Mallacc: Accelerating Memory Allocation

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

Abstract

Recent work shows that dynamic memory allocation consumes nearly 7% of all cycles in Google datacenters. With the trend towards increased specialization of hardware, we propose Mallacc, an in-core hardware accelerator designed for broad use across a number of high-performance, modern memory allocators. The design of Mallacc is quite different from traditional throughput-oriented hardware accelerators. Because memory allocation requests tend to be very frequent, fast, and interspersed inside other application code, accelerators must be optimized for latency rather than throughput and area overheads must be kept to a bare minimum. Mallacc accelerates the three primary operations of a typical memory allocation request: size class computation, retrieval of a free memory block, and sampling of memory usage. Our results show that malloc latency can be reduced by up to 50% with a hardware cost of less than 1500 μm^2 of silicon area, less than 0.006% of a typical high-performance processor core.

1. Introduction

In the long term, the confluence of technology trends points steadily towards hardware specialization. Continued transistor density increases, coupled with the end of Dennard scaling, result in the inability to power a whole chip at maximum performance – the problem known as dark silicon. Hardware specialization has been widely adopted in processors to solve this problem.

Much existing effort in hardware specialization has focused on "deep" acceleration following the classic Amdahl's 90/10 rule. This involves identifying "killer applications" and optimizing their most costly kernels, be it ranking in websearch [19], convolutions in image processing [20], or matrix-vector products in neural network inference [21]. This strategy has seen especially great traction in mobile systems-on-chip, where the majority of silicon area in current designs is dedicated to specialized blocks [22]. However, the server chips powering cloud workloads remain predominantly general-purpose.

A major reason for this omission is that modern datacenter workloads are simply too diverse, without any opportunities for 90% optimization. Not only do they run thousands of different applications, but the individual workloads themselves have also been shown to not have significant hotspots that can be optimized with deep approaches [12]. This does not mean hardware acceleration in datacenters is infeasible. Characterization studies show that a large fraction of cycles is spent on the so-called "datacenter tax" – low-level routines like remote procedure calls, data serialization and memory allocation. While each individual component of this tax is a relatively mild hotspot (in the 2-8% range), together they can comprise up to 30% of all cycles in Google datacenters [12].

The ubiquity of the datacenter tax suggests an alternative "broad" approach to acceleration: speeding up multiple shared low-level routines that appear in many applications. This approach may not provide the 10× application speedups typically associated with hardware specialization. But accumulating several instances of such several-percent optimizations can save significant amounts of CPU cycles, especially when deployed broadly across the hundreds of thousands of servers that cloud providers operate. Borkar refers to this approach as "10×10 optimization" [2] and argues that it is a necessity for continued performance increases in the era of dark silicon.

Of the components that comprise the datacenter tax, perhaps the most familiar one is `malloc`: dynamic memory allocation. `malloc` is such a popular programming paradigm that many collective developer-years have been spent researching and optimizing allocation strategies and techniques. For example, a typical `malloc` call takes only 20 CPU cycles on a current-generation general-purpose processor, setting the bar high for potential hardware implementations. `malloc` exemplifies the unique set of challenges facing broad acceleration: because calls to these routines tend to be very frequent, fast, and interspersed inside other application code, accelerators must be optimized for latency rather

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037736

than throughput, and because each such accelerator brings a limited amount of overall application speedup, overheads must be kept to a bare minimum.

In this work, we present the design of *Mallacc*, a memory allocation accelerator that meets these constraints. Mallacc is a tiny in-core hardware block which accelerates the three primary operations of a typical memory allocation request: size class computation, retrieval of a free memory block, and sampling of memory usage. Mallacc is designed not for a specific allocator implementation, but for use by a number of high-performance memory allocators commonly found in datacenters today. Our goal is to make the already fast (20-30 cycle) `malloc` calls even faster, because they are so frequent, and Mallacc achieves that goal. It can reduce `malloc` latency by up to 50% while occupying less than 1500 μm^2 of silicon area. As we will show, Mallacc far exceeds the "1% speedup for 1% area" mantra that has informally guided processor development over the past decades.

2. Background

Dynamic memory allocation has been studied for decades. In this section, we place our work in the context of past literature. We discuss historical research on allocators, general techniques and structures that are still used in modern allocators, and factors that drove evolution of allocators over the decades.

At a very high level, a dynamic memory allocator sits between an application and the operating system (often as a part of the platform's standard library). It requests continuous blocks of memory from the OS and distributes chunks of them, with different sizes, to call sites in the application that explicitly request them. Allocators are judged on both the speed with which they satisfy a request and their memory fragmentation, which measures how much memory is requested from the OS vs. how much memory the application actually uses.

In the very early days, main memory was expensive and scarce, so allocator design focused on minimizing memory fragmentation and overhead. Starting from the 1960s, researchers studied data structures for storing free objects, notably linked lists [7] and trees [24]. Various strategies for searching through free lists of memory blocks to identify the right object to return were examined: such as returning the first block large enough ("first fit"), the exact size ("best fit"), and many more [6]. Techniques for efficiently splitting and coalescing free memory objects were also studied; one notable example is the buddy system, in which a free object can split into two "buddy" objects for small allocations, but can only be merged with that same "buddy" when a large allocation is needed [14]. The notion of *size classes* – allocating memory from a set of specific sizes – was also conceived decades ago [26].[1] Many of these techniques and data structures are still used in today's allocators.

Over time, two trends motivated significant changes in allocator design. First, main memory costs dropped and densities increased exponentially thanks to Moore's Law. However, unlike CPU speeds, main memory access latencies stagnated. The increasing gap between CPU and memory speeds shifted the focus from memory fragmentation to speed. Second, the rise of multi-core processors and multithreaded applications in the last decade motivated allocator designs that were fast and efficient in the face of problems like lock contention, false cache sharing, and memory blowup with large numbers of threads. Modern allocators like Google's `tcmalloc` [11], FreeBSD's `jemalloc` [8], Hoard [1], and others were all designed to support robust multithreaded performance.[2]

Modern multithreaded allocators like the ones listed above all share a common set of design principles. First, they logically organize available memory in a hierarchical fashion. The top level is a pool of memory that can only be accessed by a limited number of threads (often just one) to mitigate the cost of synchronization. These pools are highly optimized in software such that a hit in one is likely to be considered "fast enough". They are backed by lower-level pools, which are shared among threads. Memory is migrated back and forth as necessary. Second, they select a set of size classes and round requested sizes to the next nearest size class, which simplifies splitting and coalescing of larger memory blocks and reduces the amount of metadata needed. Third, they use different methods to allocate "small" and "large" chunks of memory (though they differ on the exact thresholds of considering a chunk small). Finally, they ensure that memory can migrate from thread to thread to avoid memory blowup in scenarios where one thread allocates memory and another thread frees memory.

