc Design Review

By Dudcom

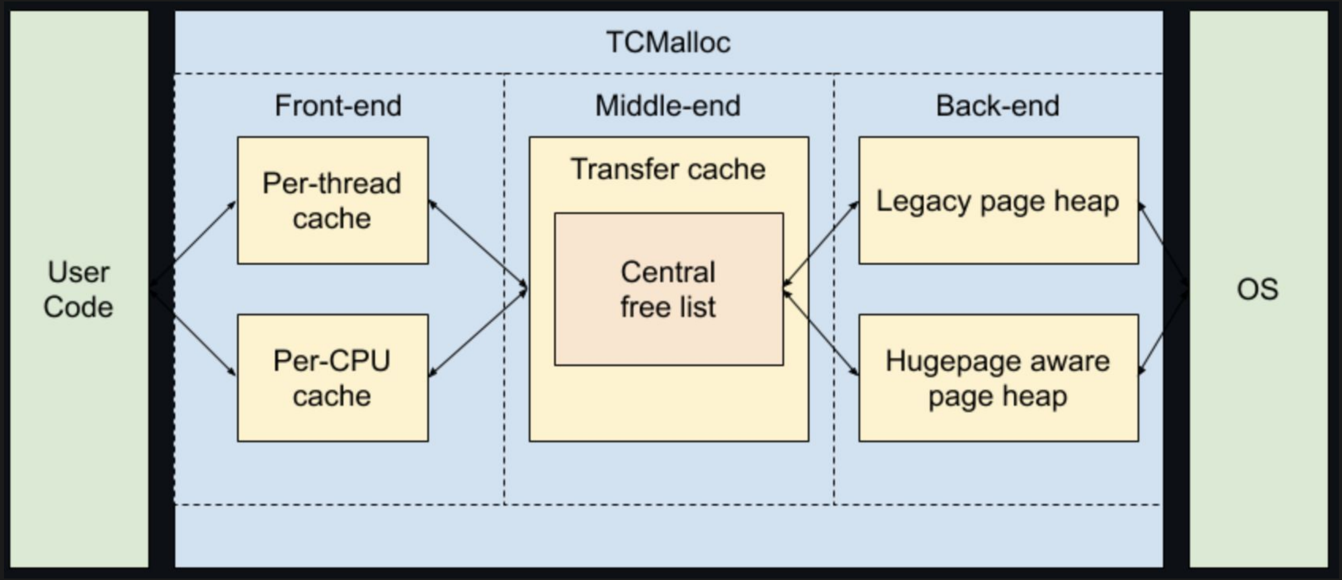
Month Year

Sections

2026

2026

3 Layer System for optimized hot paths



3 Layer System for optimized hot paths

Layer 1 (Front-End - Lock-Free)

- Most allocations use a lock-free path via per-thread cache (ThreadCache) or per-CPU cache (CpuCache) when there are many threads. This handles the majority of requests without locking.

Layer 2 (Transfer Cache - Shared, Lock-Protected)

- When the front-end cache is empty, it fetches from the Transfer Cache, a shared cache between threads/CPUs. This reduces contention compared to going directly to the Central Free List.

Layer 3 (Central Free List - Per Size Class, Lock-Protected)

- If the Transfer Cache is empty, it pulls from the Central Free List, which manages spans of pages for each size class. This happens less frequently (only when caches are empty).

Layer 4 (Page Allocator - Back-End)

- If the Central Free List needs memory, it requests pages from the Page Allocator. For large allocations (exceeding kMaxSize), the request bypasses all cache layers and goes directly to the Page Allocator.

