Mallacc: a malloc hardware accelerator

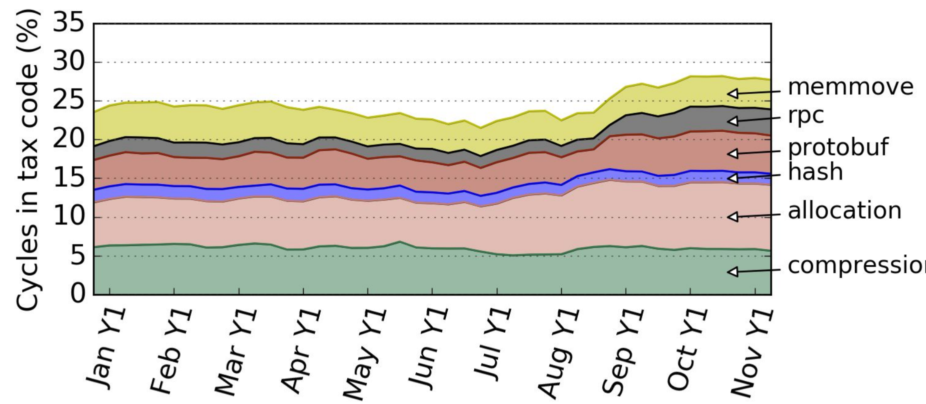
Save 30-50% malloc latency with this one little trick!*

**little = 1500μm^2, 0.006% of a Haswell core*

Svilen Kanev, Sam (Likun) Xi, Gu-Yeon Wei, David Brooks
Harvard University

Deep vs broad hardware acceleration

We are used to deep accelerators
motivated by 90/10 rules



[Kanev et. al, ISCA 15]

What if there is no 90% hotspot?
instead largest hotspots at 1-7%
datacenter tax: interspersed, fast, frequent

Different design targets for “broad” accelerators
limited gains → limited overheads
latency over throughput

What does an allocator do anyway?