Within this framework of common design principles, modern allocators can differ significantly in their implementations. For example, size classes are selected based on different upper bounds of memory fragmentation. Heuristics for determining when to migrate memory from lower to upper levels, as well as how many blocks to move, vary greatly too. Lower level pools tend to store larger blocks of memory that are then sliced into smaller chunks for top level pools, which is a time-consuming process that requires synchronization. Similarly, at some point additional memory must be requested from the operating system, which requires a costly system call. Developers must balance the frequency of these requests with the overall memory usage and consider various allocation patterns from different applications. Therefore, the parameters of these procedures tend

[1] Research in allocators has been especially prolific – for a significantly more complete survey of early approaches, refer to Wilson et al. [27].

[2] Similarly, Ferreira et al. [10] provide a succinct overview of the structure of modern allocators.

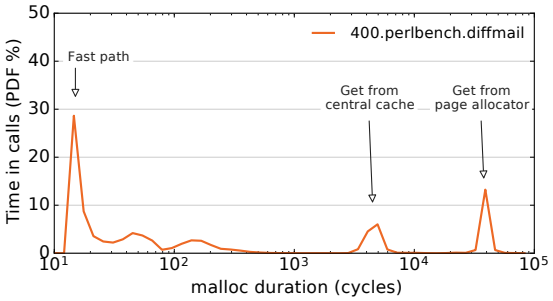

Figure 1: The costs of hits and misses in several allocation pools in TCMalloc vary by orders of magnitude.

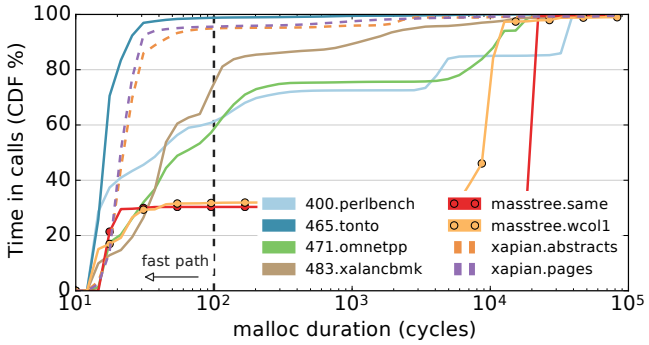

Figure 2: Majority of time in `malloc` in CPU2006 is spent on calls taking less than 100 clock cycles.

to change relatively frequently as developers seek out new optimizations and tradeoffs.

Compared to the effort spent on software optimization and tuninig, creating custom hardware for allocators has received next to no attention. We are only aware of one feasibility study [17] and several variations of the buddy technique [3, 4, 5, 16], which show that it easily maps to purely combinational logic. While buddy allocation has been available for decades, modern allocators have converged to simpler techniques in their highest-level pools (most frequently, first-fit free lists), most likely due to buddy systems' reported high degrees of fragmentation [27] and relative complexity.

This presents an opportunity for hardware designers looking to accelerate allocation. Rather than design a whole new algorithm from scratch to simplify hardware implementation, they can speed up the common elements of modern allocators – the "fast enough" top-level pools – and allow different allocator algorithms to tune the details on the lower levels in software for their own workload assumptions. In the rest of the paper, we demonstrate the feasibility of this approach by optimizing the top-level pools of TCMalloc [11]. While TCMalloc makes for a good anchor point to demonstrate gains – it is mature, robust and among the faster allocators [10] – the optimizations we propose can easily be used by other modern allocators.

3. Understanding TCMalloc

We start by describing how TCMalloc allocates and deallocates memory and compare and contrast it with other multi-threaded allocators. We profile the costs of several allocator code paths and find that the **fast path** is an overlooked area for potential optimization.

3.1 TCMalloc overview

Allocation pools Like many other allocators, TCMalloc allocates memory from a hierarchy of memory pools. At the top are *thread caches* assigned to each thread of a process, and meant to service small requests (< 256KB). Each cache contains many singly-linked *free lists* – lists with addresses to free chunks of memory of the same size. There is one free list per size class. TCMalloc currently has 88 size classes, a relatively large number picked to keep memory fragmentation low. When a free list is not empty, a small allocation can be satisfied by simply popping the head off the list. Since these caches are thread-local, no locks need to be acquired and a thread-cache hit is relatively fast. `jemalloc`'s thread caches were inspired by TCMalloc [9], and their size class organization is quite similar.

If a free list is empty, the allocator must first fetch blocks into a thread cache from a next-level pool. In TCMalloc, it either attempts to "steal" some memory from neighboring thread caches, or gets it from a *central free list*. Both approaches require locking, and are orders of magnitude slower than hitting in a thread cache. Should both of these sources be empty themselves, TCMalloc allocates a *span* (a contiguous run of pages) from a page allocator, breaks up the span into appropriately sized chunks, and places these chunks into the central free list and the thread-local cache. Large requests (> 256KB) go directly to spans and bypass the prior caches. Should the page allocator also be out of memory, TCMalloc then requests additional pages of memory from the operating system.

Figure 1 illustrates the cycle costs associated with hitting or missing in several of these pools for `400.perlbench` from SPEC CPU2006. It is a simulated distribution (details on our methodology follow in Section 5) of time spent in each `malloc()` call over the call's duration in cycles. The three major peaks correspond to hitting in a thread cache, missing in a thread cache and hitting in the central free list, and grabbing a span. Missing in a thread cache has a cost at least three orders of magnitude higher than that of a hit. Because of the high costs, too many misses in the highest-level pool can be detrimental to allocator (and application) performance. TCMalloc employs several heuristics to transfer chunks of memory between the various pools in an effort to maximize thread cache hit rates. These heuristics (and the particular implementation details of the lower-level pools) are what distinguishes different modern allocators from one another. Note, however, that despite their very low per-call cost, thread cache hits represent a significant chunk of allo-

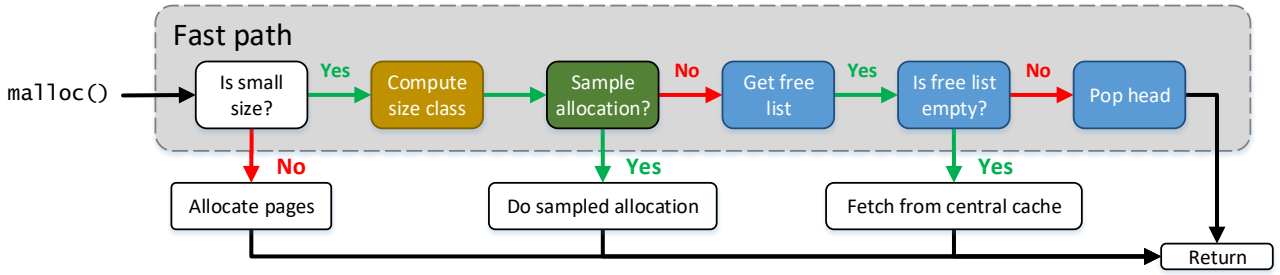

Figure 3: The steps that an allocation requests goes through. The colored boxes represent the major operations on the fast path, which we aim to optimize.

cator cycles overall for `400.perlbench`. We will come back to this observation in the following section.

Memory deallocation Deallocation follows a similar path. When memory is being freed, TCMalloc first determines the size class of the freed object. If that object is small, it gets pushed to the top of the appropriate thread cache free list, and if that free list now exceeds a certain size (2MB), TCMalloc returns unused objects back to the central free list. If the freed object is large, pages of memory get returned back to the page allocator.

