parallel memory allocation, timescale statistics, higher order theory of memory demand

Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA

Timescale Functions for Parallel Memory Allocation

Pengcheng Li, Hao Luo, Chen Ding
Pengcheng Li
Google Inc.
Mountain View, California, USA
landy0220@gmail.com

Hao Luo
Google Inc.
Mountain View, California, USA
haoluo@google.com

Chen Ding
University of Rochester
Rochester, New York, USA
cding@cs.rochester.edu

Abstract

Memory allocation is increasingly important to parallel performance, yet it is challenging because a program has data of many sizes, and the demand differs from thread to thread. Modern allocators use highly tuned heuristics but do not provide uniformly good performance when the level of concurrency increases from a few threads to hundreds of threads.

This paper presents a new timescale theory to model the memory demand in real time. Using the new theory, an allocator can adjust its synchronization frequency using a single parameter called allocations per fetch (*apf*). The paper presents the timescale theory, the design and implementation of APF tuning in an existing allocator, and evaluation of the effect on program speed and memory efficiency. APF tuning improves the throughput of *MongoDB* by 55%, reduces the tail latency of a Web server by over 60%, and increases the speed of a selection of synthetic benchmarks by up to 24× while using the same amount of memory.

CCS Concepts •**General and reference → Metrics; Performance;** •**Software and its engineering → Main memory;**

Keywords memory allocator, timescale, APF, HOTM

ACM Reference format:
Pengcheng Li, Hao Luo, Chen Ding, Pengcheng Li, Hao Luo, and Chen Ding. 2019. Timescale Functions for Parallel Memory Allocation. In *Proceedings of Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management, Phoenix, AZ, USA, June 23, 2019 (ISMM '19),* 15 pages.
DOI: 10.1145/3315573.3329987

1 Introduction

In parallel C/C++ applications, dynamic memory allocation is ubiquitous, frequent and expensive. Studies of past and recent systems have found that the time spent in allocation and de-allocation ranges from a few percent to 30% in many benchmark programs [3, 7, 19, 54]. Furthermore, memory allocation is critical to the scalability of highly threaded applications such as Web servers [26] and databases [42].

The design of a memory allocator, as exemplified by dlmalloc [34], is a balancing act between many overlapping and even conflicting goals, including speed, low fragmentation, tunability, portability, and compatibility. In concurrent allocation, earlier allocators [30, 34] use a shared heap and serialize allocation and de-allocation using locks. Frequent locking leads to poor scaling [30, 57]. Scalable allocators, Hoard [5], jemalloc [10], tcmalloc [16], SuperMalloc [31] and others [14, 49], give each thread a local reserve, in which allocation and de-allocation are thread local and synchronization free. Synchronization is only needed for a memory fetch, when a thread-local reserve is empty, or a memory return, when it has too much free space.

Table 1. Execution times of tcmalloc on Intel Haswell and IBM Power 8 on two tests from [15], running 1 to 256 threads. A fixed design does not serve all thread counts equally well.

Benchmark/ platform		Exe. time (sec.) for # threads			
		1	16	64	256
MicroServer	Haswell	360	26.9	764	1, 078
	Power8	377	30.5	7,015	>3 hrs.
Producer-Consumer	Haswell	9.79	15.4	79.2	101
	Power8	12.0	9.81	77.2	129

In scalable memory allocations, a basic problem is therefore how much free space to fetch into a local reserve when it is empty. In existing allocators, the choice does not depend on the workload. They follow the tradition and ideal as phrased by Doug Lea that "[an allocator that is] configured using default settings should perform well across a wide range of real loads" [34]. This strategy is effective in single-thread memory allocation, as evident by the success of dlmalloc.

In parallel memory allocation, however, one size does not fit all. Table 1 shows the execution times of tcmalloc [16] on two tests from [15] running 1 to 256 threads on two machines. As more threads are used, the running time is reduced but then increased by two orders of magnitude in the worst case. The allocator works well for some but not all thread counts.

In this paper, we augment a memory allocator with tuning using a single parameter called *allocations per fetch* (*apf*). The parameter specifies the target time interval between consecutive fetches from the central reserve. We use different character styles to distinguish between the technique and the parameter. APF tuning is the technique that realizes a given *apf*. By controlling the fetch frequency, APF removes contention in memory allocation.

Given an *apf*, APF keeps just enough free memory in a thread local reserve — not too little to cause too many fetches, and not too much to waste memory. To determine this optimal amount, we use a new theory to quantify the memory demand. The main contributions of the paper are:

- *The timescale theory*, which models the dynamic memory demand using a set of timescale functions, gives the algorithms for linear-time, online measurement of the memory

ISMM '19, Phoenix, AZ, USA

978-1-4503-6722-6/19/06…$15.00
DOI: 10.1145/3315573.3329987

demand, and proves its monotonicity, which is needed for tuning (Section 3).
- *Design and implementation of APF tuning*, which controls the size of thread-local reserve for each thread and each size class in a principled way based on the memory demand (Section 4).
- *Optimal APF*, which is an offline algorithm to compute the optimal local reserve size (Section 5).
- *Evaluation*, which adds APF in an existing allocator, tcmalloc, and evaluates it on two applications and six synthetic benchmarks for up to 128 threads on two types of machines (Section 6) for speed, memory consumption, and for a Web server, tail latency.

In the rest of the paper, we present the objective in Section 2, the four main contributions in Sections 3 to 6, and related work in Section 7 before summarizing at the end.

2 Thread-local Memory Reservation

A program has a number of threads. To a memory allocator, each thread is a sequence of heap operations. For our purpose, we consider only the operations of allocation and de-allocation, also called a free or a reclamation. The unit of an allocation or a free is an object. The memory location holding an object is a heap slot.

A heap slot belongs to a set of pre-defined *size classes*. A *memory reserve* is a list of free heap slots of the same size class. A parallel allocator creates for each thread a set of *thread-local reserves*, one for each size class. Allocations and frees in thread-local reserves are synchronization-free. In addition, the allocator has a set of *central reserves*, one for each size class, shared by all threads.[1]

The number of slots in a reserve is the *reserve size*. When it is zero, the next allocation triggers a memory fetch from the central reserve. The number of free heap slots brought in is the *fetch size*, which is the *initial reserve size*. When a local reserve has too many free slots, it returns some of them to the central reserve. Fetch and return require synchronization.