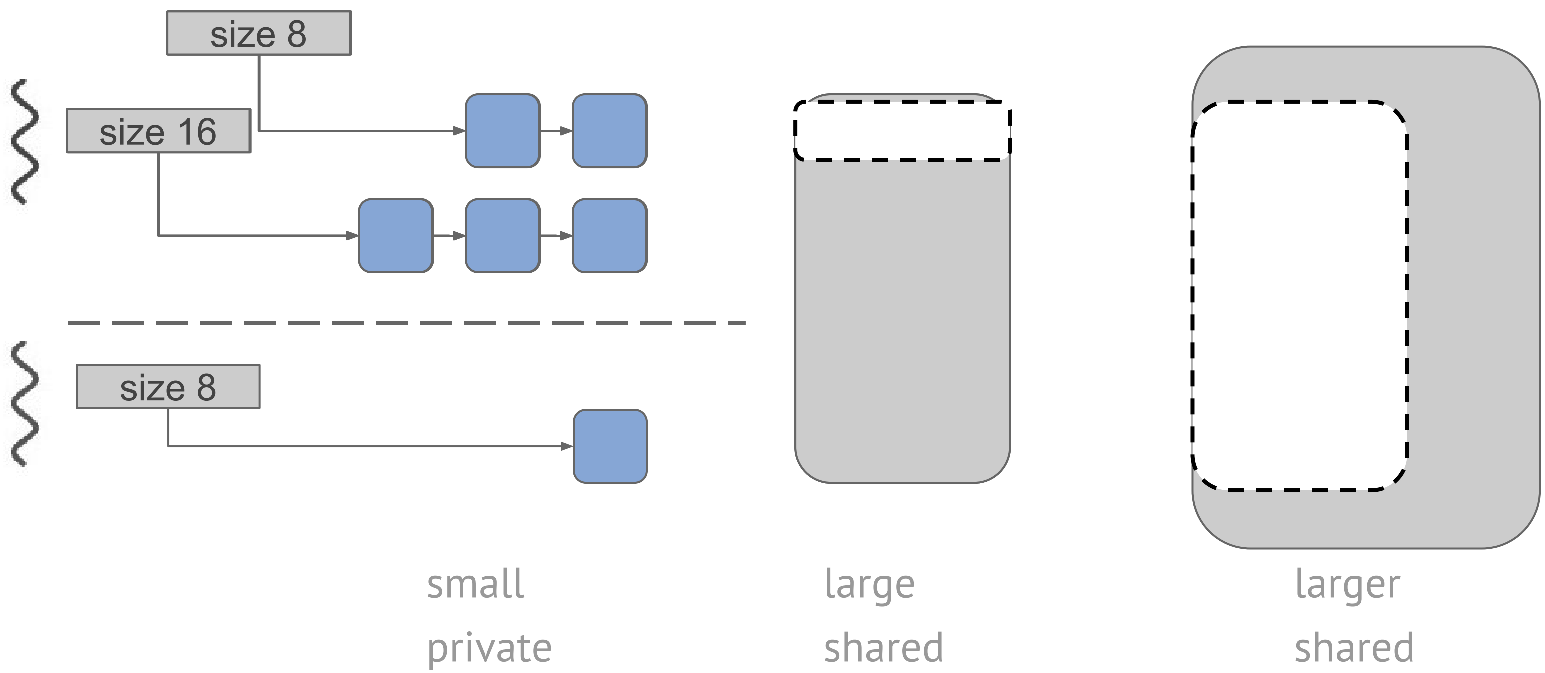
Bookkeep available memory

- get pages from the operating system

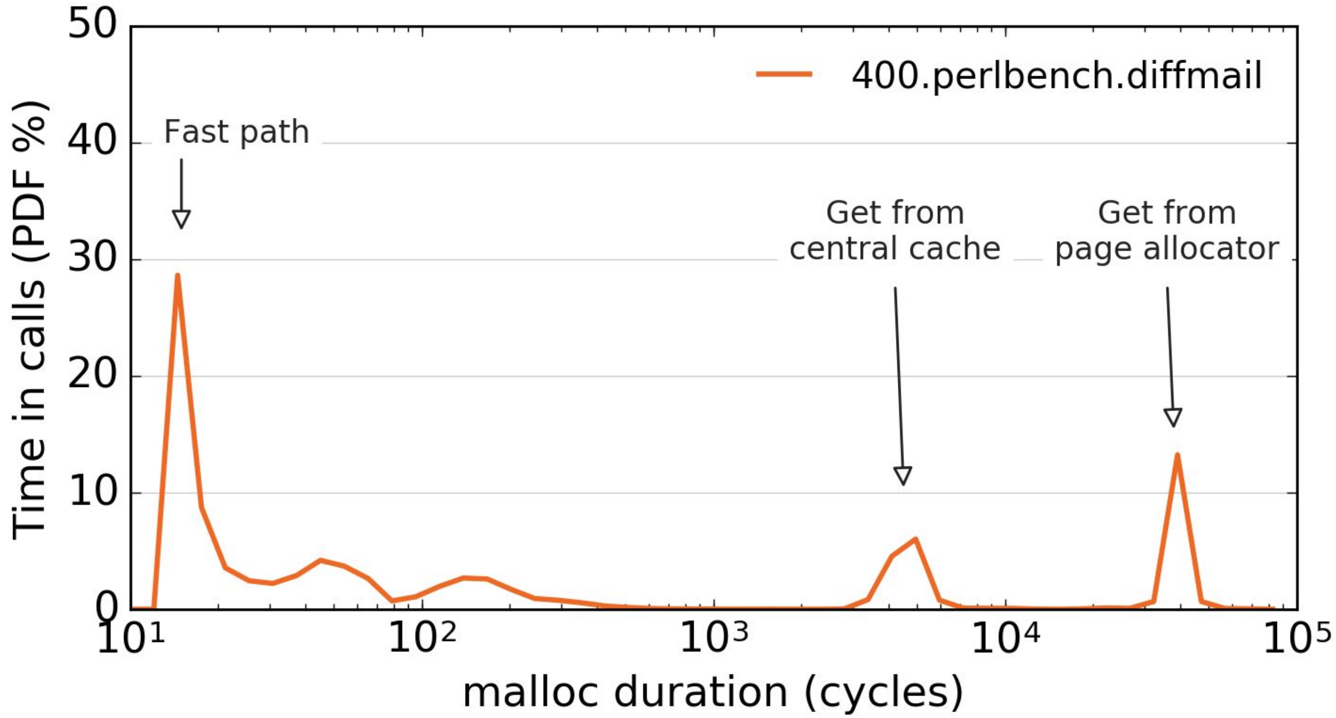
- give them out to application requests

What does an allocator do anyway?

Modern allocators (tcmalloc, jemalloc, Hoard, ...)
- round requests in size classes
- keep hierarchical pools of memory
- keep closest pools thread-private