Layer 5 (System Allocator)

- The Page Allocator ultimately requests memory from the OS through the System Allocator.

tcmalloc.cc - The starting point

```
void* __libc_malloc(size_t size) TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalMalloc);
void __libc_free(void* ptr) TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalFree);
void* __libc_realloc(void* ptr, size_t size) TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalRealloc);
void* __libc_calloc(size_t n, size_t size) TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalCalloc);
void __libc_cfree(void* ptr) TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalCfree);
void* __libc_memalign(size_t align, size_t s)
TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalMemalign);
void* __libc_valloc(size_t size) TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalValloc);
void* __libc_pvalloc(size_t size) TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalPvalloc);
int __posix_memalign(void** r, size_t a, size_t s)
TCMALLOC_NOTHROW
    TCMALLOC_ALIAS(TCMallocInternalPosixMemalign);
```

Here we see that the actual code starts by overriding the glibc start functions so we can just use the standard allocation functions ie malloc/free.

Bellow we also will override the following operators in order to redfien the standard c++ memory allocations :

```
void* operator new(size_t size) noexcept(false)
    TCMALLOC_ALIAS(TCMallocInternalNew);
void operator delete(void* p) noexcept
TCMALLOC_ALIAS(TCMallocInternalDelete);
void operator delete(void* p, size_t size) noexcept
    TCMALLOC_ALIAS(TCMallocInternalDeleteSized);
```

Fast_alloc Our Main Entry Point

```
template <typename Policy, typename Pointer = typename
Policy::pointer_type>
static inline Pointer ABSL_ATTRIBUTE_ALWAYS_INLINE fast_alloc(size_t
size,
                                                                Policy
policy) {
  const auto [is_small, size_class] =
      tc_globals.sizemap().GetSizeClass(policy, size);
  if (ABSL_PREDICT_FALSE(!is_small)) {
    SLOW_PATH_BARRIER();
    TCMALLOC_MUSTTAIL return slow_alloc_large(size, policy);
  }

  if
(ABSL_PREDICT_FALSE(!GetThreadSampler().TryRecordAllocationFast(size)))
{
    SLOW_PATH_BARRIER();
    return slow_alloc_small(size, size_class, policy);
  }
// final fast path
  void* ret = tc_globals.cpu_cache().AllocateFast(size_class);
  if (ABSL_PREDICT_FALSE(ret == nullptr)) {
    SLOW_PATH_BARRIER();
    return slow_alloc_small(size, size_class, policy);
  }

  TC_ASSERT_NE(ret, nullptr);
  return Policy::to_pointer(ret, size_class);
}
```

First we check to see if the allocation is greater than kMaxSize, this si done by checking seeing the boolean result of `tc_globals.sizemap().GetSizeClass(policy, size);` in which case the we will entry the slow path as defined by `slow_alloc_large`

Next we check `TryRecordAllocationFast()` against the size. This will tell our allocator if the requested size needs the following: /

- Gets the per-thread sampler (maintains bytes_until_sample_).
- Subtracts size + 1 from bytes_until_sample_ using __builtin_usubl_overflow.
- Returns true if no underflow (no sampling needed).
- Returns false if underflow (sampling needed).

The sampler: Bytes are randomly marked given a poisson point, we essentially will allocate till we hit such a point and in that case we will resample the system. The chance you hit a point is given by `1 - e^(-X/Y)` which in terms of stats == the more you allocate the more likely we are to sample.

Finally we hit the base case/fast path which should only be hit if the following are valid:
- the size does not exceed the maximum size (size class > 0)
- cpu / thread cache data has been initialized.
- the allocation is not subject to sampling / gwp-asan.
- no new/delete hook is installed and required to be called.