Problem Statement We have two goals: memory efficiency and parallelism. Memory efficiency means to limit the amount of unused memory. For memory reservation, we measure the *inefficiency* by the total size of all thread-local reserves. This total size puts an upper bound on the *blowup*, which, as defined by Berger et al. [5], happens when a heap slot is available in the reserve of one thread but cannot be used by another thread which needs to allocate memory. The second goal, parallelism, is measured by the frequency of synchronization, i.e., the number of fetches and returns.

We solve an *optimization problem*: given a desired frequency of synchronization, how to minimize the free memory in each thread-local reserve. The solution enables tuning: A user or a tool can tune the allocator to support a degree of parallelism with minimal memory reservation.

Allocations Per Fetch (APF) We use capital letters APF to denote the technique and the italicized non-capital *apf* to denote the numerical value, calculated by dividing the number of allocations by the number of fetches. *APF tuning* means trying different *apf* values. The measure of *apf* was first defined in the higher-order theory of memory demand (HOTM) [40], which we discuss in Section 3.7.

[1] A reserve is also called a cache. In tcmalloc, each thread has a *thread-local cache*, which includes all its local reserves, and a *central cache*, with all the central reserves [16].

3 A Theory of Timescale Functions

We formalize the memory demand by defining a set of timescale functions and using them to represent the dynamic memory demand. For the theoretical development, we consider the memory demand for a single size class by a single thread.

3.1 From Time Series to Timescale Functions

A *timescale* is a length of time. A *timescale function* $f(k)$ is the *average* of the per-window measure of *all* windows of length k. Figure 1 shows an illustration with some heap dynamics, e.g. the number of live objects, over time in a time series, i.e. a series of data points indexed by time. Assume that we measure a time window by the total number of objects live in the time window. The graph shows three windows of the same length, i.e. the same timescale. The object count changes from window to window. To find *representative* liveness, we use the timescale function $f(k)$, which is the average object count for all windows of length k.

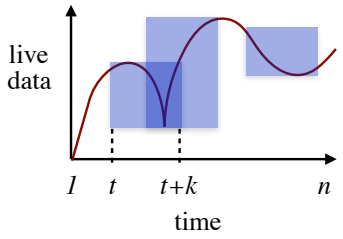


Figure 1. A time series and three example windows of length k. A timescale function $f(k)$ shows the average of all windows of length k.

A *timescale function* $f(k)$ shows the average (expected value) for all timescales $0 \le k \le n$, where n is the maximal time length. It is interesting to compare with a time series, e.g. the one in Figure 1. The time in a time series increments from 1 and n or from 0 to $n-1$. The timescale for such time series ranges from 0 to n. Gradation comes with representation. Although per window measures are not monotone, timescale measures are, as shown in Section 3.6.

Next, we define two timescale measures, *reuse* and *allocation*, and use them to compute the memory demand in Section 3.4. For a direct definition of the memory demand, a reader can refer to the one at the end of Section 3.4.

3.2 Memory Reuse

Consider the sequence of allocations and de-allocations for a single reserve. We define the *logical time* to start at zero and increment at each event. A *time window* is specified by the start and the end events and includes these and all events in between. The length of a window is its end time minus its start time plus one.

We define a memory reuse in a time window as follows:

Definition 1. Intra-window memory reuse *In a time window, if a new object is allocated a heap slot vacated (freed earlier) in the window, we call this allocation an intra-window reuse.*

The number of intra-window reuses may vary from window to window. To summarize all reuse behavior, we define the timescale

measure $reuse(k)$ as the average number of intra-window reuses in all windows of length k.

A Running Example Consider a simple program that repeatedly allocates and frees three objects. It needs three heap slots, numbered 1 to 3. The execution is an infinite sequence of allocation and free operations $\ldots a_1 a_2 a_3 f_3 f_2 f_1 \ldots$, where the subscript indicates the heap slot allocated or freed. This example uses the convention that the heap slot last freed is first allocated, but the theory does not depend on this convention.

To simplify the explanation, we use a type of logical time called the *allocation clock*, used by Li et al. in theorizing the memory demand [40]. An execution is a series of allocation and free operations, but only the allocations increment the logical time. Free operations happen at the time of the next allocation. At a time point, there is one and only one allocation, with zero or more free operations.

Definition 1 counts the reuses in a window, i.e. the number of the allocations satisfied due to the free operations in the window. In the running example, the window $a_2 a_3$ has no free operation, so the intra-window reuse must be 0. In the window $a_3 f_3 f_2 f_1 a_1 a_2$, the last two allocations reuse a heap slot, so the intra-window reuse is 2.

It is worth pointing out that the length of $a_3 f_3 f_2 f_1 a_1 a_2$ is 3 (not 6) according to the allocation clock. The theory does not depend on any particular type of logical time. The following definitions and calculations are given for any definition of logical time.

We next give an efficient solution to measure $reuse(k)$. Let n be the length of an execution. Hence, $x \in 1 \ldots n$. The total number of windows is quadratic, i.e. $\binom{n}{2} = \frac{n(n-1)}{2}$. A direct solution would take at least quadratic time just to enumerate all the windows. Instead, we transform the problem to make it solvable in linear time.

3.2.1 Computing $reuse(k)$ in Linear Time for Any k

Given a trace of n events, $reuse(k)$ is the average number of intra-window reuses of all windows of length k for $0 \le k \le n$. First, we view a memory reuse as a free interval:

Definition 2. Free interval *If a heap slot is freed at time s and then allocated at time e, the window $[s, e]$ is a free interval.*

For each heap slot at each allocation, there is a free interval (except for the first allocation to the slot). The number of intra-window reuses in a window is the number of free intervals that are contained in the window. The counting of intra-window reuses is the same as counting the free intervals. We call it interval counting:

Definition 3. Interval Counting per Window *For a window $[x, y]$, the interval count is the number of contained free intervals, i.e. interval $[s, e]$ s.t. $x \le s \le e \le y$.*

Interval counting is costly, because the total number of windows is quadratic. The "trick" of efficient counting is to convert the problem from interval counting to window counting, defined as follows:

Definition 4. Window Counting per Interval *For a free interval $[s, e]$, the window count is the number of enclosing windows, i.e. window $[x, y]$ s.t. $x \le s \le e \le y$.*

It is easier to count the enclosing windows for all intervals, because the total number of intervals is linear (at most one interval per allocation).