Pool access costs vary by orders of magnitude

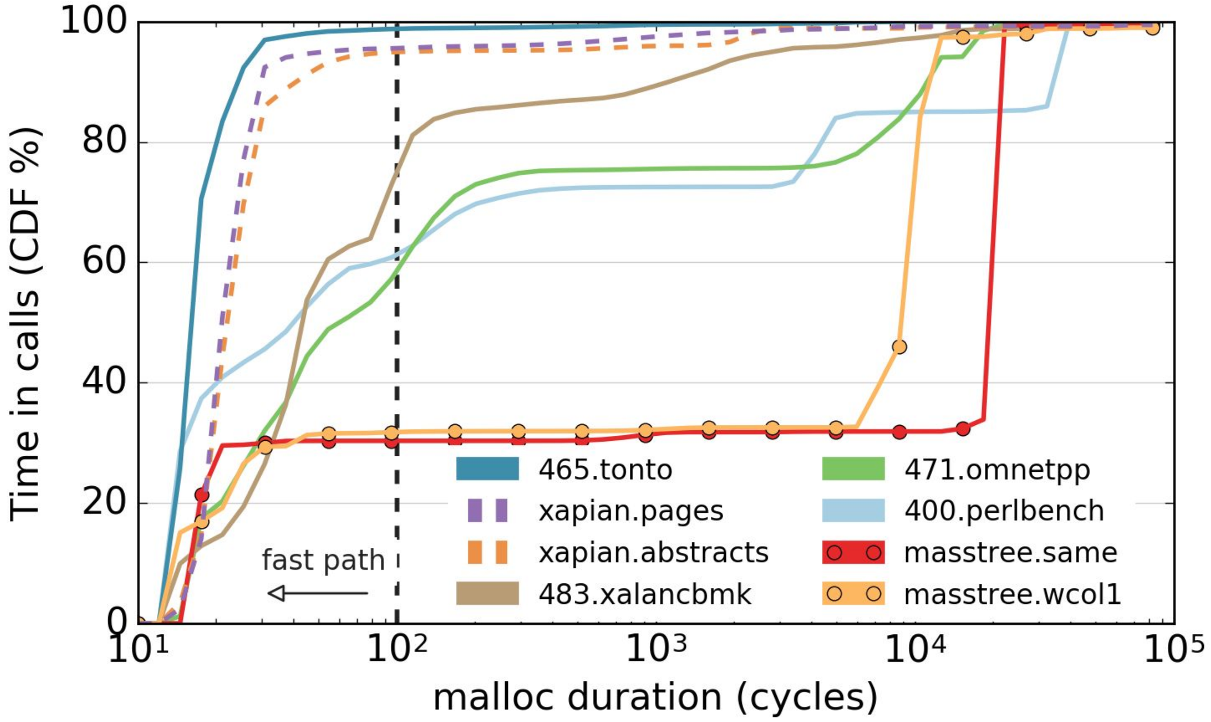


[XIOSim simulation; tcmalloc; Haswell CPU (<6% error)]

Most optimization effort spent at avoiding costly shared pools

fast paths are “fast enough”

“Death by a thousand cuts”



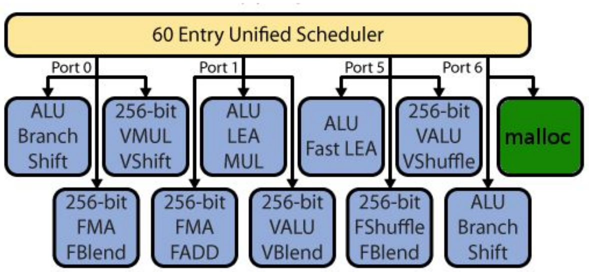
Fast paths can consume the bulk of allocation time (60+%)
underlooked optimization opportunity

Fast-path malloc() in hardware

Typical calls take ~6-7ns (20 cycles)

To do better in hardware

- integrate with the core
- dedicated functional unit + new instructions
- minimal overheads



Implementation details change frequently, don’t hardcode anything

Fast path deep dive

```
push    r13                          ; Register spilling.
push    r12
push    rbp
push    rbx
mov     rbp,rdi
sub     rsp,0x8                      ; Stack management.
mov     rax,0xffffffffffffffe8
cmp     rdi,QWORD PTR fs:[rax+0x8]   ; Is this a fast-path malloc?
jae     <__libc_malloc+0x118>        ; ... if so, continue.
mov     rbx,QWORD PTR fs:[rax]       ; Get TLS pointer.
lea     eax,[rbp+0x7]       ; Compute small size class index (<= 1KB).
shr     eax,0x3
mov     edx,eax
lea     eax,[rbp+0x3c7f]    ; Compute large size class index (> 1KB).
shr     eax,0x7
cmp     rbp,0x400           ; Determine which one to use.
cmovbe  rax,rdx
movzx   esi,BYTE PTR [rax+0x65f7a0]       ; Get size class
mov     r12, QWORD PTR [rsi*8+0x660020] ; Get rounded size
mov     rax,QWORD PTR [rbx+0x20]    ; Get current sampling threshold.
cmp     r12,rax                     ; If size < threshold ...
ja      426ee8                      ; ... Sample the allocation.
sub     rax,r12                     ; Else, update the threshold.
mov     QWORD PTR [rbx+0x20],rax    ; ... and store it.
lea     rax,[rsi+rsi*2]             ; Compute free list ptr.
shl     rax,0x3
lea     rdx,[rbx+rax*1+0x30]
cmp     QWORD PTR [rdx],0x0         ; Is free list null?
je      <__libc_malloc+0x230>
sub     QWORD PTR [rbx+0x10],r12    ; Decrement available memory ...
add     rbx,rax
mov     eax,DWORD PTR [rbx+0x38]
sub     eax,0x1                     ; Decrement free list length ...
cmp     eax,DWORD PTR [rbx+0x3c]
mov     DWORD PTR [rbx+0x38],eax    ; ... and store it.
jb      <__libc_malloc+0xb0>

mov     rbx,QWORD PTR [rdx]         ; result = *head.
mov     rax,QWORD PTR [rbx]         ; temp = head->next.
mov     QWORD PTR [rdx],rax         ; *head = temp.
mov     rax,QWORD PTR [rip+0x21df56]   ; Invoke new() hooks if present.
test    rax,rax
jne     <__libc_malloc+0xf8>
add     rsp,0x8                        ; Stack management.
mov     rax,rbx                        ; Returned ptr -> rax.
pop     rbx
pop     rbp
pop     r12
pop     r13
ret
```

