Mimalloc Paper sharing

2023.02

Mimalloc

01 Benchmarking Malloc with Doom 3

02 Mimalloc Introduction

03 Mimalloc Implementation

04 Mimalloc Benchmark

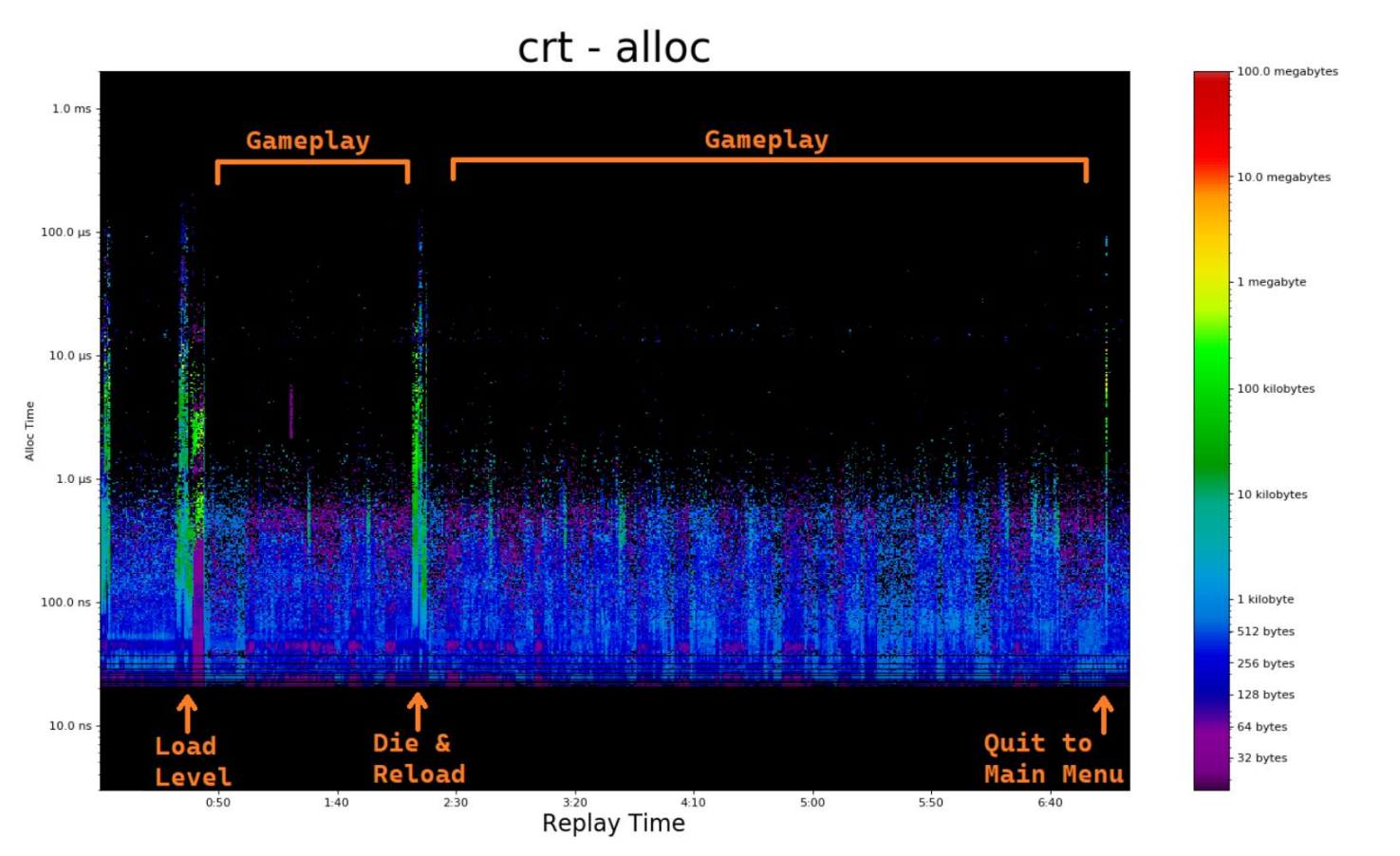
01

Benchmarking Malloc with Doom 3

一篇有趣的文章

Create the journal and visualize

Open Source Doom 3



https://github.com/RobertBeckebans/RBDOOM-3-BFG

Write every allocation to disk via std::fwrite

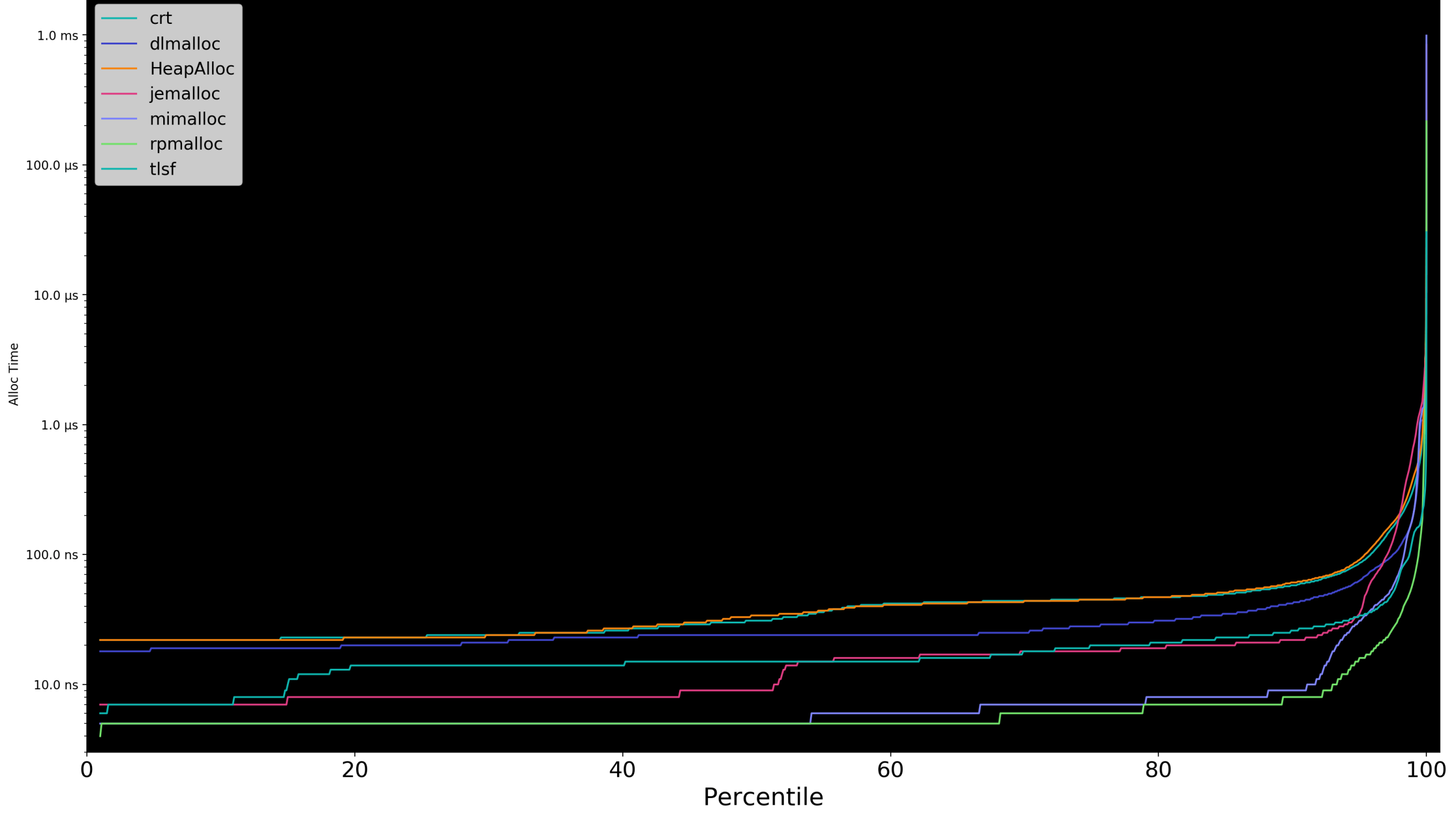
Visualize the Replay：

each pixel is one malloc or free

x-axis is replay time

y-axis is malloc/free time, in logarithmic scale

pixel color is alloc size


crt
dlmalloc
HeapAlloc
jemalloc
mimalloc
rpmalloc
tlsf
1.0 ms
100.0 µs
10.0 µs
1.0 µs
100.0 ns
10.0 ns
Alloc Time
0
20
40
60
80
100
Percentile

Observations

CRT is quite bad

jemalloc, mimalloc, and rpmalloc are "modern allocators"

- Are designed for multi-threading
- p50 malloc in under 10 nanoseconds
- p95 malloc in ~25 nanoseconds
- Worst 0.1%: 1 to 50 microseconds
- Absolute Worst: ~500 microseconds

free is comparable to malloc

Mimalloc

pronounced "me-malloc"

Mimalloc: Free List Sharding in Action

Mimalloc: Free List Sharding in Action

Microsoft Technical Report MSR-TR-2019-18, June 2019.

DAAN LEIJEN, Microsoft Research
BENJAMIN ZORN, Microsoft Research
LEONARDO DE MOURA, Microsoft Research

Modern memory allocators have to balance many simultaneous demands, including performance, security, the presence of concurrency, and application-specific demands depending on the context of their use. One increasing use-case for allocators is as back-end implementations of languages, such as Swift and Python, that use reference counting to automatically deallocate objects. We present mimalloc, a memory allocator that effectively balances these demands, shows significant performance advantages over existing allocators, and is tailored to support languages that rely on the memory allocator as a backend for reference counting. Mimalloc combines several innovations to achieve this result. First, it uses three page-local sharded free lists to increase locality, avoid contention, and support a highly-tuned allocate and free fast path. These free lists also support *temporal cadence*, which allows the allocator to predictably leave the fast path for regular maintenance tasks such as supporting deferred freeing, handling frees from non-local threads, etc. While influenced by the allocation workload of the reference-counted Lean and Koka programming language, we show that mimalloc has superior performance to modern commercial memory allocators, including tcmalloc and jemalloc, with speed improvements of 7% and 14%, respectively, on redis, and consistently out performs over a wide range of sequential and concurrent benchmarks. Allocators tailored to provide an efficient runtime for reference-counting languages reduce the implementation burden on developers and encourage the creation of innovative new language designs.