In window counting, an execution is a collection of m free intervals, $[s_i, e_i]$, for $i \in 1 \ldots m$. For a window length k, we classify these intervals into a number of categories. For example, if the interval length is greater than the window length k, i.e. $[s_i, e_i]$, $e_i - s_i > k$, the window count is 0. Otherwise, we compute the specific window count.

Eq. 1 shows the complete solution. The first two lines show the conversion from interval counting to window counting, and the remaining two lines show the final result.

$$\begin{aligned} reuse(k) &= \frac{\sum_{\text{all windows } w} (\text{number of free intervals in } w)}{n-k+1} \\ &= \frac{\sum_{\text{all intervals } [s,e]} (\text{number of windows enclosing } [s,e])}{n-k+1} \\ &= \frac{\sum_{i=1}^{m} I(e_i - s_i \le k)(\min(n-k, s_i))}{n-k+1} \\ &+ \frac{\sum_{i=1}^{m} I(e_i - s_i \le k)(-\max(k, e_i) + k + 1)}{n-k+1} \end{aligned} \tag{1}$$

where $I(p)$ is a predicate function that equals 1 if p is true and otherwise 0. In Eq. 1, $I(e_i - s_i \le k)$ selects only free intervals whose length is k or shorter.

Eq. 1 has a linear time cost, $O(m)$, for any one k. It is quadratic, $O(mn)$, for all ks. Next, we show a linear-time solution for all k.

3.2.2 Computing $reuse(k)$ in Linear Time for All k

We first separate the numerator of Eq. 1 into three terms:

$$\begin{aligned} X(k) &= \sum_{i=1}^{m} I(e_i - s_i \le k) \min(n-k, s_i) \\ Y(k) &= \sum_{i=1}^{m} I(e_i - s_i \le k) \max(k, e_i) \\ Z(k) &= \sum_{i=1}^{m} I(e_i - s_i \le k)(k+1) \end{aligned}$$

so that

$$reuse(k) = \frac{X(k) - Y(k) + Z(k)}{n-k+1} \tag{2}$$

$$X(1) = \sum_{i=1}^{m} I(e_i - s_i = 1)s_i$$

$$X(k) = X(k-1) - \sum_{i=1}^{m} I(s_i \geq n - (k-1)) + \sum_{i=1}^{m} I(e_i - s_i = k)\min(n-k, s_i)$$

$$Y(1) = \sum_{i=1}^{m} I(e_i - s_i = 1)e_i$$

$$Y(k) = Y(k-1) + \sum_{i=1}^{m} I(e_i \leq k-1) + \sum_{i=1}^{m} I(e_i - s_i = k)max(k, e_i)$$

$$Z(1) = 2\sum_{i=1}^{m} I(e_i - s_i = 1)$$

$$Z(k) = Z(k-1) + \sum_{i=1}^{m} I(e_i - s_i \leq k) + k\sum_{i=1}^{m} I(e_i - s_i = k)$$

In the above, we show the recursive formulas to compute $X(k)$, $Y(k)$ and $Z(k)$. For brevity, we save derivation getting those formulas. In each recursive formula, the predicate parts could be precomputed in $O(n)$ for all ks, so for one single k, $X(k)$ can be computed from $X(k-1)$ in $O(1)$ time, so $X(k)$ takes $O(n)$ time for all ks. So do $Y(k)$ and $Z(k)$. Since $X(k)$, $Y(k)$ and $Z(k)$ all take $O(n)$ time, $reuse(k)$ can now be computed in $O(n)$ for all ks.

The Running Example We calculate the timescale function $reuse(k)$ for the running example $\ldots a_1a_2a_3\ f_3f_2f_1\ldots$, using the definition. By the allocation clock, a time window may start at an allocation. The frees happen at the time of the succeeding allocation a_1. Since the trace is infinite, there are just three types of windows, starting at a_1, a_2, a_3 respectively. Let $k = 2$. When the window is a_1a_2, we have two reuses. When it is a_2a_3, there is no reuse. For $a_3f_3f_2f_1a_1$, there is one reuse. Taking the average, we have $reuse(2) = 1$. Following the definition, we see that $reuse(k) = k-1$ for this example for $k \geq 2$.

3.3 Allocation

Liveness measures the amount of data allocated and not freed. At a time point (a snapshot), liveness is called the population count. Li et al. extended this definition to a time window and defined its liveness as the union of the population at all time points of the window [39]. They defined $live(k)$ as the average population count in all length-k windows, and hence, a timescale function, defined formally as follows:

$$live(k) = \frac{\sum_{\text{all window } w} (\text{number of live objects in } w)}{n-k+1}$$

where $k \geq 1$.

Consider the running example $\ldots a_1a_2a_3\ f_3f_2f_1\ldots$ $live(1)$ is the average number of live objects at the time of a memory allocation. If we use the allocation clock, the number of live objects is 1, 2, 3 at the three time points of the window a_1, a_2, a_3 respectively, so the average number of live objects at an allocation is $live(1) = 2$. In general for this example, we have $live(k) = k+1$ for $k \geq 1$. We define $live(0) = live(1) - 1$, which is the number of live objects before each allocation.

We next use liveness to compute the memory demand and will discuss its measurement in Section 3.5.

3.4 Demand = Allocation - Reuse

The memory demand is computed as the number of allocations minus the number of heap slot reuses. In any time window, the frees supply memory for the allocations. The *memory demand* is the number of *extra* heap slots needed to satisfy all its allocations. It is the number of remaining "unsatisfied" allocations.