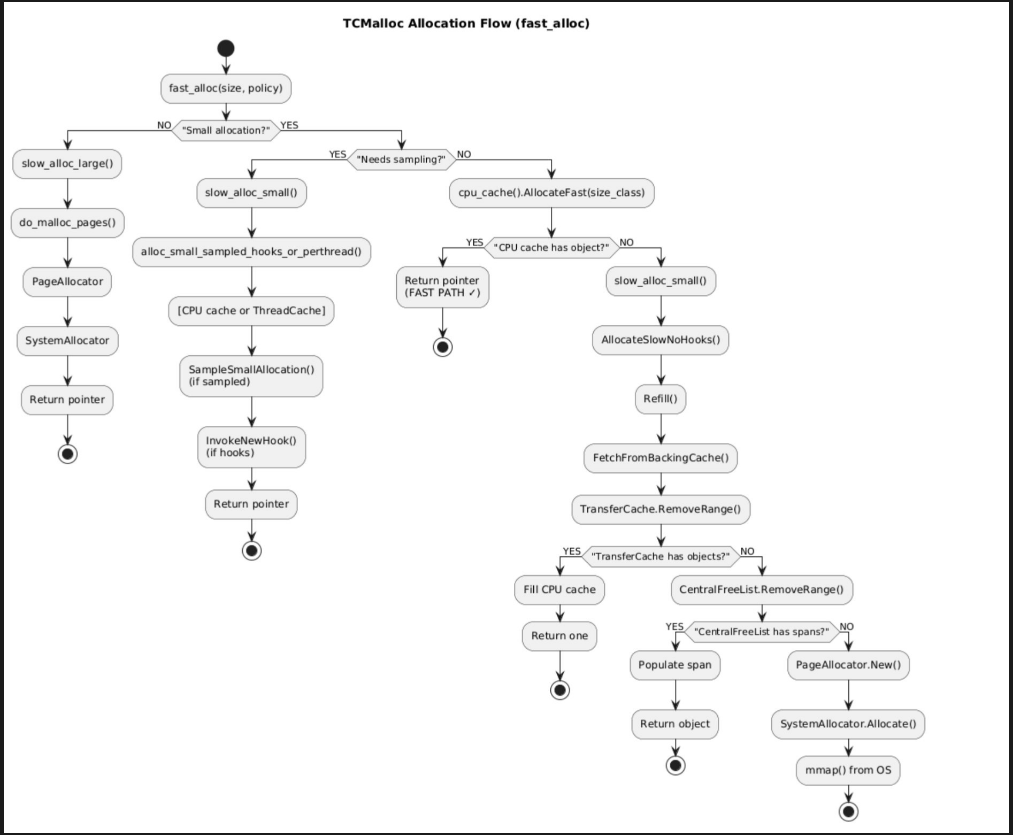
Layer 1 - CPU Fast Path

Key files: cpu_cache.h

```
template <class Forwarder>
inline ABSL_ATTRIBUTE_ALWAYS_INLINE void* CpuCache<Forwarder>::AllocateFast(
   size_t size_class) {
 TC_ASSERT_GT(size_class, 0);
 return freelist_.Pop(size_class);
}
```

- We are just popping from the freelist we want to access

After Fast Alloc Program Flow:



Layer 1 - Thread Cache Memory Allocation Design

Key files: thread_cache.h, thread_cache.cc

```
class ABSL_CACHELINE_ALIGNED ThreadCache {
public:
 explicit ThreadCache(pthread_t tid);
 void Cleanup();
 // ...
 // ...
 FreeList list_[kNumClasses];  // Array indexed by size-class
 size_t size_;       // Combined size of data
 size_t max_size_;  // size_ > max_size_ --> Scavenge()
 pthread_t tid_;
 bool in_setspecific_;
```

The FreeList is a LinkedList

- There is 1 per size class

```
inline ABSL_ATTRIBUTE_ALWAYS_INLINE  void*
ThreadCache::Allocate(
   size_t size_class) {
 const size_t allocated_size  =
tc_globals.sizemap().class_to_size(size_class);

 FreeList* list = &list_[size_class];
 void* ret;
 if (ABSL_PREDICT_TRUE(list->TryPop(&ret))) {
   size_ -= allocated_size;
   return ret;
 }

 return FetchFromTransferCache(size_class,
allocated_size);
}
```

Layer 2-Thread allocation (cpu alloc failed)