Additional Key Words and Phrases: Memory Allocation, Malloc, Free List Sharding

mimalloc (pronounced "me-malloc") is a general purpose allocator with excellent performance characteristics.

mimalloc is tailored to support languages that rely on the memory allocator as a backend for reference counting. (perform many small short-lived allocations, limit pauses when deallocating large data structures)

Mimalloc Usage

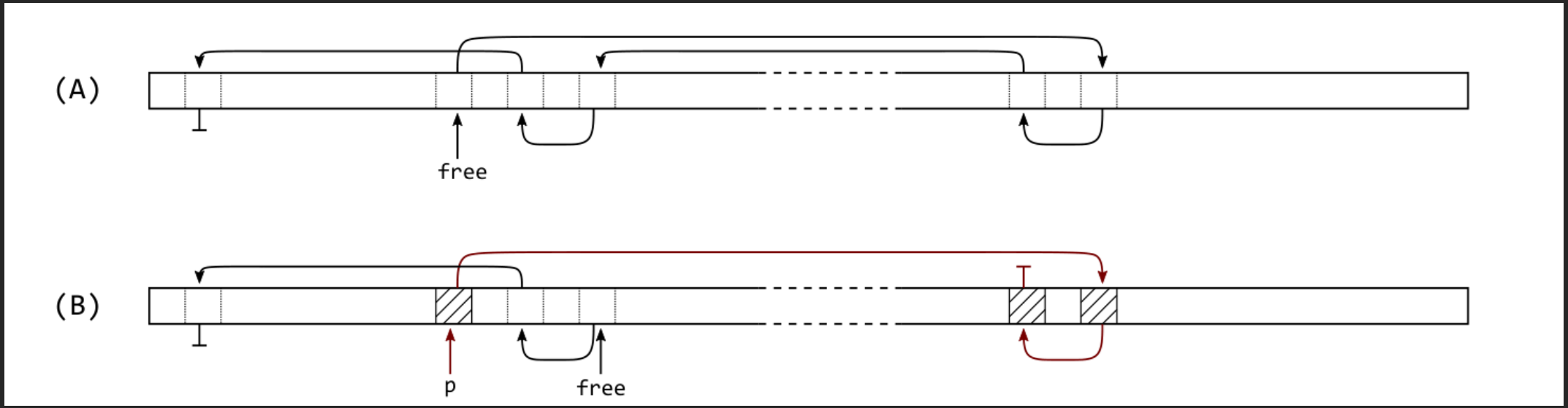
in game industry

UE 4.25

```
enum EMemoryAllocatorToUse
{
    Ansi,
    Stomp,
    TBB,
    Jemalloc,
    Binned,
    Binned2,
    Binned3,
    Platform,
    Mimalloc,
}
```

Free list sharding

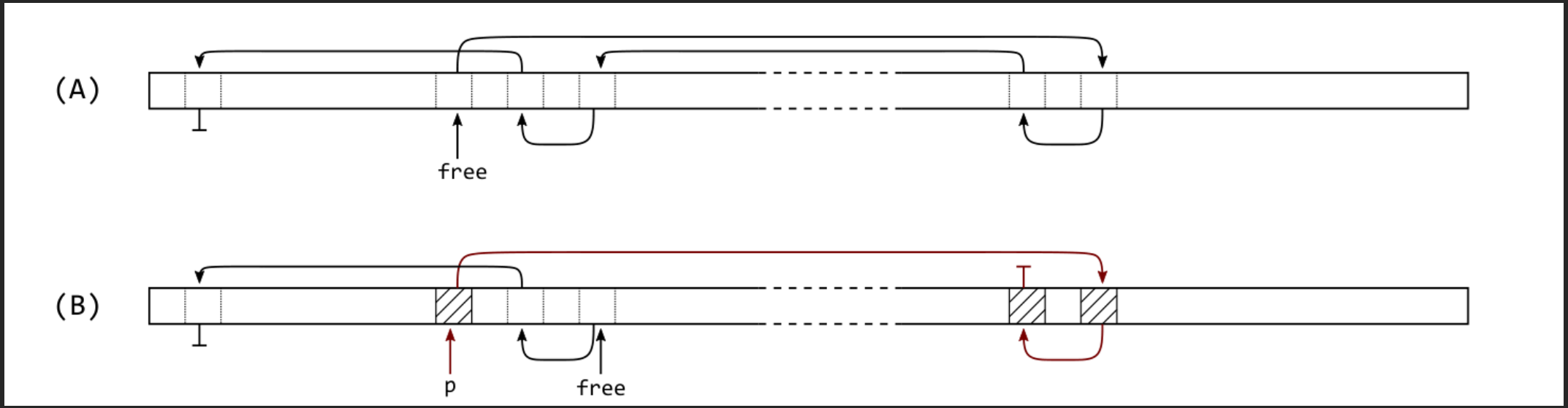
Many allocators use a single free list per size class which can lead to bad spatial locality.



Objects belonging to a single structure can be spread out over the entire heap.