3.2 Time spent in the allocator

As discussed in the prior sections, research in allocator design has focused on the lower-level memory pools because of their potentially catastrophic effects on performance. This is also partially because the *fast paths* – those that hit in thread caches – are already considered sufficiently optimized. Microbenchmark experiments often support such a hypothesis. For example, our `tp_small` microbenchmark (described later) achieves an average `malloc()` latency of only 18 cycles.

However, we find that for a range of applications, time spent on the fast path is not only a significant, but also a major fraction of time spent in the memory allocator. Figure 2 shows this property for the four SPEC CPU2006 benchmarks that actually call the system allocator. In the cumulative distribution of `malloc()` time, more than 60% of time is spent on calls that take less than 100 cycles. For `xapian`, an open source search engine, we see an even higher fraction. This need not be the case for all workloads: for example, the performance tests of `masstree`, a key-value store, never free any memory and end up continuously getting more from the page allocator (which eventually goes to the operating system). A real deployment of `masstree` does free memory and has much better thread-cache hit rates, but even such corner-case behavior spends more than 30% of allocator time on the fast path.

There are two main reasons for the high fraction of fast-path time that we observed. First, while individually cheap, fast-path calls can be very frequent – a classic "death by a thousand cuts" scenario. This is especially true for applications that allocate and deallocate at similar rates so that their requests almost never have to reach the other memory pool levels. Second, thread caches are very cheap *in microbenchmarks*, but can get significantly more expensive when the requesting application itself is cache-heavy. In that case, the application's memory accesses evict the allocator's data structures from the CPU's caches, and a cheap 18-cycle fast-path call can turn into a hefty 100-cycle stall on main memory.

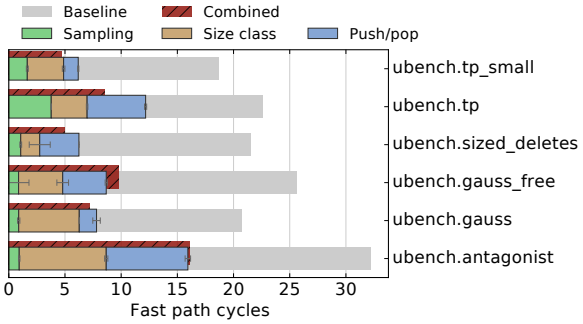

Figure 4: Time spent in the three main components of the fast path accounts for ≈ 50% of cycles.

Thus, we believe the fast path of memory allocators presents an overlooked opportunity for optimization and focus the rest of the paper on speeding it up with specialized hardware. For that, we need a detailed understanding of the work done during fast path calls, and the costs associated with it.

3.3 Analysis of the fast path

By definition, the fast path is a memory request satisfied by a thread cache free list. And by design, an access on the fast path has little work to do. For TCMalloc compiled with GCC 6.1, the fast path is only ≈ 40 static x86 instructions, and can take 18-20 cycles, assuming cache hits. It contains a few conditional branches that are easy to predict and no loops. Microbenchmarks with back-to-back allocations and deallocations can achieve an IPC of 3.0 on a 4-wide Intel Haswell core. In other words, it has been heavily optimized. Thus, speeding it up further is an exercise in performance microscopy and in reducing the latencies of the different

```
size_t SizeClass(size_t size) {
  size_t class_index;
  if (size <= 1024)
      class_index = (size + 7) >> 3;
  else
      class_index = (size + 15487) >> 7;
  return size_class_table[class_index];
}

size_t class = SizeClass(requested size);
size_t alloc_size = size_table[class];
```

Figure 5: Size class lookup function.

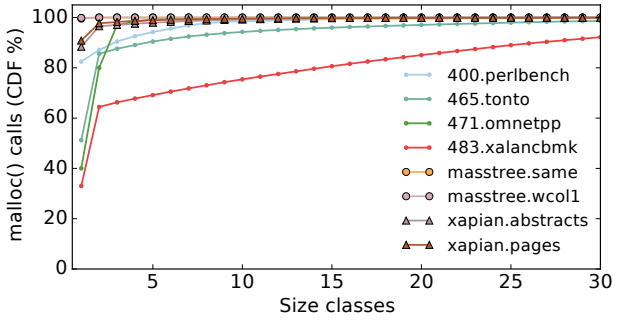

Figure 6: Many benchmarks use a very small number of size classes.

steps of a fast allocation. Figure 3 illustrates these steps in the context of an incoming allocation request: 1) finding the appropriate size class for the requested size, 2) potentially sampling the request, and 3) satisfying it by popping the head of the corresponding free list. In the rest of the section, we go into more detail about the computation in each step and its cycle costs.

Note that we can (and do) estimate these costs, even if they are caused by only several instructions, because we rely on simulation. This is how we construct Figure 4, which contains cycle costs for several microbenchmarks designed to stress different fast path aspects. With simulation, we can simply remove these instructions from simulated execution and subtract the resulting cycle count from a baseline. These are estimates, and not strictly additive, since out-of-order cores explicitly overlap work. When all removed together (the `Combined` bars in Figure 4), they make up for half the cycles of the fast path.

Size class calculation The first part of an allocation request rounds the requested memory size to the nearest size class supported by the allocator. The number and spacing of size classes is carefully tuned to balance fragmentation and allocator latency, so typically the mapping from size to size class does not have an easy closed form. For example, because small size classes are more commonly observed, the spacing between small size classes is closer, and it grows the larger the size class is. In TCMalloc, this mapping is implemented by first computing a *size class index* from the requested size and then indexing into two arrays which are precomputed at initialization time for the size class and rounded size that it represents (Figure 5). The class index only requires an add and a shift, but the two array lookups can be comparatively costly, even if they hit in the L1 cache because they are on the critical path of execution. The number of class indices (the size of the first array) is set by the threshold for a small allocation and by memory alignment requirements. This number was fixed at slightly above 2100 in 2007 when TCMalloc was open-sourced and has not changed. The second array is much smaller, currently at 88 (the number of size classes), and has seen two small increases since 2007.

```
pop:
  load temp, MEM[head]         ; Get the head.
  load next_head, MEM[temp]    ; Get head->next.
  store MEM[head], next_head   ; head = head->next.
  return temp

push:
  load temp, MEM[head]         ; Get the head.
  store MEM[head], new_head    ; Set new_head as head.
  store MEM[new_head], temp    ; new_head->next = temp.
```

Figure 7: Critical memory accesses on a free list push/pop.

Despite having 88 size classes available, we find that applications often use a relatively small subset. Figure 6 shows that, for the benchmarks we surveyed, all but one use less than 5 size classes on 90% of `malloc` calls. In fact, `masstree` almost exclusively uses a single size class.[3] `xalancbmk` has a much broader distribution, but even so, it uses two size classes over half of the time. This observation motivates techniques to memorize the most common size classes.