We have already shown that the allocations and the heap reuses are measured by timescale functions, $live(k)$ and $reuse(k)$. The memory demand can now be computed using these two functions as follows:

$$demand(k) = live(k) - live(0) - reuse(k) \tag{3}$$

where $live(k) - live(0)$ is the average number of allocations in a window, and $reuse(k)$ the average number of allocations that are satisfied due to the frees in the window. The difference is the memory demand. The memory demand is a timescale function computed from other timescale functions.

When we use the allocation clock, the number of allocations is simply the length of the window. The memory demand is computed by

$$demand(k) = k - reuse(k)$$

Direct Definition of Memory Demand The memory demand can be defined directly. For each window, if we traverse the window from start to end and count the number of heap slots occupied at each point, we have a sequence of non-negative numbers. Take the maximum of these numbers. The demand is the maximum minus the number of heap slots occupied at the beginning of the window. The timescale measure $demand'(k)$ is the average of this demand in all windows of length k.

In fact, $demand'(k) = demand(k)$, i.e. Eq. 3 computes the memory demand correctly. Both methods calculate the difference of allocation and reuse. The direct definition computes the difference for each window and then takes the average. Eq. 3 takes the average of allocation and of reuse and then computes the difference. For brevity we omit a formal theorem and proof.

The Running Example Consider the infinite sequence of allocation and free events $\ldots a_1a_2a_3\ f_3f_2f_1\ldots$ By the allocation clock, a time window must start and end at an allocation. The frees happen at the time of the succeeding allocation a_1.

We first consider the direct definition of memory demand $demand'(k)$. Since the trace is infinite, there are just three types of windows, starting at a_1, a_2, a_3 respectively. For windows of length 2 or larger, if we start at a_1, the memory demand is zero, since its allocations are reuses (after the frees). If we start at a_2, the initial heap size is one, the maximal size three, and the memory demand two. If we start at a_3, the initial size is two, the maximal size three and memory demand one. Taking the average, we have $demand'(k) = \frac{0+2+1}{3} = 1$ for $k \geq 2$.

We now show that the timescale functions compute the memory demand correctly for the running example. As explained earlier, $reuse(k) = k-1$ for $k \geq 2$. We have

$$demand(k) = k - reuse(k) = 1$$

where $k \geq 2$.[2]

Eq. 3 can be solved in linear time as we have shown, but the direct definition has no easy solution. It cannot use the idea of window counting described in Section 3.2. Therefore, a key advantage of the new timescale theory is that it solves sub-problems with mathematical functions and the whole problem by combining these functions.

3.5 Online Analysis

So far the analysis is offline because it computes all timescale measures once at the end of execution. Online analysis, however, computes them periodically during execution. For an execution of length n, online analysis computes timescale measures $O(n)$ times. The linear-time offline complexity becomes quadratic-time complexity when used online.

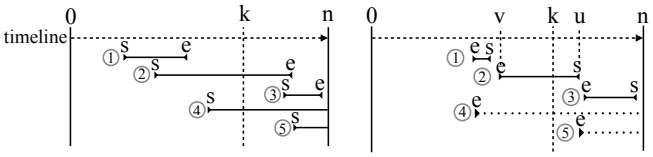
In formal terms, the online analysis computes $live(n, k)$ and $reuse(n, k)$, where n is the *current* time of execution and k is the timescale. At each execution point, we compute only one selected k, which can be any value from 1 to n. Next we show online solutions that take $O(1)$ time at each execution point for both allocation and reuse.

Liveness As introduced earlier (Section 3.3), liveness measures the amount of data allocated and not freed. Li et al. gave the following equation to compute $live(k)$ for any k in linear time [39].

$$live(k) = \frac{\sum_{i=1}^{m} \min(n-k+1, e_i) - \sum_{i=1}^{m} \max(k, s_i) + mk}{n-k+1}$$

where s_i denotes the allocation time of object i, e_i the death time, m the number of objects, and n the trace length.

The solution, however, is quadratic time if used online. Figure 2(a) shows all the five cases when calculating liveness. Up till the current time n, objects 1, 2 and 3 are dead while objects 4 and 5 are not. The reuse is accomplished for 1, 2 and 3 but not for 4 or 5. Since the online time goes only to n, the death times of objects 4 and 5 are assumed to be n. This approximation of future death does not impact liveness computation at time n.



(a) incremental liveness (b) incremental heap-reuse

Figure 2. Two examples to illustrate how to compute liveness and reuse incrementally. A segment of "s-e" denotes an object, where "s" denotes the allocation time and "e" denotes the death time. A segment of "e-s" denotes a reuse, where "e" denotes last free of a heap slot and "s" denotes next allocation to reuse the same heap slot.

We have derived the online analysis as follows.

$$live(n,k) = \frac{\sum E_{n-k+1} + (m - \#E_{n-k+1})(n-k+1)}{n-k+1} + \frac{-(\#S_k k + \sum S_n - \sum S_k) + mk}{n-k+1} \quad (4)$$

[2]The equality holds also for $k = 1$. From direct definition, we have $demand'(1) = 2/3$, because the demand of the three types of windows are 0, 1, 1, when the window length is 1. From timescale functions, we have $reuse(1) = 1/3$, and hence $demand(1) = 2/3$.

where $\#S_k$ the number of allocations before k, $\sum S_k$ the sum of these allocation times, $\#E_k$ the number of frees before k, and $\sum E_k$ is the sum of these free times. These counters and sums are all incrementally accumulated in $O(1)$ time. The space complexity is $O(n)$, where n is total time. If we use the logarithmic scale, the space cost is $O(\log n)$. Figure 3 shows the algorithm to incrementally compute liveness: each step is $O(1)$.