Free list sharding

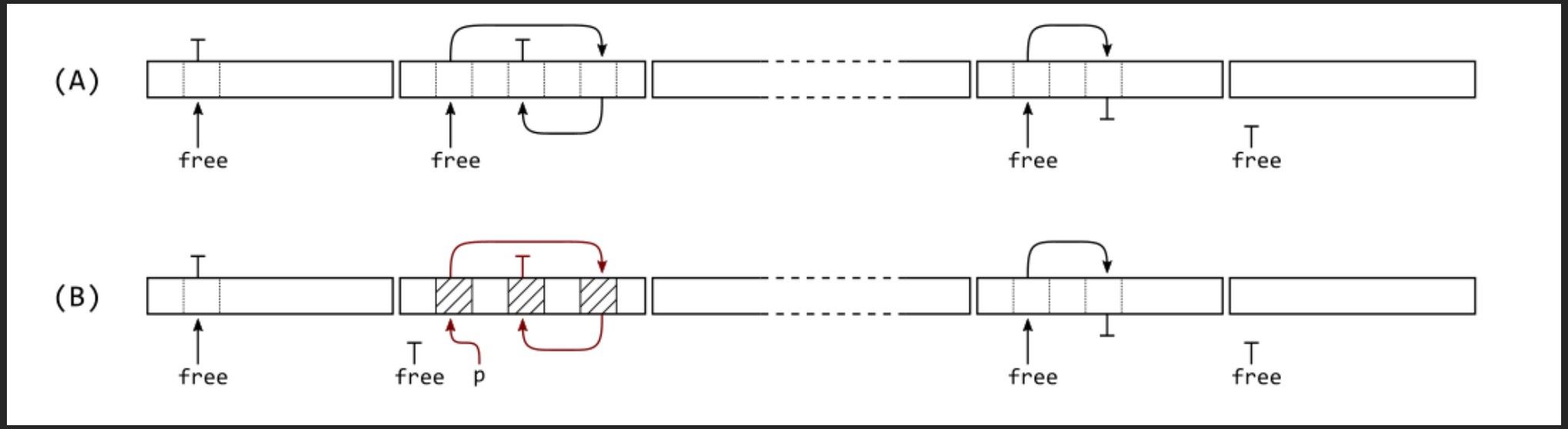
Many allocators use a single free list per size class which can lead to bad spatial locality.



When allocating a list p of three elements, we end up in state (B) where the newly allocated list is also spread out over a large part of the heap with bad spatial locality.

Free list sharding

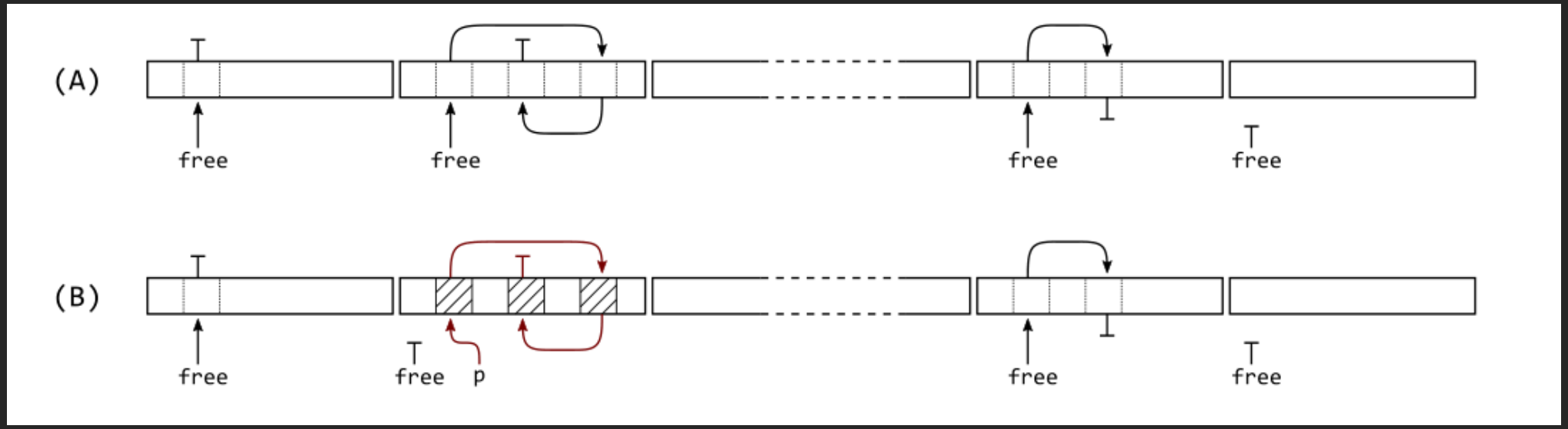
mimalloc use free list sharding



the heap is divided into pages (per size-class) with a free list per page (where pages are usually 64KiB for small objects).

Free list sharding

the good performance of mimalloc comes in a large part from the improved allocation locality.



This keeps locality of allocation as malloc allocates inside one page until that page is full, regardless of where other objects are freed in the heap.

The Local Free List

a local-free list per page for thread-local frees.

- add a sharded local free list to each page and while we allocate from the regular free list, we put any freed objects on the local free list instead. When the allocation free list becomes empty, the local-free list becomes the new free list.

```
page->free = page->local_free;  // move the list
page->local_free = NULL;        // and the local list is empty again
```

- This design ensures that the generic allocation path is always taken after a fixed number of allocations, establishing a temporal cadence. This routine can now be used to amortize more expensive operations:
  - do freeing for deferred reference count decrements
  - maintain a deterministic heartbeat
  - collect the concurrent thread-free lists
- Using the separate local-free list thus enables us to have a single check in the fast allocation path to handle all the above scenarios through the generic “collection” routine.