While usually `free` is perfectly complementary to `malloc` and we rarely mention it, there is a marked difference in size class computation. `free` does not take a size parameter, only the pointer to be deallocated, so it must perform extra work to determine the size class to return it to. In TCMalloc, this is implemented by a hash lookup from the address being freed to the size class. This hash tends to cache poorly, especially in the TLB, leading to expensive losses. C++11 ameliorates this problem because the compiler can choose to call `operator delete()` with an extra parameter equal to the size of the object being freed, as long as the object's size can be determined at compile time. With `-fsized-deallocation`, the compiler prefers calling that variant when it can. In our results, we assume sized delete when applicable.

Push/pop a free list head Once a size-class is identified, all that is left is to pop (or push) the head of its free list. Pushing to or popping from a free list generates a dependent chain of three memory accesses, as shown in Figure 7. In

[3] For allocations below 256KB only, which are handled by the fast path.

Valid	Size range (index range)	Size class	Size	Head	Next
1	0 - 1	1	8	0x8080	0x8088
1	63 - 64	25	512	0x9090	0x9290
1	5 - 6	4	48	0x0	0x0
0	-	-	-	-	-

Figure 8: A malloc cache with example values. The cache is searched by first an associative lookup over requested size and later by size class. It stores the corresponding size class, and the first two free-list elements for that size class.

these cases, the most critical operations are the two loads on the pop path, because long-latency load misses can stall execution and commit of younger instructions. Since calls to the allocator are interspersed among application code, the free lists are prone to eviction, making these loads likely to miss. Figure 4 demonstrates this clearly with the `antagonist` microbenchmark, which emulates such cache-trashing behavior, and sees a significant increase in Pop time. In contrast, stores misses are less likely to stall the execution or commit of younger instructions, making the deallocation path less performance-critical.

TCMalloc uses a trick to save memory taken up by the free lists: it stores the next pointer at the address of the block of memory it is about to return, instead of allocating a separate field in a struct for it. That is, `*head` is the value of the `next` pointer, rather than a more familiar list node with fields `node->head` and `node->next`. In addition to reducing allocator memory overhead, dereferencing `head` to get the next pointer has the side effect of prefetching the returned memory block itself, which can likely help the caller.

Sampling For monitoring and debugging purposes, TCMalloc can also sample allocation requests every N bytes. A sampled allocation dumps and stores a stack trace in addition to performing the allocation itself. Sampling is invaluable in a production setting for analyzing memory usage and debugging memory leaks without having to stop, let alone recompile, live jobs, but it adds a measurable overhead to each malloc request, since a counter must be decremented and checked against the threshold each time.

Remaining instructions The three main steps described above account for $\approx 50\%$ of fast path cycles. The remainder are split roughly evenly between: function call overhead (pushing / popping registers), addressing calculations (for example, of a free list in a thread cache) and updates to metadata fields (such as free list lengths and total size). While it is tempting to consider hard-coding the latter two in hardware, this would result in a rather narrow and inflexible accelerator, and severely limit its applicability to either other allocators, or even future revisions of the same allocator.

```
def mcszlookup(ReqSize):
  IsHit = Cache.FindRangeContaining(ReqSize)
  if IsHit:
    SizeClass = Cache[ReqSize].SizeClass
    AllocSize = Cache[ReqSize].AllocSize
    ZF = 1
  else:
    ZF = 0
  return SizeClass, AllocSize

def mcszupdate(ReqSize, AllocSize, SizeClass):
  IsHit = Cache.FindSizeClass(SizeClass)
  if IsHit:
    SizeRange = Cache[SizeClass]
    if not SizeRange.Contains(ReqSize):
      SizeRange.LowerBound = ReqSize
  else:
    if Cache.Full():
      Cache.Evict()
    SizeRange = (ReqSize, AllocSize)
    Cache.InsertRange(SizeRange, SizeClass)
```

Figure 9: Pseudocode for size class instructions.

4. Mallacc: a malloc accelerator

Based on the characterization in the previous sections, we propose Mallacc, a fast-path malloc accelerator to augment a general-purpose processor. Broadly, Mallacc consists of a tiny dedicated malloc cache and a sampling performance counter. Its design requirements are extremely stringent. Since each fast-path call is on average only a few tens of cycles long, proposed structures must be inside cores, or access latency will erase any gains, which implies very tight area constraints. In addition, we would like to hard-code as few allocator-dependent details as possible (ideally none), so that many current and future allocators can benefit from acceleration. Our proposed design demonstrates that it is possible to meet these constraints, and the rest of this section describes it in detail. Our descriptions assume the x86 architecture, but the general principles and mechanisms are not x86-specific.

4.1 The malloc cache

In Section 3.3, we identified size class computation and popping the head of a free list as the biggest contributors to fast-path cycles, especially with cache-heavy workloads invoking the allocator. We can optimize both with a tiny, two-part cache that we call *the malloc cache*. Conceptually, it learns the mappings from requested allocation sizes to both the size class they correspond to and the first two elements of the free list for that size class. In the case of a malloc cache hit, computation can almost immediately return to the caller. By only storing the most frequently-accessed size classes, the cache can be kept extremely small (several entries). Lookups, updates and prefetches in the cache are software-managed, so it is also not tied to a particular allocator implementation. A four-entry cache, populated with some example values, is shown in Figure 8. We will go over its main components.

Size class mappings By definition a single size class represents a range of allocation sizes that get rounded up and given the same amount of memory. The malloc cache learns and stores the mappings from a requested size range to the size class representing it.

When a requested size comes in, it is associatively checked with the upper and lower bounds of the ranges that currently make up all cache entries. If the request size is inside a range, the access is declared a hit, and the cache returns the size class and its corresponding rounded size. More interestingly, on a miss, execution goes to a fallback path, where software is left to compute the size class as it ordinarily would. Software is then responsible to update the cache with the new (requested size, allocated size, size class) entry. The cache either inserts a new size class entry with the new range, or it expands the range for an already existing size class. If the cache is full for an insertion, an old entry is evicted based on an LRU policy.

The cache is managed by two new instructions: `mcszlookup` and `mcszupdate` (malloc cache size lookup/update). `mcszlookup` takes the requested allocation size in an input register and returns the size class and allocation size in two output registers if the requested size is found in the cache. The zero flag (ZF) is set if found and cleared if not. `mcszupdate` takes the original requested size, the computed size class, and the allocation size in three input registers and either inserts a new entry into the cache or updates an existing one. No registers are modified. Pseudocode for the instruction mnemonics is shown in Figure 9. Figure 10 is an assembly snippet demonstrating how they integrate with the rest of allocator code.

Our actual implementation has one additional optimization – instead of keying the array on the actual requested size range, we use the range of size class indices, as defined in Figure 5, and add dedicated hardware to compute the index from the requested size. Because the space of indices is significantly smaller than the space of requested sizes, the cache can learn mappings faster, with fewer cold misses. The hard-coded hardware adds an additional cycle of latency to the cache, which we do model. It is the only TCMalloc-specific optimization in Mallacc, and can be disabled with a configuration register. In this mode, the malloc cache will build ranges of known requested sizes, although with slightly higher miss rates.