(a) Notations

n: current time, i.e. a timer
m: number of objects
k: target window length
$\sum$**s**$_i$: sum of allocation times before time i
$\sum$**e**$_i$: sum of free times before time i
#s$_i$: number of allocations before time i
#e$_i$: number of frees before time i

(b) Algorithms

```
procedure doAllocation
 1: ∑sn += n
 2: #sn += 1
 3: m += 1
procedure incTimer
 6: n += 1
 7: #sn = #sn-1
 8: ∑sn = ∑sn-1
 9: #en = #en-1
10: ∑en = ∑en-1
procedure doFree
 4: ∑en += n
 5: #en += 1
procedure doLiveness
11: i = n-k+1
12: tmp1 = (m - #ei) × i +∑ei
13: tmp2 = #sk × k + ∑sn - ∑sk
14: return (tmp1-tmp2+m×k)/(n-k+1)
```

Figure 3. Online live analysis implementing Eq. 4

Reuse In incremental liveness analysis, it knows exactly what to do at each allocation and free, because the present computation does not depend on the future. However, it is not the case for memory reuse. For example, in Figure 2(b), at time point k when two reuses 2 and 4 have not happened, we cannot count them in the online calculation because we do not know if they will happen before the next fetch. We do not know the reuse of 2 until time u. Because u is before n, this reuse should be counted before next fetch. Then, we have to go back to update in the equations of time points between v and u at latest at time n, which is not $O(1)$ cost. To be $O(1)$, the online solution at k has to be future-independent, which is impossible. Hence, incremental reuse analysis is impossible. Instead, we solve the problem through sampling.

In bursty sampling, an execution is periodically analyzed for a period, called a burst, and every two consecutive bursts are separated by a hibernation period [2, 6, 22]. We use bursty sampling as follows: in a burst, we collect all reuse intervals and then calculate the timescale reuse using the offline algorithm in Section 3.2.2. This reuse will be used until the next burst. Since the offline algorithm is linear time, the amortized time complexity is $O(1)$ overall. The limitation is that the reuse is measured for the sampling periods, not the whole execution. In practice, we can trade cost for accuracy by choosing the length of burst and hibernation periods.

3.6 Monotonicity

A formal property is that the demand for memory is greater in a larger timescale, as stated by the following theorem.

Theorem 3.1. *(Demand Monotonicity) The timescale function demand(k) defined in Eq 3 is monotonically non-decreasing, assuming the trace length $n \gg k$.*

Proof. We use $demand(k, i)$ to denote the memory demand of the length-k window starting at i. We use the allocation clock here, so i denotes the i-th allocation.

First, for $\forall i$, $demand(k+1, i) \geq demand(k, i)$ holds. Consider the window of length-$(k+1)$ starting at i and its last allocation. If the allocation reuses a slot freed in the window, $demand(k+1, i) = demand(k, i)$, otherwise $demand(k+1, i) = demand(k, i) + 1$.

Then,

$$\begin{aligned} demand(k+1) &= \frac{1}{n-k}\sum_{i=1}^{n-k} demand(k+1, i) \\ &\geq \frac{1}{n-k}\sum_{i=1}^{n-k} demand(k, i) \\ &\approx \frac{1}{n-k}\sum_{i=1}^{n-k+1} demand(k, i) \\ &\geq \frac{1}{n-k+1}\sum_{i=1}^{n-k+1} demand(k, i) \\ &= demand(k) \end{aligned}$$

The reason for approximate equal is that when $n \gg k$, the demand of the last length-k window, $demand(k, n-k+1)$, which is at most k by itself, contributes approximately zero to the average when n is sufficiently large. □

We call the property *demand monotonicity*. From Eq 3, the property is equivalent to $live(k+1) - live(k) \geq reuse(k+1) - reuse(k)$. Another corollary of demand monotonicity is APF monotonicity, to be discussed in Section 4.

3.7 Comparison with HOTM

This section builds on higher-order theory of memory demand (HOTM) [40]. In particular, it uses the definitions of *apf* (called the peak demand *pd*) and *reuse* from HOTM. However, HOTM does not show how to measure the memory demand. The measurement problem is solved in this section, with new, linear-time offline (Eq. 1 for single k and Eq. 2 for all ks) and online analysis of reuse and new online analysis if live (Eq. 4). The presentation of timescale measures in Section 3.1 is also new. Furthermore, HOTM did not show demand monotonicity, which is important for APF tuning.

The timescale measure shows the average memory demand, it ignores the variation in actual memory demand. The average demand may not be the *actual* demand in any specific window, e.g. those in the illustration in Figure 1. The next section addresses the demand variation, unavoidable when adopting APF in a real memory allocator.

4 APF-based Memory Reservation

APF-based memory reservation is to set the initial reserve size, at each fetch, equal to the memory demand of the next window of *apf* allocations. This is the best reserve size: a greater value will use more memory, and a lesser value will cause a fetch from the central cache before making *apf* allocations. As mentioned earlier, APF is the technique for memory reservation, and *apf* its parameter.

Per-thread, Per-class Control of Reserve Size The *apf* value is set globally for all threads and all size classes. The reservation is customized for every size class of every thread based on the memory demand, which we measure using the online timescale theory (Section 3). When the same *apf* is satisfied for every thread and every size class, it is satisfied for the entire application.

Adaptive Control APF reservation takes in as the parameter the target *apf* and the memory demand $demand(k)$. For adaptive control, it adjusts the dynamic *apf*. In this discussion, we call the overall target and the dynamic target T_{apf} and N_{apf} respectively. Let the current time be n and the number of fetches so far c. Since the global target is to have $c+1$ fetches in $T_{apf} \times (c+1)$ allocations, we set the dynamic target as follows.

$$N_{apf} = T_{apf} \times (c+1) - n \tag{5}$$

If the right-hand side is 0 or negative, we set N_{apf} to T_{apf}.[3]

Figure 4 illustrates the adaptive control. The "timeline" at the bottom is punctuated by two sets of markings. The first are even-space marks of T_{apf} per interval. The second are irregular spaced marks showing when the local reserve becomes empty. The adaptive control sets the initial reserve size at the beginning and whenever the reserve becomes empty. They are computed by Eq. 5 and shown by stars followed by a line segment.

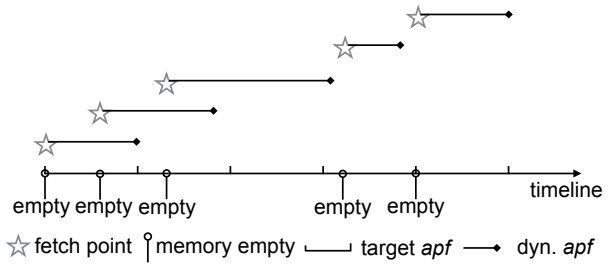


Figure 4. Illustration of adaptive control. The timeline shows two sets of markings: T_{apf} intervals and actual points when the reserve is empty. The adaptive control increases or decreases N_{apf} (shown by the starred lines) to achieve T_{apf}.