The Thread Free List

- **Separate thread-free lists for frees by other threads to avoid atomic operations in the fast path of malloc**
  - pages belong to a thread-local heap and allocation is always done in the local heap.
  - these thread-free lists are also sharded per page to minimize contention among them. Such list is moved to the local free list atomically every once in a while which effectively batches the remote frees

```
atomic_push(&page→thread_free, p);

void atomic_push(block_t **list, block_t *block)
{
    do
    {
        block→next = *list;
    } while (!atomic_compare_and_swap(list, block, block→next));
}
```

  - use the generic routine to collect the thread free list and add it to the free list

```
tfree = atomic_swap(&page→thread_free, NULL);
append(page→free, tfree);
```

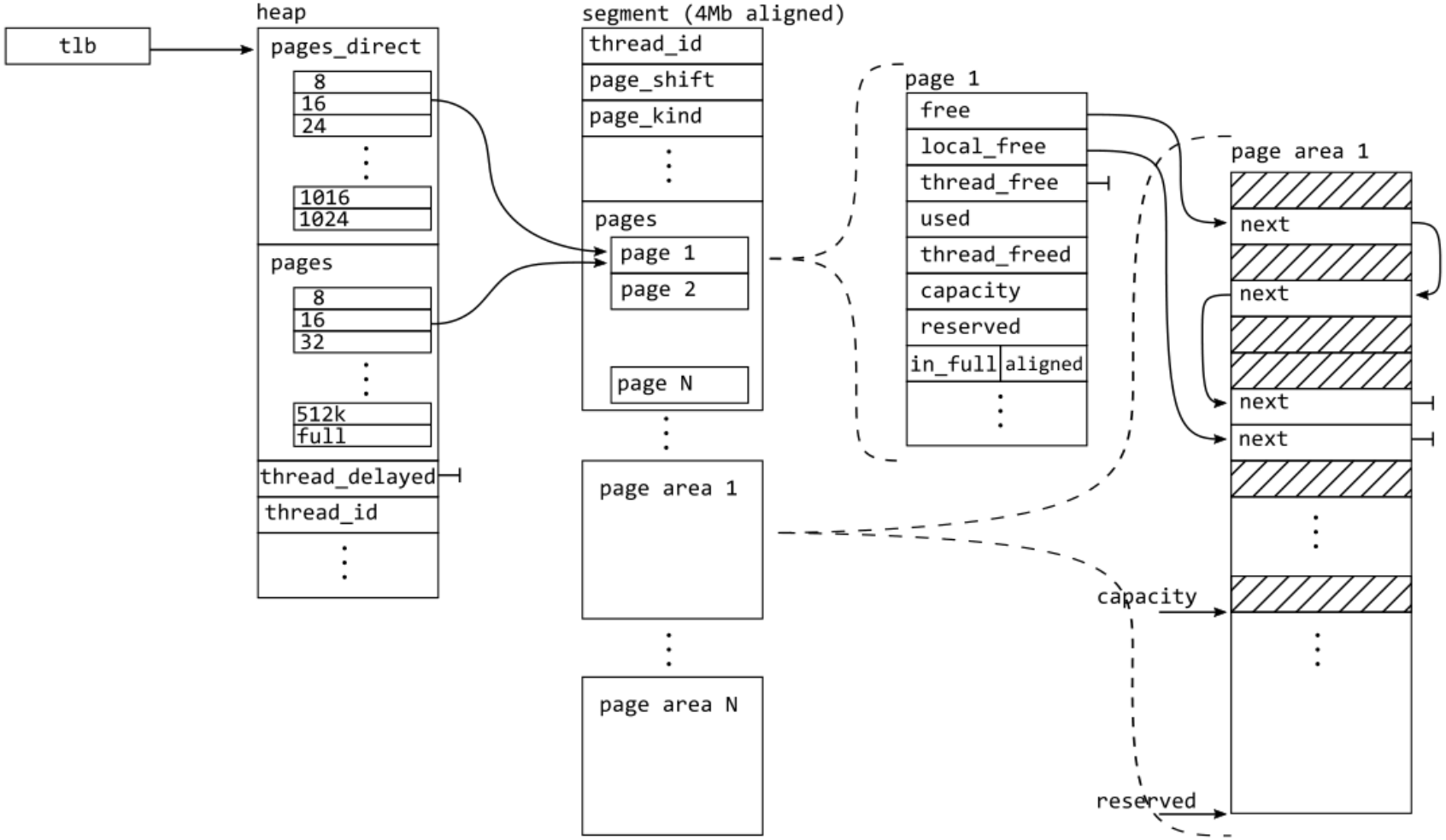
FREE FIRE

03

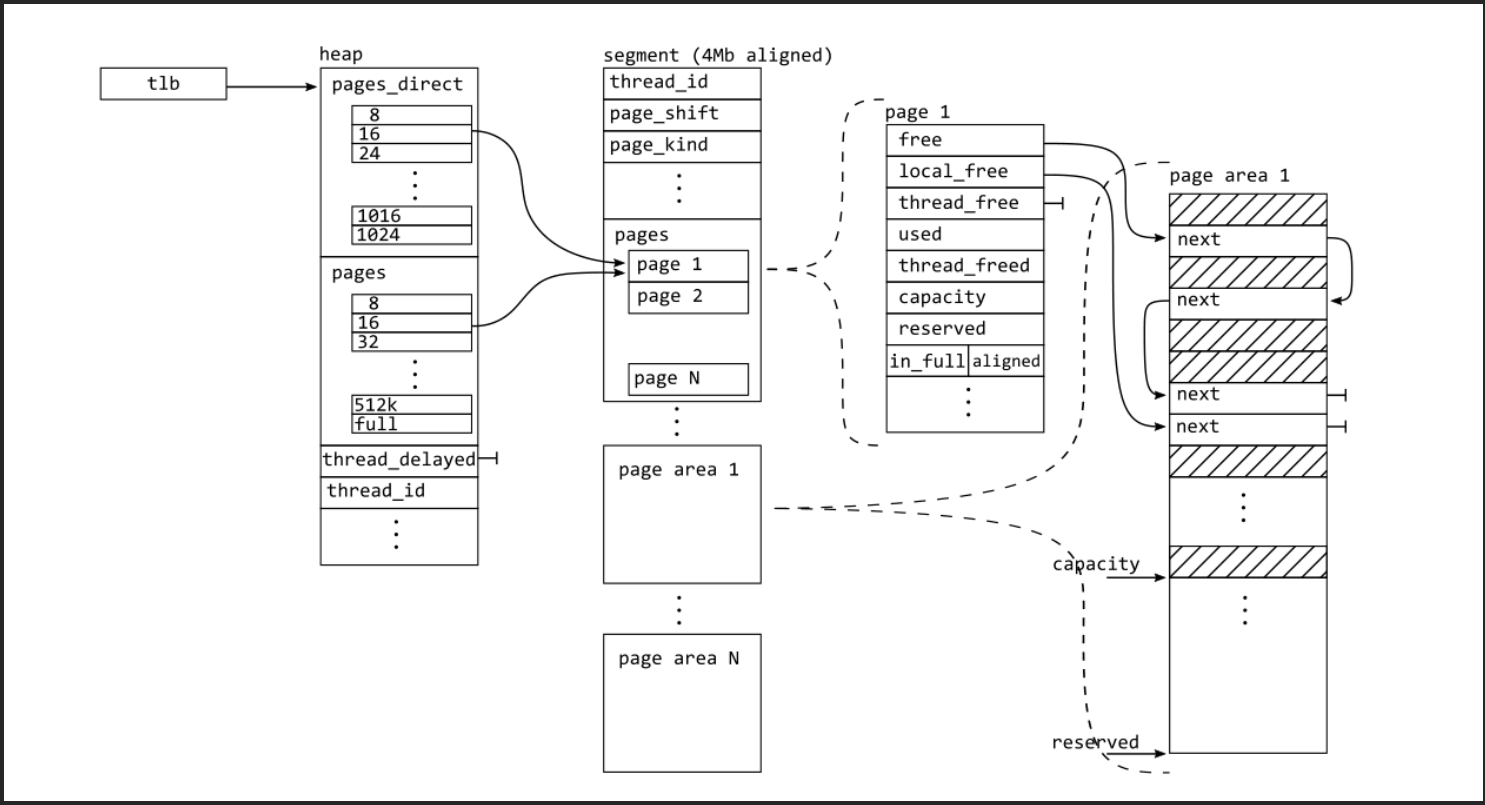
Mimalloc Implementation

understand the full design of the allocator

Heap layout


heap
tlb
pages_direct
8
16
24
1016
1024
pages
8
16
32
512k
full
thread_delayed
thread_id
segment (4Mb aligned)
thread_id
page_shift
page_kind
pages
page 1
page 2
page N
page area 1
page area N
page 1
free
local_free
thread_free
used
thread_freed
capacity
reserved
in_full
aligned
page area 1
next
next
next
next
capacity
reserved

Heap layout



The pages and the page meta-data all live in large segments, these segments are 4MiB (or larger for huge objects that are over 512KiB), and start with the segment and page meta data, followed by the actual pages where the first page is shortened by the size of the meta data plus a guard page.