Free list caching An allocation call requires popping an element off a free list. As mentioned in Section 3.3, this is the most performance-critical part of a fast-path call, because it caches poorly and accesses memory prone to eviction by the application's own cache accesses. The malloc cache tackles this bottleneck by storing copies of the head and the *next* head of the free list associated with a size class alongside the size class mappings. Figure 8 illustrates this with an example.

```
Start:
  ; rax = size class (dest)
  ; rbx = allocated size (dest)
  ; rcx = requested size (source)
  mcszlookup rax, rbx, rcx    ; Sets ZF
  je ComputeSizeClass         ; if ZF = 1, jump.
Resume:
  ; Continue with the rest of malloc.

ComputeSizeClass:
  ; The usual software calculation for the size class (rax)
  ; and allocated size (rbx). Then update the cache.
  mcszupdate rcx, rbx, rax
  jmp Resume
```

Figure 10: Integration of size class instructions in an allocator.

Storing the first *two* list items in the malloc cache allows a Mallacc-enabled allocator to immediately return a value to the application after a hit. It also allows the next instruction in a linked list pop, the one that sets the head of the linked list to the current next element, to commit immediately without waiting for an often-required L2 or L3 miss in order to first fetch that next element. We find that second effect especially important, because the long-latency miss often blocks other otherwise-ready instructions from committing.

We introduce two new instructions to enable such operation. Most importantly, `mchdpop` (Figure 11) takes in the requested size class as an input (which we have ideally gotten from the previous optimization), and returns the cached copies of the first two list elements on a hit. If either of them is not present (NULL) in the cache entry, the access is declared a miss, the other one is also invalidated, and execution falls back on software to perform the pop (Figure 12). Its dual operation, `mchdpush`, is invoked on the deallocation side and updates the cached `Head` with the pointer being freed, shifting the previous head to the `Next` slot.

Note that these instructions are merely performance optimizations meant to isolate free lists from cache antagonists. The real free list head pointer is always valid and updated in software on both a hit and a miss, as is any metadata (length, total size, etc.).

For a pop operation to consistently hit, we need two elements already cached. To maintain that for differently-balanced allocation patterns (i.e., with different rates of allocations and deallocations over time), we introduce yet another instruction, `mcnxtprefetch`. Conceptually, `mcnxtprefetch` prefetches a memory location into the malloc cache's `Next` entry, and is called after a pop hits and moves its `Next` element in the `Head` position. In this case, a subsequent pop request can immediately hit as long as the prefetch has had enough time to return from the cache hierarchy. While not necessary for correctness, enabling a prefetch to update the `Head` field of an empty cache entry as well as the `Next` field allows for the prefetch instruction to be called on a miss, and leads to higher hit rates. We assume that in

```
def mchdpop(SizeClass):
  Found = Cache.FindSizeClass(SizeClass)
  if Found:
    Head = Cache.GetHead(SizeClass)
    Next = Cache.GetNext(SizeClass)
    if Head and Next:
        Cache.SetHead(SizeClass, Next)
        Cache.InvalidateNext(SizeClass)
        ZF = 1
        return Head, Next
  ZF = 0
  return NULL, NULL

def mchdpush(SizeClass, NewHead):
  FoundEntry = Cache.FindSizeClass(SizeClass)
  if FoundEntry:
    CurrHead = Cache.GetHead(SizeClass)
    Cache.SetNext(SizeClass, CurrHead)
    Cache.SetHead(SizeClass, NewHead)

def mcnxtprefetch(SizeClass, NewNext):
  CurrHead = Cache.GetHead(SizeClass)
  CurrNext = Cache.GetNext(SizeClass)
  if CurrHead and not CurrNext:
    Cache.SetNext(NewNext)
  elif not CurrHead:
    Cache.SetHead(NewNext)
```

Figure 11: Pseudocode for linked list instructions.

```
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
```

Figure 12: Integration of linked list instructions in an allocator.

Figure 12. Finally, to ensure that the copies of the list elements stored in the malloc cache are always consistent (`Head` always points to `Next`), entries with an outstanding prefetch block and do not service pushes or pops until the prefetch comes back, or gets rolled back on a misprediction.

Core integration First, it is important to point out that the malloc cache only stores copies of list elements for fast access – the definitive version of free lists is always in regular memory. Thus, at interrupts or context switches, the whole cache can always be flushed without writebacks or correctness concerns. Similarly, at branch mispredictions, entries from the mispredicted epoch can be discarded, just as they would in any other long-latency unit.

Second, as part of the core, the malloc cache can potentially see instructions out-of-order. In order to not break the invariant that a cached `Head`'s next pointer always points to the adjacent `Next` element, our three linked list instructions are ordered with each other. We implement that by an implicit read-write register dependency through an architecturally-invisible register, which out-of-order execution has to observe. As discussed earlier, blocking the cache when a prefetch is outstanding is also required to preserve the linked list invariant.

Finally, the prefetch instruction is slightly unconventional. Like a software prefetch in L1, it is allowed to commit, so that it does not block subsequent code, but a result still has to make its way from the cache hierarchy to the malloc cache. From the core's point of view, this is treated in a virtually identical manner to a *store*, which is also allowed to commit with an outstanding memory access, but reserves a slot in a structure (sometimes called a senior store queue), where it waits for an acknowledgment from coherency controllers.

4.2 Sampling

The sampling code in TCMalloc (and its equivalents in `jemalloc` [9] and others) presents an additional opportunity to remove several cycles from the allocation critical path. The operation performed by the sampler – accumulate a value and capture a stack trace at a threshold – is precisely what a performance counter does and what the `perf_events` subsystem performs when the performance monitoring unit (PMU) raises an interrupt on an event. We propose dedicating a hardware performance counter for sampling allocation sizes, which entirely removes a conditional branch on the fast path. Stack traces are only required when a user explicitly requests them, and this can be handled through the `perf_events` interface as it typically is currently.

The main difference between our proposal and current performance counters is that it will need to increment by the value of a register, which holds the requested allocation size. As a result, the PMU will need access to the actual register file (or just the ROB), instead of the more typical occupancy statistics. As the design of a performance counter is fairly straightforward, we will focus on the design of the malloc cache for the remainder of this paper.

	cycle error (%)
gauss	5.32
gauss_free	3.67
tp	12.3
tp_small	5.92
sized_delete	4.21
Average	6.28

Table 1: Simulator validation on `malloc` microbenchmarks.

5. Methodology

To evaluate Mallacc, we ran simulations on two systems – a conventional aggressive out-of-order processor as a baseline, and the same processor equipped with Mallacc, as described in the previous section. We also performed limit studies on our optimizations for an optimistic upper bound of speedup. To do so, we ran simulations in which the instructions comprising the three steps from Section 3.3 are simply ignored by performance simulation.

Microbenchmarks To better understand allocator performance and the effect from our optimizations, we first constructed a suite of microbenchmarks to stress certain aspects of the fast path code. They are divided into two categories based on their allocation patterns: strided and Gaussian. Strided benchmarks allocate in increments of N bytes, up to some value, while Gaussian benchmarks issue allocation requests by drawing from normal distributions. All strided benchmarks fit perfectly in L1 caches and represent the very best baseline cases. Gaussian benchmarks allocate more and subsequently have larger working sets and more interesting caching behavior.