Key files: thread_cache.h, thread_cache.cc

```
ABSL_ATTRIBUTE_NOINLINE static
  typename Policy::pointer_type slow_alloc_small(size_t size,
                                                 uint32_t size_class,
                                                 Policy policy) {
size_t weight = GetThreadSampler().RecordedAllocationFast(size);
if (ABSL_PREDICT_FALSE(weight != 0) ||
    ABSL_PREDICT_FALSE(tcmalloc::tcmalloc_internal::Static::HaveHooks()) ||
    ABSL_PREDICT_FALSE(!UsePerCpuCache(tc_globals))) {
  return alloc_small_sampled_hooks_or_perthread(size, size_class, policy,
                                                weight);
}

void* res = tc_globals.cpu_cache().AllocateSlowNoHooks(size_class);
if (ABSL_PREDICT_FALSE(res == nullptr)) return policy.handle_oom(size);
return Policy::to_pointer(res, size_class);
}
```

```
weight = T + k - f

T = sample interval (e.g.,
512KB)
k = allocation size
f = bytes_until_sample_


weight == 0:
  -   Not sampled; no profiling
      needed
weight > 0:
  -   Sampled; this sample
      represents approximately
      weight bytes
```

Layer 1 - AllocateFast Execution Tree

We actually end up calling AllocateFast several times because its so fast (just a pop on LN list so O(1))

```
fast_alloc()
    ↓
[Try AllocateFast() - First Try] ← Line 1171
    ├─ SUCCESS → Return pointer
    |
    └─ FAIL (nullptr) → slow_alloc_small()
                         ↓
                       [Need sampling/hooks?]
                         ├─ NO → AllocateSlowNoHooks()
                         |
                         └─ YES → alloc_small_sampled_hooks_or_perthread()
                                    ↓
                                  [Try AllocateFast() - Second Try] ← Line 1074
                                    ├─ SUCCESS → Use it, then sample
                                    |
                                    └─ FAIL → AllocateSlow() or Thread Cache
```

Layer 1 - AllocateSlow (CPU slow path)

```
template <class Forwarder>
void* CpuCache<Forwarder>::AllocateSlowNoHooks(size_t size_class) {
  if (BypassCpuCache(size_class)) {
    return forwarder_.sharded_transfer_cache().Pop(size_class);
  }
  auto [cpu, cached] = CacheCpuSlab();
  if (ABSL_PREDICT_FALSE(cached)) {
    if (ABSL_PREDICT_FALSE(cpu < 0)) {
      // The cpu is stopped.
      void* ptr = nullptr;
      int r = FetchFromBackingCache(size_class,
absl::MakeSpan(&ptr, 1));
#ifndef NDEBUG
      TC_ASSERT(r == 1 || ptr == nullptr);
#else
      (void)r;
#endif
      return ptr;
    }
    if (void* ret = AllocateFast(size_class)) {
      return ret;
    }
  }
  RecordCacheMissStat(cpu, true);
  return Refill(cpu, size_class);
}
```

The idea here is that we will refill the CPU cache from the 2nd layer
Ie the central free list with a lock.

Here we see BypassCpuCache(size_class) check if the size of the request is to big and if it is we directly go to the shared transfer cache instead of checking if its empty

We cache the CPU's slab address (allowing allocate fast work on the next attempt).

Finally at the bottom we see us seeing of the AllocateFast works well and the return from there will hit the Refill if it needs to.

Layer 2 - CPU cache refill process/Accessing the central free list

```
template <class Forwarder>
inline int CpuCache<Forwarder>::FetchFromBackingCache(size_t size_class,
                                                      absl::Span<void*> batch) {
  if (UseBackingShardedTransferCache(size_class)) {
    return forwarder_.sharded_transfer_cache().RemoveRange(size_class, batch);
  }
  return forwarder_.transfer_cache().RemoveRange(size_class, batch);
}


template <class Forwarder>
inline bool CpuCache<Forwarder>::UseBackingShardedTransferCache(
    size_t size_class) const {
  // Make sure that the thread is registered with rseq.
  TC_ASSERT(subtle::percpu::IsFastNoInit());
  // We enable sharded cache as a backing cache for all size classes when
  // generic configuration is enabled.
  return forwarder_.sharded_transfer_cache().should_use(size_class) &&
         forwarder_.UseGenericShardedCache();
}
```

So the very first thing we do in the code here is call the boolean function `UseBackingShardedTransferCache` which will inform us weather or not we want to use the ShardedTransferCache or the regular TransferCache.