requested size → size class
(cheap hash + lookups)

sampling
(for profiling)

update metadata
(still in software)

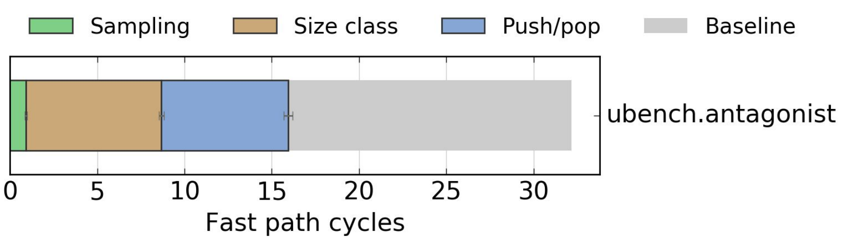
get free address for size class
(pop free list head)

Fast path deep dive

```
mov     rax,0xffffffffffffffe8
cmp     rdi,QWORD PTR fs:[rax+0x8]   ; Is this a fast-path malloc?
jae     <__libc_malloc+0x118>        ; ... if so, continue.
mov     rbx,QWORD PTR fs:[rax]       ; Get TLS pointer.
lea     eax,[rbp+0x7]        ; Compute small size class index (<= 1KB).
shr     eax,0x3
mov     edx,eax
lea     eax,[rbp+0x3c7f]     ; Compute large size class index (> 1KB).
shr     eax,0x7
cmp     rbp,0x400            ; Determine which one to use.
cmovbe  rax,rdx
movzx   esi,BYTE PTR [rax+0x65f7a0]      ; Get size class
mov     r12, QWORD PTR [rsi*8+0x660020] ; Get rounded size
mov     rax,QWORD PTR [rbx+0x20]     ; Get current sampling threshold.
cmp     r12,rax                      ; If size < threshold ...
ja      426ee8                       ; ... Sample the allocation.
sub     rax,r12                      ; Else, update the threshold.
mov     QWORD PTR [rbx+0x20],rax     ; ... and store it.
lea     rax,[rsi+rsi*2]              ; Compute free list ptr.
shl     rax,0x3
lea     rdx,[rbx+rax*1+0x30]
cmp     QWORD PTR [rdx],0x0          ; Is free list null?
je      <__libc_malloc+0x230>
sub     QWORD PTR [rbx+0x10],r12     ; Decrement available memory ...
add     rbx,rax
mov     eax,DWORD PTR [rbx+0x38]
sub     eax,0x1                      ; Decrement free list length ...
cmp     eax,DWORD PTR [rbx+0x3c]
mov     DWORD PTR [rbx+0x38],eax     ; ... and store it.
jb      <__libc_malloc+0xb0>

mov     rbx,QWORD PTR [rdx]          ; result = *head.
mov     rax,QWORD PTR [rbx]          ; temp = head->next.
mov     QWORD PTR [rdx],rax          ; *head = temp.
mov     rax,QWORD PTR [rip+0x21df56]   ; Invoke new() hooks if present.
test    rax,rax
jne     <__libc_malloc+0xf8>
```

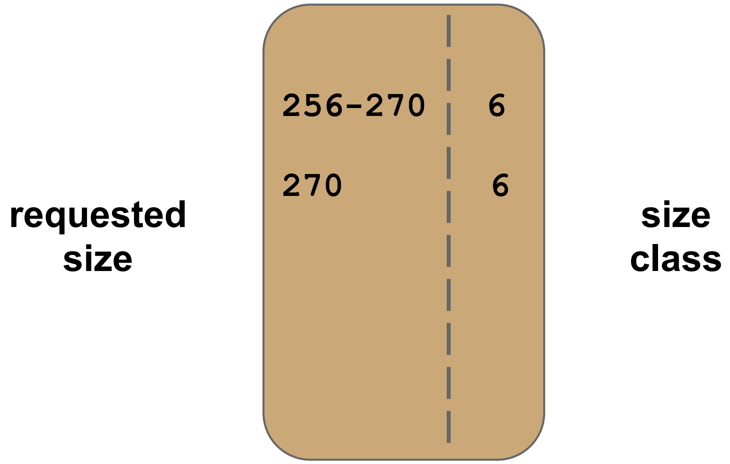
requested size → size class
(cheap hash + lookups)



get free address for size class
(pop free list head)