The control so far is about memory fetch. We handle the return as follows. At every free operation, we check whether the number of available slots equals $2 \times demand(N_{apf}) + 1$, and if so, we return $demand(N_{apf}) + 1$ slots to the central reserve.

Adaptive vs. Fixed Control The timescale theory measures the average behavior in allocations and frees. We may set a fixed control to always fetch $demand(T_{apf})$. The fixed strategy may miss the target given some adversarial pattern. More commonly, an execution may have phases with radically different memory demands. During each phase, the adaptive control can actually behave like the fixed control but set the reserve size properly for the current phase.

[3] At the beginning of an execution, *demand* is not yet known. Any existing heuristic, e.g. `tcmalloc`, may be used. In a steady state, the initialization does not matter to performance.

APF Monotonicity The property states that *a larger* apf *value means less synchronization and more parallelism.* A larger *apf* means a larger initial reserve size. It follows from Theorem 3.1 that this satisfies the memory demand of more allocations and hence fewer memory fetches.

5 Optimal Memory Reservation

We have developed an offline optimal solution to measure the performance upperbound of any technique. The optimization problem is formulated for a single size class. Given m as the number of transfers, including fetches and returns, allowed in execution, what is the lowest memory inefficiency? Here a fetch may happen at any time, not just at an allocation when a local cache is empty. Let the length of the execution be n. We have $\binom{n}{m}$ ways to place m transfer operations. For each choice, we have 2^m cases since each communication is either a fetch or a return. For each communication, the maximal volume to transfer is n. Hence, a brute force solution needs to evaluate $O(\binom{n}{m} \times 2^m \times n)$ choices. In this section, we give a more efficient solution.

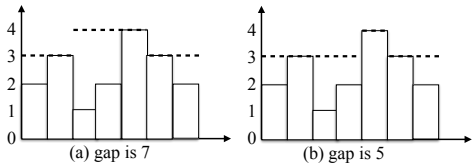


Figure 5. Two placements of 3 roofs over 7 bars. Dashed lines denote roofs. The second placement has less gap.

We convert the optimization problem into the following problem. Suppose we have a bar graph. The height of each bar is the number of live objects. The number of bars is n, which is the total time. Our goal is to use m segments (or rooflines) to cover the n bars to minimize gap between the roofs and the bars. Figure 5 gives an example to illustrate this problem. In Figure 5, there are 7 bars and 3 rooflines. (a) and (b) show two placements. The two gaps are 7 and 5, respectively. Thus the second placement is better.

Now we convert the "roofing" problem into the optimization problem. The first roofline shows that the first communication is a fetch and its height shows the fetch size. The next fetch initiates a new communication. If the next roofline is lower than the previous one, the communication is a return; otherwise, it is a fetch. The height difference is the fetch or return size. Hence, once we have m rooflines over n bars, we know exactly when to communicate and and how much to communicate in which direction. Next we solve the "roofing" problem optimally.

We use dynamic programming [55]. Eq. 6 shows the optimal substructure of the solution:

$$minGap(m, n) = \min_{i=m-1 \ldots n-1} \{minGap(m-1, i) + oneRoofGap(i+1, n)\} \quad (6)$$

wherein $minGap(m, n)$ calculates the minimal gap using m roofs to cover bars from 1 to n, and $oneRoofGap(i, j)$ calculates gap value of using one roof to cover from bar i to bar j. To understand it, consider the last roofline. Its left endpoint ranges from m to n. We select the minimal gap among all results of this range of roofs. For the $oneRoofGap(i, j)$ calculation, it is the height of highest bar between bars i and j. With this optimal substructure, a dynamic programming algorithm follows. We do not describe the algorithm here. The time complexity of the optimization is $O(n^3)$.

6 Evaluation

6.1 Experimental Setup

APF-based tcmalloc We implemented APF based on tcmalloc [16] by replacing all the heuristics based code controlling thread-local cache (see the following) and part of the code of central cache. In each thread for each size class, APF measures the memory demand online as described in Section 3 and sets the reserve size as described in Section 4. The rest of the allocator design stays the same. For example, large objects are allocated directly by mmap or sbrk from OS [16]. The same theory and control may be used by other allocators or a new allocator. Using a base allocator is necessary for us to cleanly compare the new techniques with prior state of the art.

tcmalloc ***and Other Allocators*** Our choice of tcmalloc as the base allocator is motivated by the fact that it is one of the most widely used, it performs well in tests, and it is the default allocator for our largest test program MongoDB.

tcmalloc uses a number of thresholds to control the time and space cost. The most important are *batch size* and *overall cache size*. The *batch size* controls the granularity of fetch, i.e., initial reserve size after a fetch. It is carefully tuned and fixed to 64KB. The *overall cache size* bounds the total reserve size. The *overall cache size* is the only parameter under user control. When a user increases *overall cache size*, program performance usually increases, but the benefit stops at a point when the fixed *batch size* (64KB) becomes the limiting factor. *Unless otherwise noted, the evaluation reports* tcmalloc *after individually tuned for each test and thread count for maximal performance.*

If *overall cache size* is limited, tcmalloc uses various thresholds and heuristics to re-size the reserves. These include memory stealing, garbage collection, scavenging, and "per-list low-water-mark", which is used to "quickly move free chunks from the thread cache to the central free list" when "a thread stops using a particular size."[16].

We also compare with Hoard [5], jemalloc [10] and ptmalloc-v3 [15], which is based on dlmalloc [34] and serves as the default allocator of Linux glibc. Like tcmalloc, other allocators use global parameters to control the size of local reserves. The comparison with them is not a clean comparison, because the performance may differ due to other aspects of the allocator design.

Methodology We measure performance and memory consumption. Memory consumption is the total size of reserves. We use the average, measured by taking the total size of the reserves at each allocation and computing the average.