- A regular transfer cache will bypass the cpu cache and direct perform the allocation and this is used only for large allocations

The sharedTransferCache is faster in this case because

1. Reduced contention
   a. Each L3 cache domain has its own shard
2. Better NUMA locality
   a. Objects stay closer to the CPU that allocated them
3. Scalability
   a. Performance improves with more CPU cores

Well what do these functions actually do?

Layer-2 CPU Cache ShardedTransferCache tree

```
// Returns the cache shard corresponding to the given size class and the
 // current cpu's L3 node. The cache will be initialized if required.
 TransferCache& get_cache(int size_class) {
   const uint8_t shard_index =
       cpu_layout_->CpuShard(cpu_layout_->CurrentCpu());
   TC_ASSERT_LT(shard_index, num_shards_);
   Shard& shard = shards_[shard_index];
   absl::base_internal::LowLevelCallOnce(
       &shard.once_flag, [this, &shard]() { InitShard(shard); });
   return shard.transfer_caches[size_class];
 }

[[nodiscard]] int RemoveRange(int size_class, const absl::Span<void*> batch)
     ABSL_LOCKS_EXCLUDED(lock_) {
   TC_ASSERT(!batch.empty());
   TC_ASSERT_LE(batch.size(), kMaxObjectsToMove);
   auto info = slot_info_.load(std::memory_order_relaxed);
   if (info.used) {
     AllocationGuardSpinLockHolder h(lock_);
     // Refetch with the lock
     info = slot_info_.load(std::memory_order_relaxed);
     int got = std::min<int>(batch.size(), info.used);
     if (got) {
       info.used -= got;
       SetSlotInfo(info);
       void** entry = GetSlot(info.used);
       memcpy(batch.data(), entry, sizeof(void*) * got);
       remove_hits_.LossyAdd(1);
       remove_object_hits_.LossyAdd(got);
       low_water_mark_ = std::min(low_water_mark_, info.used);
       return got;
     }
   }

   remove_misses_.LossyAdd(1);
   remove_object_misses_.Inc(batch.size());
   return freelist().RemoveRange(batch);
 }
```

The RemoveRange function from before will end up calling the get_cache function. I think this is a pretty interesting function so lets look into it

Whats going on here?

- We get the CPU ID via rseq
- Map the cpu to L3 cache domain
  - `CpuShard(cpu_layout_->CurrentCpu())`
- Returns the TransferCache for that shard
- Each shard has its own array of TransferCaches
  - one per size class

The idea after that is we will call RemoveRange on the actual cpu shard and see if we can get memory out of it, we do this by checking to see if the cache has any data in it (if info ==0)

- When it does we will enter the thread safe section by locking
  - We look to prevent allocations and makes sure objects are removed safely
- Check again cuz another thread may have ended
- After that we get the number of objects we need to return
  - We then loop till we get the necessary number of objects and

Finally if there is/was nothing in the cpu l3 cache we just go to the global freelist ie our layer 3 !