- `tp`: A throughput-oriented microbenchmark. It performs a series of back-to-back `malloc` and `free` calls, which always hit in thread caches. Each iteration strides through request sizes in increments of 16 bytes from 32 to 512 bytes.
- `tp_small`: Same as above, but we only stride up to 128 bytes. This ensures that: (i) each iteration accesses a different free list; and (ii) we only use four size classes. This microbenchmark captures the fastest possible fast path on the allocation side.
- `sized_deletes`: A variant of `tp_small` that uses eight size classes and sized deletes to speed up deallocation.
- `gauss`: A more realistic allocation pattern. `gauss` chooses randomly between small (16-64 byte) and relatively large (256-512 byte) allocations. 90% of allocations are chosen from the small set to represent typical behaviors for strings and small lists. Within each range, the size is selected from a Gaussian distribution. However, no objects are ever freed, which renders free lists virtually useless. This is a lower bound on the gains from any free-list centric optimizations.
- `gauss_free`: Same allocation behavior as `gauss`, but it randomly frees allocated memory with 50% probability.
- `antagonist`: Same allocation behavior as `gauss_free`, but after every allocation, invokes a simulator callback which evicts the less used half of each set of the L1 and L2 data caches. This mimics the behavior of an application that strides through a large working set, without simulating the millions of instructions required for the stride.

All microbenchmarks explicitly minimize the number of instructions between allocator calls (which is important when each call is only 40 instructions) and are run with sufficient warmup time.

Macrobenchmarks We evaluate our optimizations on the four benchmarks from SPEC CPU2006 that use the system allocator and two workloads that are more likely to be found in datacenters. For datacenter-like workloads, we use the open-source search engine `xapian` and the key-value store `masstree` [18]. `xapian` is configured as a leaf node and performs searches on an index of 1.6 million English Wikipedia pages, as well as on a smaller index of the same number of page abstracts. The set of queries focuses on popular Wikipedia pages, obtained from Wikipedia's weekly top 25 article digests. For `masstree`, we run its `wcol1` and `same` performance tests. For SPEC benchmarks, we simulate several SimPoints [23] of 1B instructions each per benchmark, for `xapian` we skip query parsing and only simulate query execution, and for `masstree` we run from start until completion.

Allocator We use TCMalloc at revision `050f2d`. To model our proposed instructions, we annotate potential optimization sites in TCMalloc code by inserting special x86 marker instructions. Later, in simulation we replace these instructions with our proposals. These marker instructions were carefully chosen to a) not already appear in TCMalloc and b) have the same number and type of operands as our proposed instructions. This is done so the compiler does not optimize surrounding code sub-optimally.

Compiler All benchmarks and allocators are built with GCC 6.1 at -O3 with `-fsized-deallocation`.

Simulator All experiments are run using the XIOSim simulator [13], configured for an aggressive out-of-order core modeled after an Intel Haswell microarchitecture. Since we are evaluating `malloc` fast path code, we validated our performance model on microbenchmarks against a Haswell desktop processor and achieved a mean error of 6.3% (Table 1). We omitted `antagonist` because it uses a simulator callback to emulate cache trashing and does not run natively.

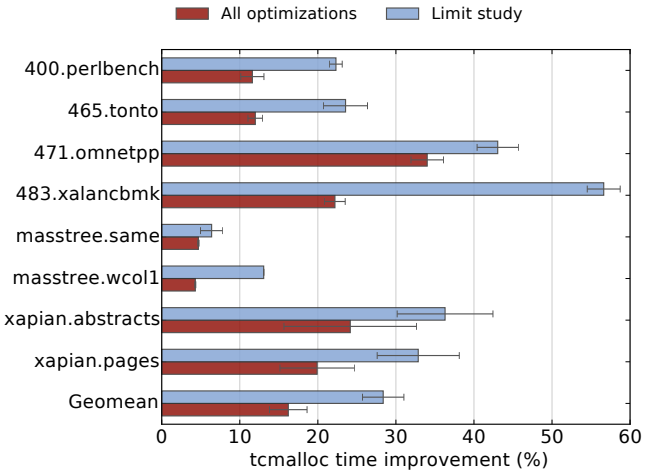

Figure 13: Improvement of time spent in the allocator.

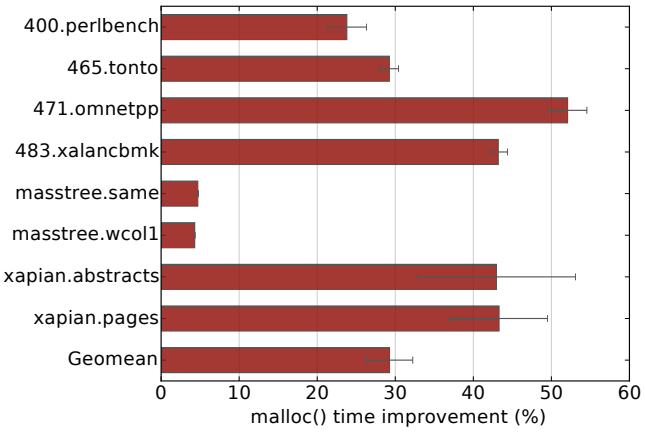

Figure 14: Improvement in time spent on `malloc()` calls (both fast and slow paths).

6. Results

6.1 Allocator time speedup

Figure 13 shows the reduction of time spent in allocator code for our SPEC and cloud workloads. These results use a 32-entry malloc cache to demonstrate the potential of our accelerator; we will later present a cache size sensitivity study. On the total time spent in the allocator (including both `malloc` and `free`), Mallacc is able to achieve an average of 18% speedup, out of 28% projected by the limit study. Most of that is due to improvements on `malloc` calls, which see an average of nearly 30% speedup (Figure 14). The amount of speedup achieved is highly correlated with the fraction of time on the fast path shown in prior sections. We call out three particular benchmarks to get a deeper understanding of the causes for improvement, or lack thereof.

Xapian `xapian` uses a very small set of size classes, and `malloc` calls almost exclusively take the fast path. As shown in Section 3, this is true whether it is searching over an index of small documents (abstracts) or an index of large documents (full articles). This makes `xapian` a great candidate for fast path acceleration and Figure 14 confirms that – the malloc cache provides over 40% speedup on `malloc` calls.

Figure 15 implies that the causes for this improvement are the latency-reducing portions of Mallacc – size class lookups, sampling, and, to a much smaller degree, linked list caching. It is a distribution of time in `malloc` calls over the call duration for three cases: the baseline implementation, our limit study, and Mallacc. The baseline case is already very fast – with virtually all calls between 20 and 40 cycles, not unlike our striding microbenchmarks, which implies very small effects from cache antagonism. Our best-case latency optimizations manage to reduce the average call length almost twofold, with median calls now at 13 cycles, and a distribution very close to that of the limit study. The size class cache in particular is very effective because of the small number of size classes used by `xapian`.