Mallacc: a 2-part, tiny, in-core, software-managed cache

Malloc cache: memorize hot size classes

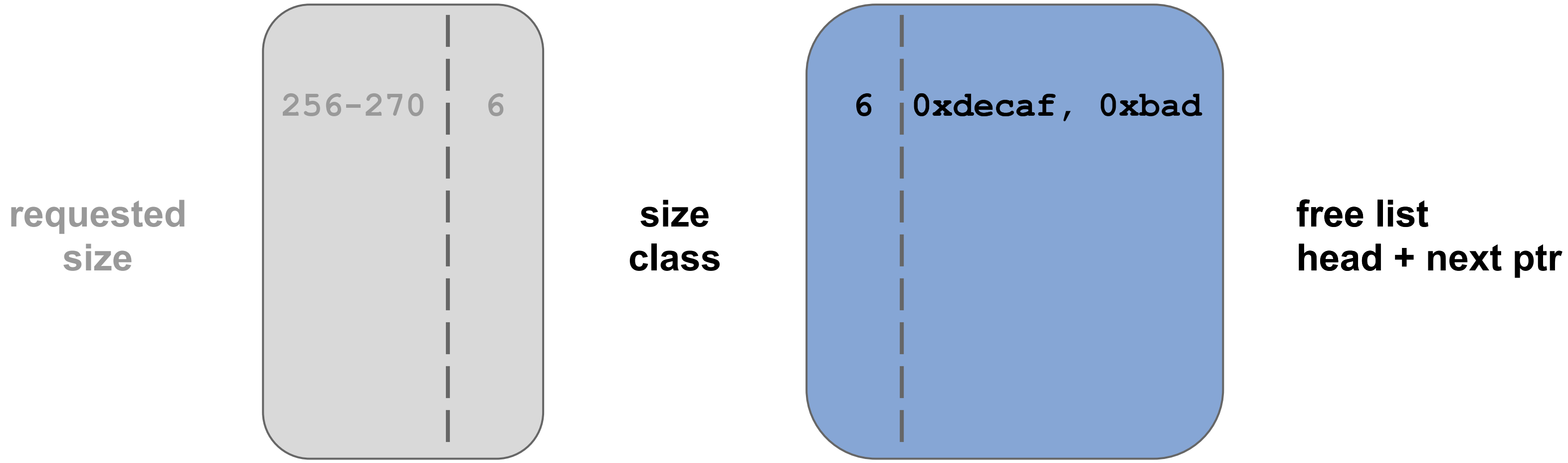


Lookup / replacement in software (2 new instructions)

no hardcoded allocator logic

Compares against size ranges

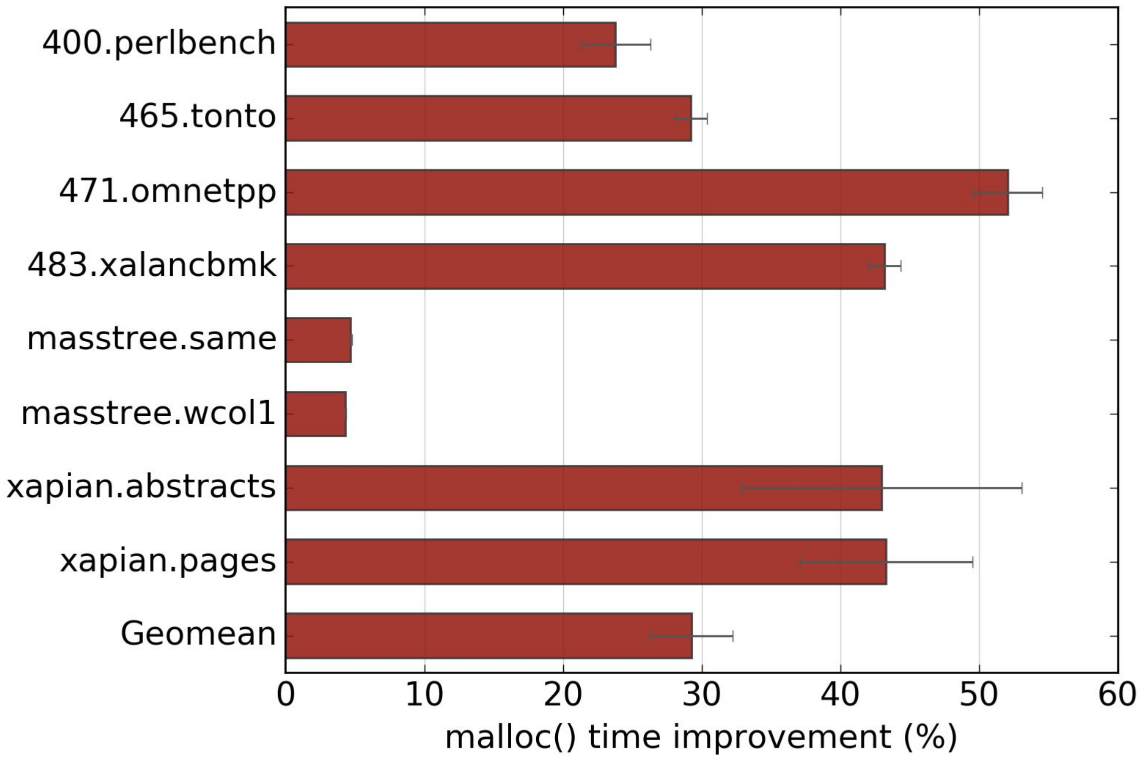
Malloc cache:
prefetch and store free list heads