Layer-3 CentralFreeList::RemoveRange

```
template <class Forwarder>
inline int CentralFreeList<Forwarder>::RemoveRange(absl::Span<void*> batch) {
  TC_ASSERT(!batch.empty());
  if (objects_per_span_ == 1) {
    Span* span = AllocateSpan();
    if (ABSL_PREDICT_FALSE(span == nullptr)) {
      return 0;
    }
    batch[0] = span->start_address();
    return 1;
  }

  size_t object_size = object_size_;
  int result = 0;
  size_t num_spans = 0;

  absl::base_internal::SpinLockHolder h(lock_);

  do {
    num_spans++;
    Span* span = FirstNonEmptySpan();
    if (ABSL_PREDICT_FALSE(!span)) {
      result += Populate(batch.subspan(result));
      break;
    }
    const uint16_t prev_allocated = span->Allocated();
    const uint8_t prev_bitwidth = absl::bit_width(prev_allocated);
    const uint8_t prev_index = span->nonempty_index();
    int here = span->FreelistPopBatch(batch.subspan(result), object_size);
    TC_CHECK_GT(here, 0, "Failed to make progress.  Freelist corrupted?");
    const uint16_t cur_allocated = prev_allocated + here;
    TC_ASSERT_EQ(cur_allocated, span->Allocated());
    const uint8_t cur_bitwidth = absl::bit_width(cur_allocated);
    if (cur_bitwidth != prev_bitwidth) {
      RecordSpanUtil(prev_bitwidth, /*increase=*/false);
      RecordSpanUtil(cur_bitwidth, /*increase=*/true);
    }
    if (span->FreelistEmpty(object_size)) {
      nonempty_.Remove(span, prev_index);
    } else {
      const uint8_t cur_index =
          IndexFor(span->is_long_lived_span(), cur_allocated, cur_bitwidth);
      if (cur_index != prev_index) {
```

Going through the execution flow for this
- We get a ptr to the new span that we allocation

We then check a value inside of the central free list which is
- Objects_per_span_ == 1
  - Which means that every span (span is a non-owning slice (pointer + length) describing a batch array you pass) has 1 object and we should just allocate data
- If there are no spans we will just call Populate()

If there a span with objects that fufill the allocation request we will use that from the free list and fix the mappings as needed.

Which in the code is done by
- Getting the previous state for the current span
  - Prev_allocated = # of obj currently used
  - Prev_index = which bucket/list the current span is in

We then finally attempt to pop by using `FreelistPopBatch` which
- here is the number of objects successfully popped from this span.
- It must be positive; otherwise something is wrong with the freelist.

Layer-3 CentralFreeList::RemoveRange

```
1. CpuCache::FetchFromBackingCache()
   └─ ShardedTransferCacheManager::RemoveRange()
     └─ get_cache(size_class) ← Selects shard based on CPU's L3 cache
       └─ Shard[L3_id].transfer_caches[size_class].RemoveRange()
         └─ TransferCache::RemoveRange() [Per-shard instance]
           ├─ If has objects: Return from shard's cache
           └─ If empty: CentralFreeList::RemoveRange()
             └─ CentralFreeList::Populate()
               └─ StaticForwarder::AllocateSpan()
                 └─ PageAllocator::New()
                   └─ HugePageAwareAllocator::New()
                     └─ SystemAllocator::Allocate()
                       └─ SystemAllocator::MmapAlignedLocked()
                         └─ mmap(hint, size, PROT_NONE, flags, -1, 0) ← OS
=============================================================================
1. CpuCache::FetchFromBackingCache()
   └─ TransferCacheManager::RemoveRange()
     └─ cache_[size_class].tc.RemoveRange() ← Direct access, no sharding
       └─ TransferCache::RemoveRange() [Single shared instance]
         ├─ If has objects: Return from shared cache
         └─ If empty: CentralFreeList::RemoveRange()
           └─ CentralFreeList::Populate()
             └─ StaticForwarder::AllocateSpan()
               └─ PageAllocator::New()
                 └─ HugePageAwareAllocator::New()
                   └─ SystemAllocator::Allocate()
                     └─ SystemAllocator::MmapAlignedLocked()
                       └─ mmap(hint, size, PROT_NONE, flags, -1, 0) ← OS
```