Xalancbmk As demonstrated by Figure 2, `xalancbmk` uses the most number of size classes, requiring 30 size classes for 90% coverage. Nevertheless, it has enough size class locality to also benefit from Mallacc, achieving over 40% speedup on `malloc` calls. Figure 16 shows the `malloc` call duration distribution for this benchmark. The first spike corresponds to the fastest of fast path calls, where the effects are similar to those seen in `xapian`. The next large spike, between 20 and 70 cycles includes fast path calls that missed in L1 and L2 caches and had to go to L3 (34 cycles latency on Haswell). The malloc cache is particularly beneficial in this region because of its cache isolation properties. Finally, note that Mallacc only improves fast-path behavior without affecting slower calls.

Masstree `masstree` has the lowest overall malloc speedup of all the workloads we tested. As we pointed out in Section 3.2, this is because the `masstree` performance tests never free any memory, so many `malloc` calls must request large amounts from the page allocator. The little time spent on the fast path results in an allocator time improvement of just 5%. However, a real deployment of `masstree` would inevitably free memory and likely have significantly higher thread-cache use, so we would expect different results.

6.2 Sensitivity to malloc cache size

The malloc cache is a part of the core, where silicon real estate is expensive, so we must maximize performance gains with the least number of entries. To understand the effects of malloc cache sizing, we sweep malloc cache sizes from 2 to 32 on our suite of microbenchmarks. The results of this sweep are shown in Figure 17.

Unsurprisingly, we find that too small of a cache will result in slowdown rather than speedup. At a high enough miss rate, not only is execution going through the fallback paths (the same instructions that we started optimizing away), but also with the additional malloc cache lookups to determine that. However, once the cache is large enough to capture the

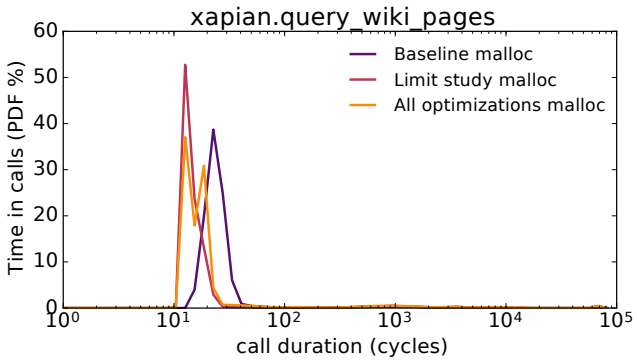

Figure 15: Xapian sees a significant improvement on already-fast calls.

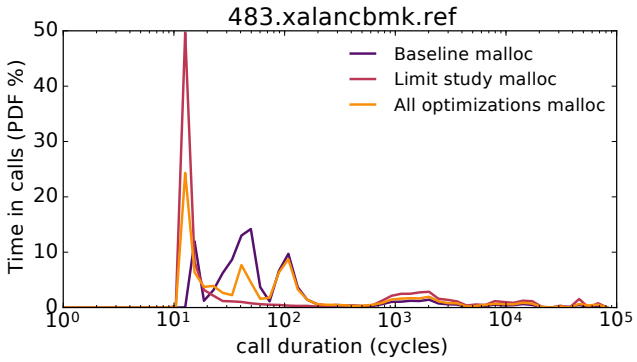

Figure 16: Xalan benefits both from latency reduction and cache isolation.

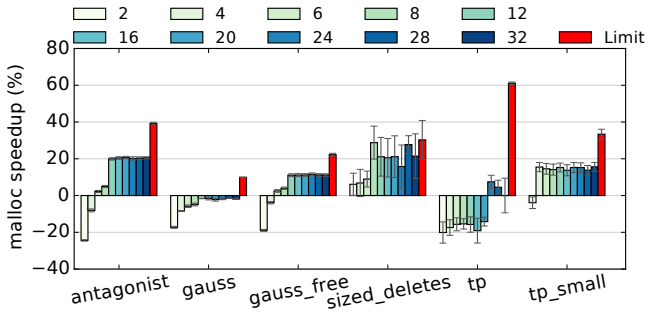

Figure 17: Effect of cache size on malloc speedup.

majority of allocation requests, we quickly achieve speedup. One example are the strided benchmarks, which have *no* size class locality until we can capture *all* of their requests, resulting in very sharp jumps. `sized_deletes`, `tp`, and `tp_small` use 8, 25, and 4 size classes, respectively, and we see that the speedup inflection points occur precisely at those malloc cache sizes. The Gaussian benchmarks have more size class locality because they are more likely to allocate small size classes, which results in a more gradual increase in speedup until cache size 12, because Gaussian benchmarks allocate from 13 possible size classes.

Once the malloc cache is sufficiently sized, Mallacc can achieve within 10-20% of ideal speedup. The lone exception is `tp`. For certain points of execution, this microbenchmark allocates and deallocates from the same size class in a very tight loop ($\approx$ 30 cycles for a malloc-free pair). In this case, the malloc cache blocks until each of the malloc prefetches returns with a value, causing the slowdown. The prefetch instruction is based on exactly the opposite assumption – that there is enough time between requests to prefetch for the next one and this slowdown is expected. None of our macro workloads exhibit slowdown due to prefetch blocking.

It is important to remember that these microbenchmarks are designed to stress the fast path of malloc, not to exhibit realistic allocation behavior. As we showed in Figure 6, most benchmarks use a very small number of size classes. We swept malloc cache sizes and only `xalancbmk` is meaningfully affected by a smaller size – it loses 6 percentage points of allocator time improvement between 32 and 16. Because of that, we consider 16 sufficient for most workloads.

6.3 Full program speedup

Finally, we present improvements on full benchmark execution time, not only allocator time. This speedup is obviously bounded by the total time each benchmark spends in the allocator. Figure 18 shows these fractions for our workloads, compared to published data from Google's datacenters [12]. Most of our workloads spend a much lower fraction of time in allocator code, so we can expect small gains. As mentioned before, the `masstree` performance tests have very high `malloc` time because they exclusively allocate memory and never free any, resulting in many slow path calls.

Table 2 shows full program speedup for workloads where the measured speedup through simulation is statistically significant. For them, the mean program speedup is 0.43%, with a maximum of 0.78% for `perlbench`.

Because absolute time in the allocator speedup tends to be small, run-to-run variance on some of the workloads is enough to mask out any improvements we achieved with the `malloc` accelerator. More precisely, we do not include the workloads for which a single-sided Student's T-test fails to reject a hypothesis of full-program slowdown with 95+% probability. Note that for all workloads, the speedup *in allocator code* is always statistically significant: in Figure 13 the improvement in allocator time is always much higher than typical run-to-run variation (error bars), even so if we pessimistically add simulation bias (6% from Table 1) in the least favorable direction. Non-allocation code is unchanged between our baseline and optimized experiments, but it dominates execution time, so even small random variations in simulating it can appear to mask all the gains in allocator code. This is why we prefer reporting gains in the allocator alone, where we can be certain in the significance of results.