For small objects under 8KiB the pages are 64KiB and there are 64 in a segment; for large objects under 512KiB there is one page that spans the whole segment, and finally huge objects over 512KiB have one page of the required size.

Malloc

- **Highly optimize the common allocation and free code paths and defer to the generic routine in other cases.**
  - To allocate an object, mimalloc first gets a pointer to the thread local heap (tlb). From there it needs to find a page of the right size class. For small objects under 1Kb the heap contains a direct array of pointers to the first available page in that size class.
  - This means that the data structures need to be very regular in order to minimize conditionals in the fast path. This consistent design also reduces special cases and increases code reuse – leading to more regular and simpler code. The core library is less than 3500 LOC(now 8k), much smaller than the core of other industrial strength allocators like tcmalloc (~20k LOC) and jemalloc (~25k LOC).
- **For small object allocation, the code becomes:**

```
// 0 < n ≤ 1024
void *malloc_small(size_t n)
{
    heap_t * heap = tlb;
    // divide up by 8
    page_t * page = heap→pages_direct[(n + 7) >> 3];
    block_t *block = page→free;
    if (block == NULL)
        return malloc_generic(heap, n); // slow path
    page→free = block→next;
    page→used++;
    return block;
}
```

Free

Pages and their meta data are allocated in a segment mostly to reduce expensive allocation calls to the underlying OS.

But there is another important reason: when freeing a pointer, we need to be able to find the page meta data belonging to that pointer.