VRIG

Layer 3 - The Span AllocateSpan

Key files: central_freelist.cc

```
Span* StaticForwarder::AllocateSpan(int size_class, size_t
objects_per_span,
                                    Length pages_per_span) {
  const MemoryTag tag = MemoryTagFromSizeClass(size_class);
  const AccessDensityPrediction density =
AccessDensity(objects_per_span);

  SpanAllocInfo span_alloc_info = {.objects_per_span =
objects_per_span,
                                   .density = density};
  TC_ASSERT(density == AccessDensityPrediction::kSparse ||
           (density == AccessDensityPrediction::kDense &&
            pages_per_span == Length(1)));
  Span* span =
      tc_globals.page_allocator().New(pages_per_span,
span_alloc_info, tag);
  if (ABSL_PREDICT_FALSE(span == nullptr)) {
    return nullptr;
  }
  TC_ASSERT_EQ(tag, GetMemoryTag(span->start_address()));
  TC_ASSERT_EQ(span->num_pages(), pages_per_span);

  tc_globals.pagemap().RegisterSizeClass(span, size_class);
  return span;
}
```

We first figure out the memory tag

- kNormal, kNormalP0, kNormalP1, kCold, kSampled, kMetadata

We then try and figure out the density of the code base based on

- AccessDensityPrediction
  - This is used by the allocator to optimze span placement later on
  - We create the span_alloc_info object with all the info

Finally calling the function
`tc globals.page allocator().New(pages_per_span, span_alloc_info, tag)`

Which will actually allocate a page and is the start of our layer 4

Layer 4 - The Page Allocator

Key files: page_allocator.cc

```
PageAllocator::PageAllocator() {
  has_cold_impl_ = ColdFeatureActive();
  size_t part = 0;

  normal_impl_[0] = new (&choices_[part++].hpaa)
HugePageAwareAllocator(HugePageAwareAllocatorOptions{MemoryTag::kNormal});
  if (tc_globals.active_partitions() > 1) {
    normal_impl_[1] = new (&choices_[part++].hpaa) HugePageAwareAllocator(
        HugePageAwareAllocatorOptions{MemoryTag::kNormalP1});
  }
  sampled_impl_ = new (&choices_[part++].hpaa) HugePageAwareAllocator(
      HugePageAwareAllocatorOptions{MemoryTag::kSampled});
  if (has_cold_impl_) {
    cold_impl_ = new (&choices_[part++].hpaa)
HugePageAwareAllocator(HugePageAwareAllocatorOptions{MemoryTag::kCold});
  } else {
    cold_impl_ = normal_impl_[0];
  }
  alg_ = HPAA;
  TC_CHECK_LE(part, ABSL_ARRAYSIZE(choices_));
}
```

The general idea is we are going to stat by allocating page-level memory chunks vai the HugePageAwareAllocator

The memory tags from before come into picture here
If we have
- If we have the normal tag we create a normal implmetnation of the Huge Page

If we have a sampled partition we can create a swamped allocation (profilling ie we track of utilzaiton)

Finally if we have cold enabled we can create a expended allocator

VRIG

Layer 5 - The System Allocator

Key files: system_allocator

```
AddressRange StaticForwarder::AllocatePages(size_t bytes, size_t align,
                                            MemoryTag tag) {
  return tc_globals.system_allocator().Allocate(bytes, align, tag);
}

void StaticForwarder::Back(Range r) {
  tc_globals.system_allocator().Back(r.start_addr(), r.in_bytes());
}

MemoryModifyStatus StaticForwarder::ReleasePages(Range r) {
  return tc_globals.system_allocator().Release(r.start_addr(),
r.in_bytes());
}

void* MmapAlignedLocked(size_t size, size_t alignment, MemoryTag tag) {
  // ... setup code ...

  for (int i = 0; i < 1000; ++i) {
    hint = reinterpret_cast<void*>(next_addr);
    int flags = MAP_PRIVATE | MAP_ANONYMOUS | map_fixed_noreplace_flag;

    void* result = mmap(hint, size, PROT_NONE, flags, -1, 0);
    if (result == hint) {
      if (numa_partition.has_value()) {
        BindMemory(result, size, *numa_partition);  // NUMA binding
      }
      next_addr += size;  // Try to keep contiguous
      SetAnonVmaName(result, size, /*name=*/std::nullopt);
```