A close copy of the current free list head node
- protected from cache antagonists between calls to malloc()
- immediate result, if a hit

Explicitly updated by software (pop/push on malloc/free)
- in parallel with the definitive copy in memory
- prefetch head→ next for sustainable high hit ratios

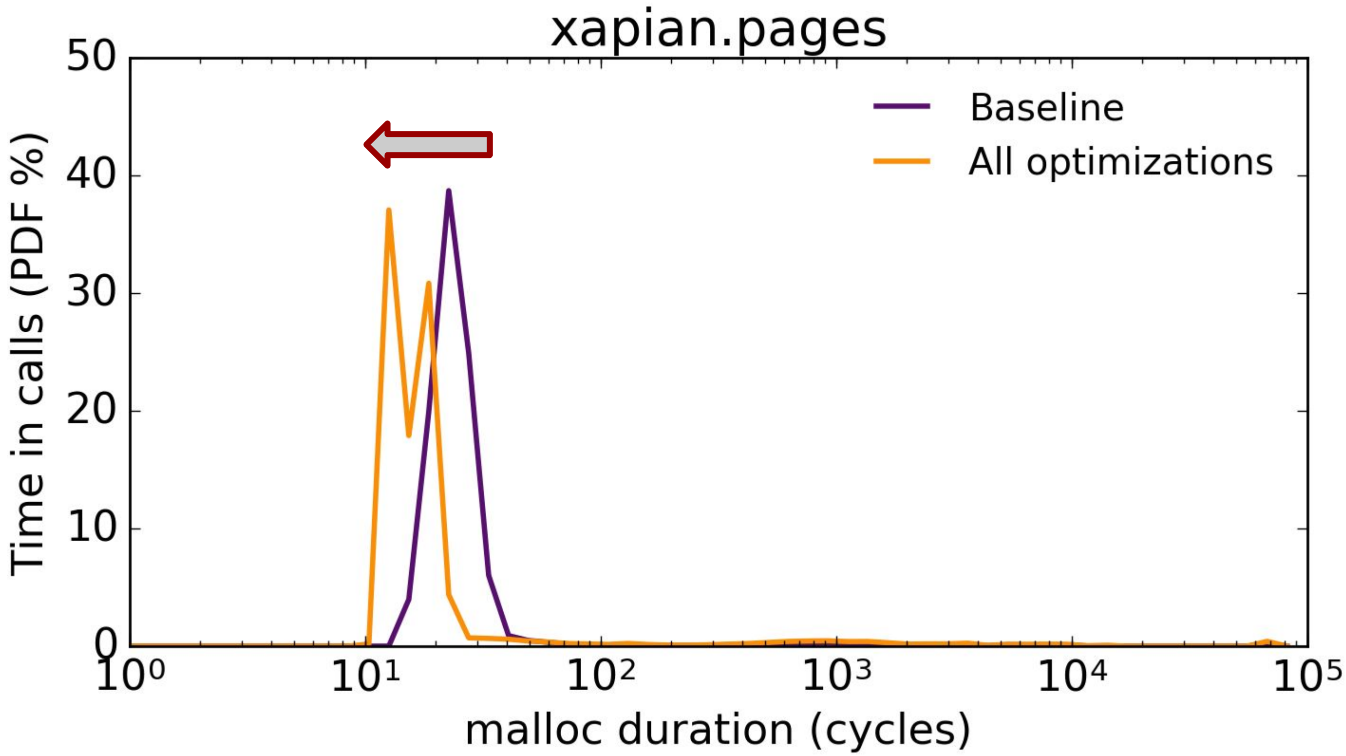
30-50% reduction in malloc time



Only 16 entries are sufficient (size class locality)