The way this is done in mimalloc is to align the segments to a 4MiB boundary. Finding the segment holding a pointer p can then be done by masking the lower bits.

```
void free(void *p)
{
    segment_t *segment = (segment_t *)((uintptr_t)p & ~(4 * MB));
    if (segment == NULL)
        return;
    page_t *page =
         &segment→pages[(p - segment) >> segment→page_shift];
    block_t *block = (block_t *)p;
    if (thread_id() == segment→thread_id)
    {
        // local free
        block→next = page→local_free;
        page→local_free = block;
        page→used--;
        if (page→used - page→thread_freed == 0)
            page_free(page);
    }
    else
    {
        // non-local free
        atomic_push(&page→thread_free, block);
        atomic_incr(&page→thread_freed);
    }
}
```

Generic Allocation

- **malloc_generic, is our “slow path” which is guaranteed to be called every once in a while.**
  - This routine gives us the opportunity to do more expensive operations whose cost is amortized over many allocations, and can almost be seen as a form of garbage collector.

```
void *malloc_generic(heap_t *heap, size_t size)
{
    deferred_free();
    foreach (page in heap→pages[size_class(size)])
    {
        page_collect(page);
        if (page→used - page→thread_freed == 0)
        {
            page_free(page);
        }
        else if (page→free ≠ NULL)
        {
            return malloc(size);
        }
    }
    .... // allocate a fresh page and malloc from there
}
```

```
void page_collect(page)
{
    // move the local free list
    page→free = page→local_free;
    page→local_free = NULL;
     ...  // move the thread free list atomically
}
```

Other details

There are no locks, and all thread interaction is done using atomic operations.

- It has bounded worst-case allocation times, and meta-data overhead is about 0.2% with at most 16.7% (1/8th) waste in allocation size classes.

Security

- It puts OS guard pages in-between every mimalloc page such that heap overflow attacks are always limited to one mimalloc page and can never overflow into the heap meta data.
- The initial free list in a page is initialized randomly such that there is no predictable allocation pattern.
- To guard against heap block-overflow attacks that overwrite the free list, xor-encode the free list in each page.
- It efficiently supports multiple heaps. This can further increase security by allocating internal objects like virtual method tables etc. in a separate heap from other application allocated objects.

The Full List

- In the case of the gcc benchmark it happens to use its own custom allocators and allocate manylarge objects initially that than stay live for a long time. For mimalloc this leads to many (over 18000) full pages that are now traversed linearly every time in the generic allocation routine.
- have a separate full list that holds all the pages that are full, and move those back to the regular page lists when an object is freed in such page, unfortunately this seemingly small change introduces significant complexity for the multi-threaded case.

04

Mimalloc Benchmark

test mimalloc against many other top allocators over a wide range of benchmarks

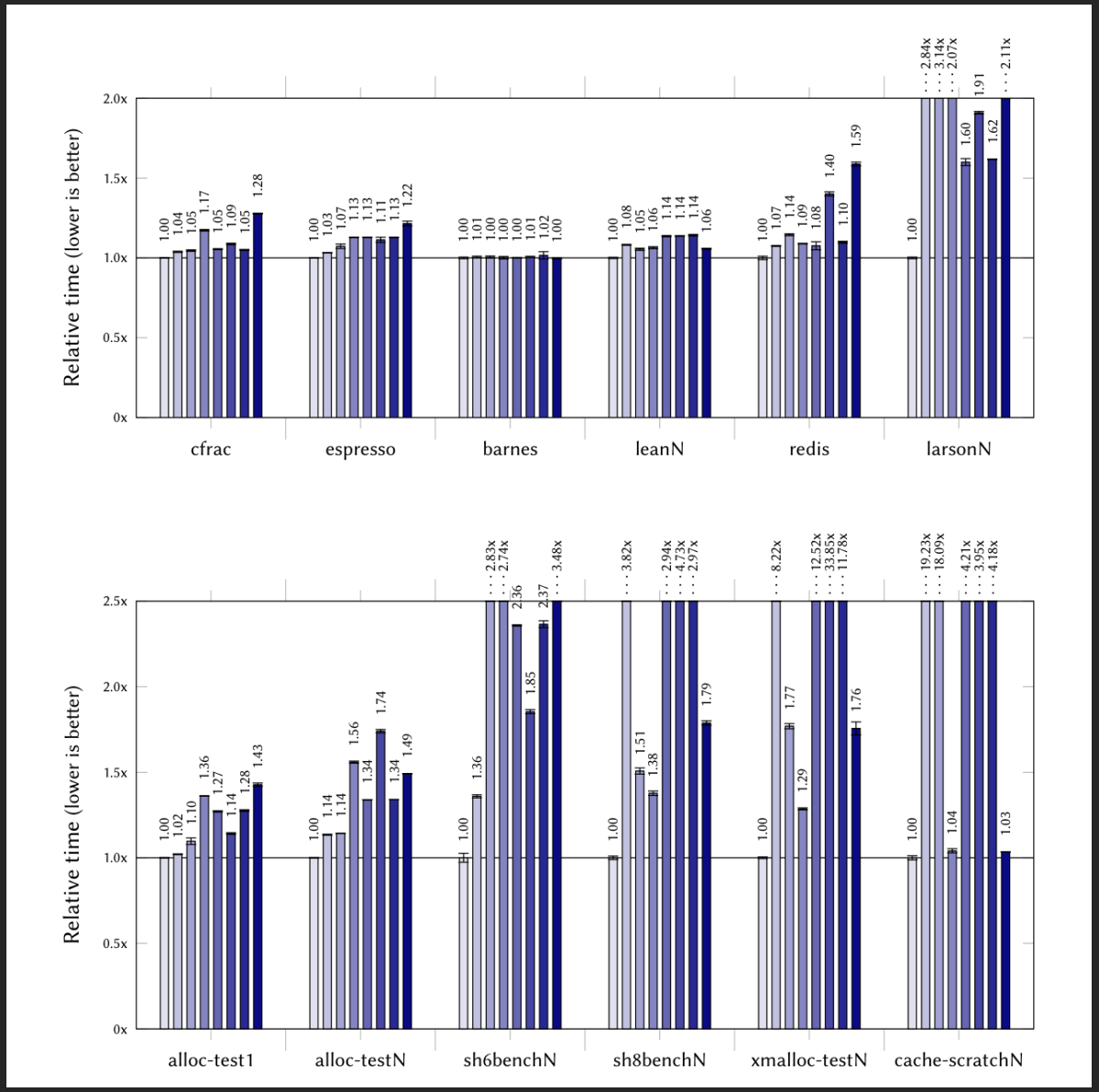
Evalution

- In our benchmarks, mimalloc always outperforms all other leading allocators (jemalloc, tcmalloc, Hoard, etc), and usually uses less memory (up to 25% more in the worst case).
- A nice property is that it does consistently well over the wide range of benchmarks: only snmalloc shares this property while all other allocators exhibit sudden (severe) underperformance in certain situations.