Our test programs all have the steady-state behavior and run for a long time. The online profiling of $demand(k)$ is done in a burst of 20,000 allocations, and performance is measured in the steady state after profiling.

We measure each test 5 times and take the average. If the deviation from the average is significant, we may use up to 10 runs to take the average. Except for the cost of fetch and return, all experiments are performed on a machine with two Intel Xeon E5-2699 2.30GHz (Haswell) processors. Each processor has 18 cores, and each core supports two hyper-threads. The machine supports 72

hardware threads. The OS is Linux kernel 3.18.7. All allocators (and tests) are compiled by GCC using "-O3 -g". The cost of fetch and return is also measured using a IBM Power 8 machine, with two 10-core processors, with 64GB of RAM. Each core is running at 4.1GHz, with 8 threads per core. The OS is CentOS 7 Linux with kernel version 3.10.0.

6.2 MongoDB Throughput

MongoDB [25] is a popular open-source NoSQL database. It adopts a thread-per-connection architecture. In each connection, a thread allocates memory to store key-value pairs of documents, and the memory allocation manifests various, drastically different demands [17]. *MongoDB* by default uses tcmalloc as its custom memory allocator. The thread-per-connection architecture may lead to very large numbers of threads in certain workloads, in which case per-thread caching could incur excess memory consumption, and the performance of the memory allocator is critical to the overall performance [24].

MongoDB workloads provide multiple configurable input parameters, such as the number of connections, request document size, the number of requests and request type. Our first workload is *iiBench*. We followed Callaghan's test design [23] by inserting 100 million documents to *MongoDB*. We tested both fixed document size, 1024 bytes, and variable size that follows a random distribution. We have also tested a second workload, the *Yahoo! Cloud Serving Benchmark* (*YCSB*) [58], and found similar results, which we omit for lack of space. The two workloads were run with 1 to 128 concurrent connections.

APF Effect Curve Increasing *apf* increases the parallelism of memory allocation. We tune *apf* from 1 to 10,000.[4] As Figure 6(b) shows, greater allocator parallelism causes the allocator to use more memory but MongoDB to run faster. Both thread counts show a sharp increase at the beginning. For 16 threads, it is followed by a slow incline and a plateau. For 32 threads, the increase is muted at first but starts to accelerate towards the end. Hence, 16-thread MongoDB does not benefit from continued increase in allocator parallelism, but 32-thread MongoDB does.

Similar effect is shown in Figure 6(c) when MongoDB serves variable size documents. It has a similar throughput but nearly 10 times the memory consumption. In both thread counts, the throughput drops after reaching the peak when the reserve size exceeds 4000 slots. While the allocator parallelism continues to increase, its benefit in the overall MongoDB throughput is out weighed by the cost of greater memory consumption. The detection is simple since it is a single turning point with monotone changes on both sides.

Multi-objective Tuning We demonstrate tuning for three objectives: *maximal performance*, which is MongoDB throughput; *memory efficiency*, which is the increase over tcmalloc in throughput for similar reserve size; and *memory saving*, which is the reduction over tcmalloc in reserve size for similar performance. In Figure 6(b,c), (untuned) tcmalloc performance is shown as squares. The three types of tuning has geometric explanations. For maximal performance, we find the highest point of the curve. For memory efficiency or saving, we find the point on the curve whose x-axis or y-axis value equals to that of tcmalloc. These are marked by stylized points.

Figure 6(a) tabulates the effect of MongoDB on *iibench* with equal-size documents using six different thread counts between 1 and 128.[5] For maximal performance, APF increases the whole-application throughput by 11% to 55%. It costs more memory to improve at higher thread counts — the increase in reserve size is less than 2 times for 1 and 4 threads, less than 4 for 16 to 64 threads, and 10.6 for 128 threads.

Tuning improves over tcmalloc. With less memory, APF improves MongoDB throughput by at least 8% and for as high as 26%. The memory saving is more dramatic: APF uses just 3% to 10% of the reserve size to achieve a similar throughput.[6] Such large reductions, between 10 times and 33 times, allow *MongoDB* launch many more connections.

6.3 Tail Latency

Tail latency, e.g., 99th+ percentile, is important in interactive Web services, such as search, recommendations, games, and finance. For example, a study on a system called *Few-to-Many (FM)* found that by reducing the 99% tail latency by 26% for Bing, the number of servers can be reduced by 42%, and as a result, *FM* has been deployed on thousands of production servers [21]. *FM* and other systems, *Request Clairvoyant* (*RC*) [28] and *TPC* [27], reduce tail latency by incremental parallelism, i.e., parallelizing long-running requests.

If the long running time is caused by the worst-case cost of memory allocation, these solutions are effective only if the allocator responds well to adding more threads. In parallel allocation, the worst-case cost comes from fetches and returns because they require synchronization. In tcmalloc, the worst-case cost also comes from the running of memory-stealing heuristics. Adding parallelism may actually increase rather than reduce these worst-case costs, because it increases the contention on the allocator. In this section, we evaluate the effect of APF on tail latency.

Olio Web Server We test the *Cloud-Stone Web Serving* benchmark in the CloudSuite [12]. The client generates a workload for a server called *Olio* by simulating 100 users and sending 7 types of requests in parallel for 600 seconds, with a warm-up period of 30 seconds. The client runs in 100 threads on a separate machine (Intel 3.4GHz 8-core machine with 8GB memory and Linux 2.6.32). *Olio* runs on the Intel Haswell machine described earlier, where we also run a database and a geocoder backend. *Olio* is written and runs multi-threaded PHP. The test is the PHP runtime on the server with 4 threads. We replace the default memory allocator of PHP with tcmalloc, APF-based tcmalloc, and three other allocators. Running 100 users does not overload the system, so we measure the performance only by the response time.