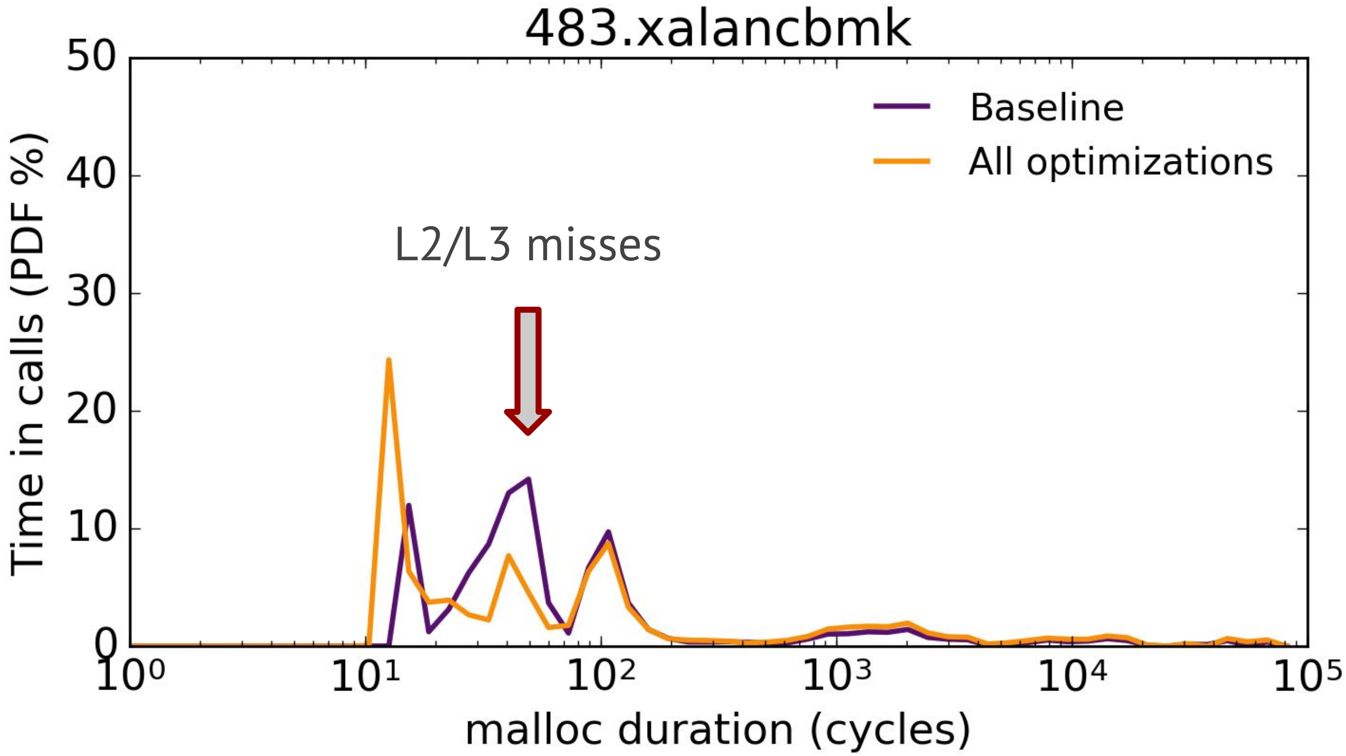
1500 μm^2 -> 0.006% of a Haswell core

Reasons for speedup: shorter critical paths



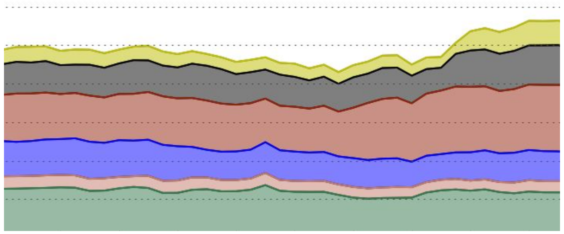
On a malloc cache hit, the call can produce a result in 5-6 instructions

Reasons for speedup:
protection from cache antagonists

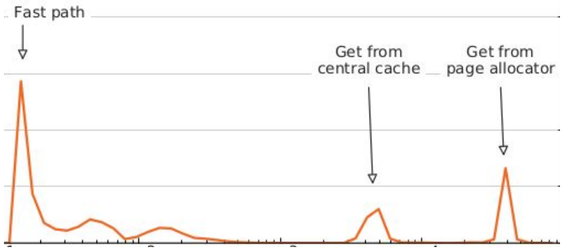


Allocator structures not evicted by the application

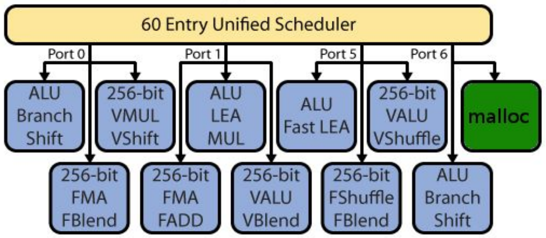
To sum up



Broad acceleration in the era of datacenter tax and dark silicon. Limited gains per accelerator, but also negligible overheads.