Evalution

On a 16-core AMD EPYC

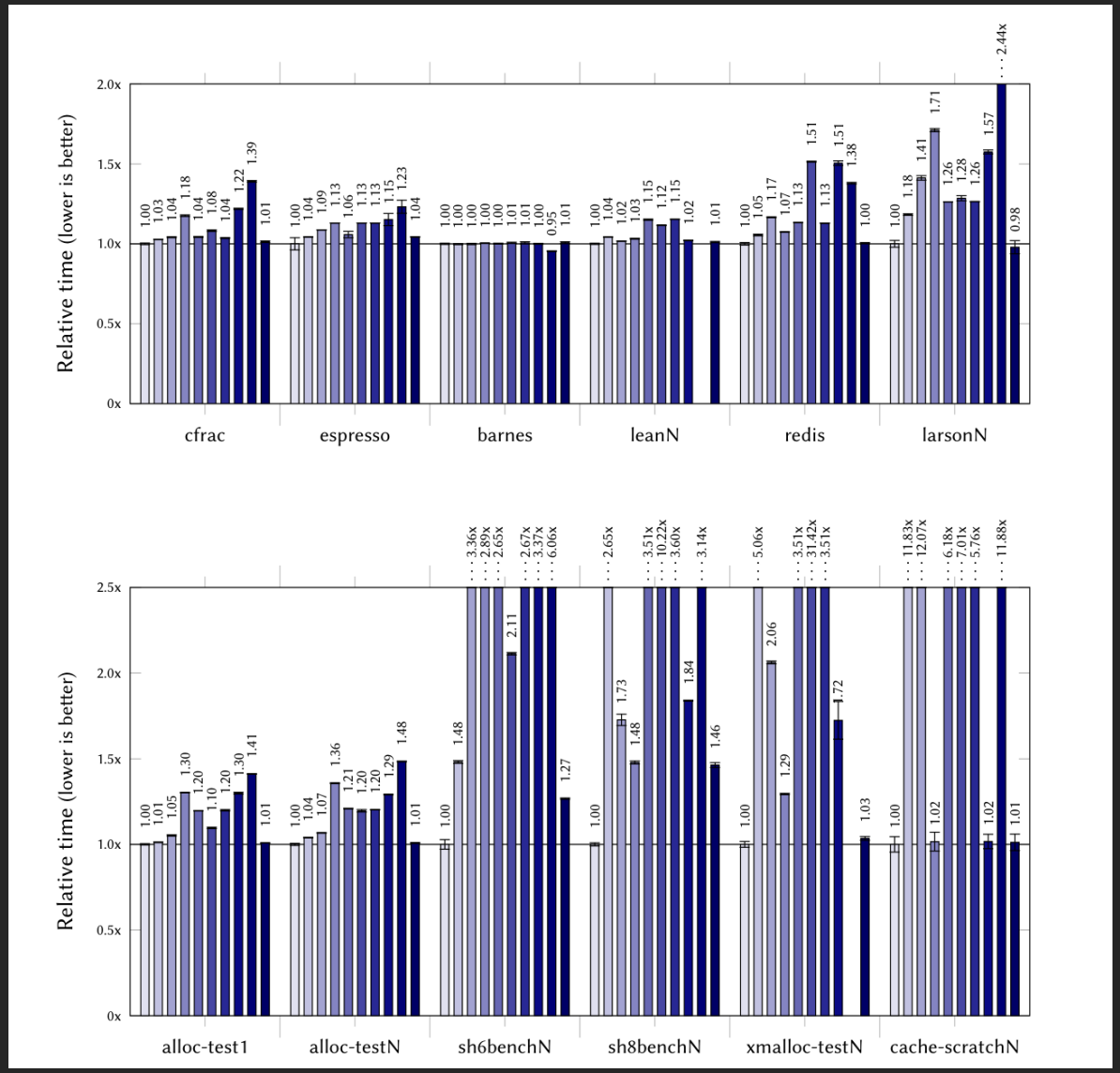
Time benchmark on a 16-core AMD Epyc r5a-4xlarge instance. Benchmarks ending with "N" run in parallel on all cores.

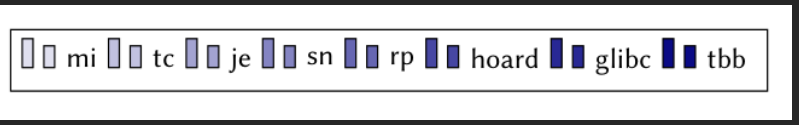


Evalution

On a 4-core Intel Xeon workstation

the relative results are quite similar as before. Most allocators fare better on the larsonN benchmark now – either due to architectural changes (AMD vs. Intel) or because there is just less concurrency.






FREE FIRE
THANK YOU