6.4 Area cost of Mallacc

Mallacc consists of the malloc cache and a performance counter. The malloc cache requires 152 bits of storage per entry. Because the malloc cache is fully associative, it must be implemented using content addressable memories (CAMs) for lookup and standard SRAM for storage. We do not lay out the malloc cache to provide precise area estimates, but it is so small that a reasonable upper bound will suffice. Also, we ignore the area of the performance counter, since it is just one 64-bit register per hardware thread.

The malloc cache requires three CAM arrays to implement the index and size class search and LRU functions, while the rest of the data – allocated size and list pointers – can be stored in an SRAM array. The index CAM requires 24 bits per entry to store two 12-bit indices, while the size class CAM requires 8 bits per entry to store size classes, and the LRU CAM stores $\log_2 n$ bits per entry, where n is the number of entries. The SRAM array requires 117 bits per entry to store two 48-bit pointers (currently, x86 only uses the lower 48-bits of 64-bit addresses), 20 bits for the allocated size, plus a valid bit. Our analysis has shown 16 entries to be sufficient for the workloads analyzed; this means the CAMs and SRAM are 72 bytes and 234 bytes, respectively.

We used CACTI 6.5+ [15] to estimate the sizes of these four arrays in 28nm. The CAMs collectively occupy 873 μm^2 and the SRAM occupies 346 μm^2 for a total of 1219 μm^2. This is certainly a pessimistic upper bound; Jeloka et al. recently demonstrated a 512 byte configurable CAM array occupying merely 1208 in 28nm μm^2 [25]. We scale published area numbers of shifters and adders (for the additional index computation) from accelerator models [22] by ITRS technology scaling factors and estimate a total area of 265 μm^2, bringing our upper bound to about 1500 μm^2.

Consider this area in the context of a typical high-performance CPU. An Intel Haswell core measures 26.5 mm^2 (including private L1 and L2 caches). If integrated into a Haswell CPU, Mallacc is merely 0.006% of the core area. Pollack's Rule states that historically, the performance increase of a chip is approximately proportional to the square root of the increase in complexity, where complexity refers to area [2]. By this rule, an area increase of 0.006% would only produce 0.003% speedup. In contrast, Mallacc demonstrates average speedup of 0.43%, which is over 140× greater. It is clear that Mallacc far surpasses the "1% performance for 1% area" rule of thumb that has informally guided processor development over the last few decades.

7. Conclusion

Dynamic memory allocation is a widely used programming paradigm that has seen decades of software research and optimization. Recent work has discovered that despite being well-optimized, memory allocation can consume a significant percentage of datacenter cycles. In this work, we present Mallacc, a tiny in-core hardware block for accelerating dynamic memory allocation. Mallacc does not implement a new allocator; rather, it is designed to accelerate various operations that are common to many existing high-performance allocators. Unlike many hardware accelerators that target maximum throughput, Mallacc is designed to minimize latency. We show that Mallacc can accelerate the most commonly observed `malloc` behavior – fast allocation requests that only take 20-30 cycles on modern processors – by up to 50%, while consuming less than 1500μm^2 of silicon area. Integrating Mallacc into a CPU provides speedups that greatly outstrip the typical "1% performance for 1% area" Pollack's Rule tradeoff.

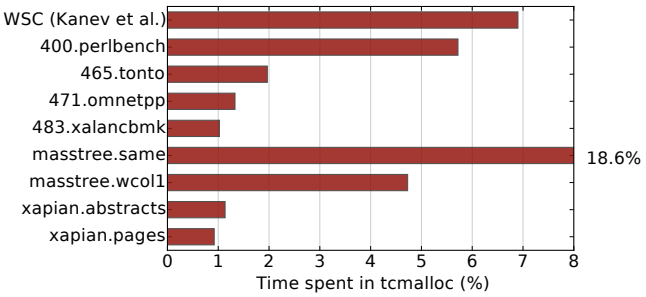

Figure 18: Fraction of time spent in the allocator.

	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
`xapian.pages`	0.16%	0.02%	0.012

Table 2: Full program speedup.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback and suggestions. We reserve our special thanks for the TCMalloc team at Google, and specifically Aliaksei Kandratsenka, Darryl Gove, Tipp Moseley and Parthasarathy Ranganathan, for many technical discussions and feedback on early versions of this manuscript.

This work was partially supported by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. S. Kanev was partially supported by a Siebel Scholarship and S. Xi was partially supported by a National Science Foundation Graduate Fellowship. The work was also supported by generous gifts from Google and Intel Corporation. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government or any other sponsor.

References

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[2] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 2011.

[3] Hasan Cam, Mostafa Abd-El-Barr, and Sadiq M Sait. A high-performance hardware-efficient memory allocation technique and design. In *Computer Design (ICCD)*, 1999.

[4] J. Morris Chang and Edward F Gehringer. A high performance memory allocator for object-oriented systems. *Transactions on Computers*, 1996.

[5] J Morris Chang, Witawas Srisa-An, and C-TD Lo. Architectural support for dynamic memory management. In *Computer Design (ICCD)*, 2000.

[6] George O Collins Jr. Experience in automatic storage allocation. *Communications of the ACM*, 1961.

[7] WT Comfort. Multiword list items. *Communications of the ACM*, 1964.

[8] Jason Evans. A Scalable Concurrent malloc Implementation for FreeBSD. In *Proceedings of the Technical BSD Conference*, 2006.

[9] Jason Evans. Scalable memory allocation using jemalloc. https://goo.gl/rvl2oK, 2011.

[10] T.B. Ferreira, R. Matias, A. Macedo, and L.B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2011.

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

[12] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Tipp Parthasarathy, Ranganathan amd Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA)*, 2015.

[13] Svilen Kanev, Gu-Yeon Wei, and David Brooks. XIOSim: power-performance modeling of mobile x86 cores. In *Low-power electronics and design (ISLPED)*, 2012.

[14] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10), 1965.

[15] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture (MICRO)*, 2009.

[16] Wentong Li, Saraju P Mohanty, and Krishna Kavi. A page-based hybrid (software-hardware) dynamic memory allocator. *Computer Architecture Letters (CAL)*, 2006.

[17] Wentong Li, Mehran Rezaei, Krishna Kavi, Afrin Naz, and Philip Sweany. Feasibility of decoupling memory management from the execution pipeline. *Journal of Systems Architecture*, 2007.

[18] Yandong Mao, Eddie Kohler, and Robert Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[19] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA)*, 2014.

[20] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *Computer Architecture (ISCA)*, 2013.

[21] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Computer Architecture (ISCA)*, 2016.

[22] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. The Aladdin Approach to Accelerator Design and Modeling. *IEEE Micro*, 2015.

[23] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Computer architecture (ISCA)*, 2002.

[24] CJ Stephenson. New methods for dynamic storage allocation (fast fits). In *Operating systems principles (SOSP)*, 1983.

[25] Supreet Jeloka and Naveen Bharathwaj Akesh and Dennis Sylvester and David Blaauw. A 28nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. *Journal of Solid-State Circuits (JSSC)*, 2016.

[26] M. Tadman. Fast-fit: A new hierarchical dynamic storage allocation technique. Master's thesis, 1978.

[27] Paul R Wilson, Mark S Johnston, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *International Workshop on Memory Management*, 1995.