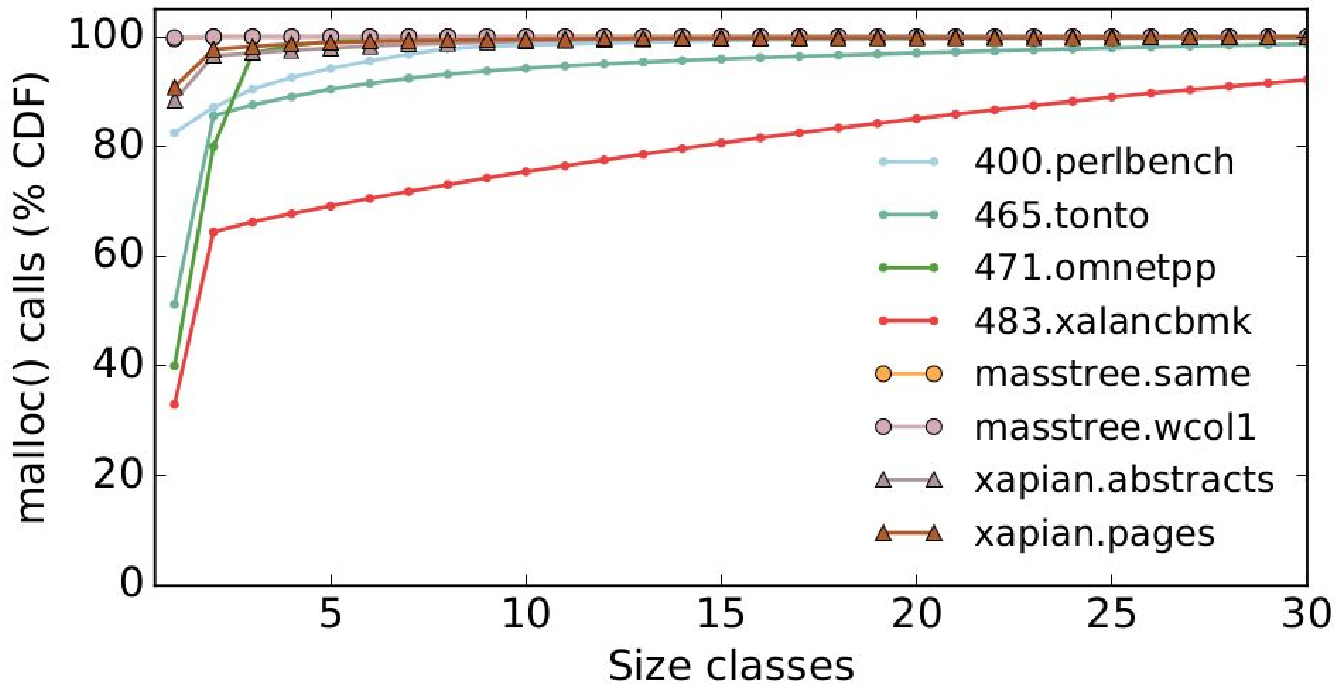
Memory allocation fast path is an overlooked opportunity for speedup. Many calls, each individually fast, aggregate to significant costs.



A small, dedicated, software-managed malloc cache can speed up allocation by 30%, with only 0.006% area overheads.

Backup

Some workloads have great size class locality



a 5-10-element cache should be enough

malloc cache example

```
malloc:
  ; rax = size class.
  ; rbx = location of the head of the free list.
  ; rcx = returned: head element.
  ; rdx = returned: next head element.
  ; rdi = temporary.
  mchdpop       rcx, rdx, rax           ; Search malloc cache.
  je cache_fallback                     ; If we missed, go to fallback.
  mov           QWORD PTR [rbx], rdx    ; Otherwise, update head.
  ; ...                                 ; ... and metadata.
  jmp malloc_ret
cache_fallback:
  ; Execute the original software.
  mov rcx, QWORD PTR [rbx]              ; head = *freelist->head.
  mov rdx, QWORD PTR [rcx]              ; next = *head.
  mov QWORD PTR [rbx], rdx              ; freelist->head = next.
malloc_ret:
  mcnxtprefetch  rax, QWORD PTR [rdx]   ; Prefetch the next head.
  ; Clean up stack and return value.

free:
  ; rax = freed pointer.
  ; rcx = size class.
  mchdpush  rcx, rax                    ; Update malloc cache head.
  ; The rest of free
```

Overall speedups

	Speedup	Stddev	p-value
400.perlbench	0.78%	0.05%	<0.001
465.tonto	0.35%	0.08%	0.025
483.xalancbmk	0.27%	0.06%	0.043
masstree.same	0.49%	0.05%	0.002
xapian.abstracts	0.55%	0.05%	0.002

Table 2: Full program speedup.