Tail Latency of Olio We test the Web server *Olio* for 7 operations: *AddPerson*, *Homepage*, *Login*, *EventDetail*, *AddEvent*,*PersonDetail*, and *EventDetail*. Among them, *AddPerson* and *AddEvent* take the longest time, and their response time depends most on memory allocation. Figure 7 shows *AddPerson* using APF-based tcmalloc

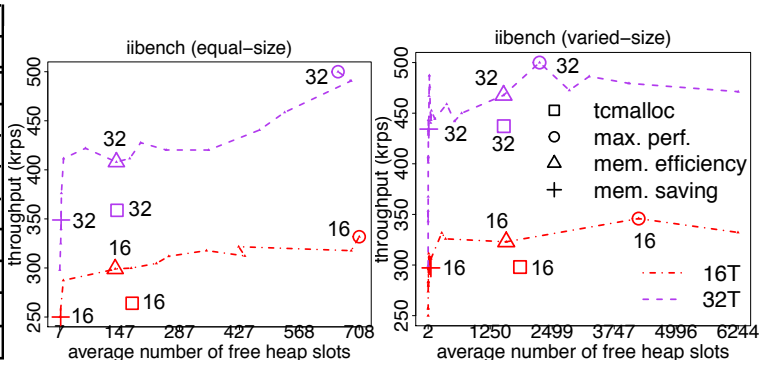
[4] The *apf* values used are 1 2, . . . , 10, 20, . . . , 100, 200, . . . , 1000, 2000, . . . , 10000.

[5] For two thread counts, similar improvements for variable-size documents are shown by stylized markings in Figure 6(c).

[6] Note that tcmalloc is tuned to use as much memory as it needs to improve throughput (Section 6.1).

Tuning objectives and effects		Concurrency					
		1	4	**16**	**32**	**64**	**128**
Maximal perf.	**rel.speed**	1.11	1.12	1.29	1.43	1.55	1.17
	rel.mem.	2.19	2.37	4.02	4.68	4.68	11.64
	apf	5000	9000	6000	6000	8000	7000
Memory efficiency	**rel.speed**	1.09	1.08	1.20	1.15	1.26	1.13
	rel.mem.	0.73	0.91	0.97	0.99	0.97	0.85
	apf	400	50	600	200	50	300
Memory saving	**rel.speed**	1.00	0.98	0.94	0.97	0.97	1.00
	rel.mem.	0.10	0.06	0.05	0.07	0.07	0.03
	apf	5	10	30	20	20	30

(a)



(b) (c)

Figure 6. APF and tcmalloc on *MongoDB* for *iibench* for equal-size documents (a,b) and variable-size documents in (c). The table in (a) shows multi-objective tuning: performance and memory efficiency increase/saving (compared with tuned tcmalloc for similar reserve size/performance).

with *apf*s from 1 to 500, followed by original tcmalloc and three other allocators.

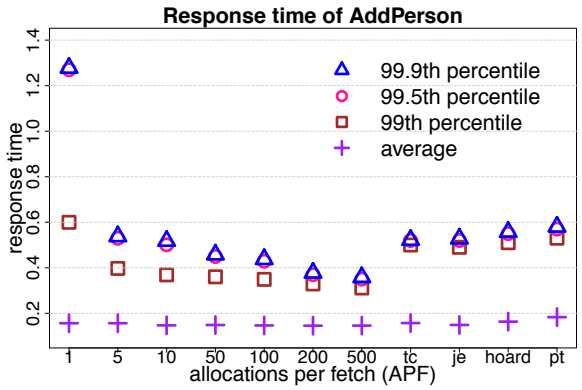


Figure 7. The average and 99th+ percentile response time (seconds) of the *AddPerson* request to *Olio*.

Compared to original tcmalloc, the best tuned APF reduces the 99%, 99.5%, and 99.9% tail latencies by 61%, 49% and 40%. As in other tests, we have tuned tcmalloc by increasing the *overall cache size* from the default 32MB till there is no more performance improvement (2GB).

Memory allocation has relatively mild effects on the average latency, which is between 0.15 and 0.17 seconds in APF (for all *apf*s) and original tcmalloc. Still, the best tuned APF, 0.146 second, is 7% faster than original tcmalloc.

Tail Latency of MongoDB We test the 99% tail latency for the *YCSB* workload on *MongoDB*. *apf* tuning is consistently better than the best tuned tcmalloc. At small thread counts from 1 to 16, the improvement by *apf* is moderate, 5% on average, with average reserve size of just 57% of original tcmalloc. However, at thread counts from 32 to 128, the improvement increases to 51%, indicating that *apf* tuning would be extremely effective in high throughput server applications which routinely run hundreds of thousands of threads simultaneously.

Reducing Worst-case Time by APF APF has four advantages over heuristic-based solutions such as the ones used in original tcmalloc. First, nonuniform reserve size improves memory utilization. Second, APF eliminates the cost and complexity of heuristics. There is no complex heuristic to evaluate. In comparison, tcmalloc uses them occasionally, e.g. for memory stealing, and as a result increases the worst-case time. Third, in highly concurrent memory allocation, the worst-case allocation time is far greater than the average time because of the cost of fetch and return operations. APF directly controls the frequency of these fetch and return operations. Finally, it is necessary to tune without a limit on the reserve size, as it is by APF. In comparison, tcmalloc, due to design considerations, uses the 64KB fixed-reserve design. Therefore, if the tail latency is caused by the worst-case cost of memory allocation, APF is an effective alternative.

6.4 Synthetic Tests

Synthetic benchmarks were widely used to evaluate memory allocators. We have tested six such tests: *MicroServer* and *ProducerConsumer* that are based on *t-test1* and *t-test2* from ptmalloc [15] and *larson*, *threadtest*, *shbench* from Hoard [5] and *SuperServer* from SuperMalloc [31]. For lack of space, we focus on three: *MicroServer*, whose threads do not share objects; *ProducerConsumer*, whose threads do; and *SuperServer*, whose objects have diverse sizes. All three simulate multi-threaded server applications. We use the default input. Table 2 profiles their memory allocation behavior.

The synthetic tests are useful to estimate the cost of fetch and return in a parallel memory allocator. Table 3 shows the time costs of fetch and return operations compared to allocation and free operations in original tcmalloc. The first three data columns show the ratios in total times of these operations, and the last two columns the ratios of per operation times. The total time of fetches and returns accounts 30% or higher of all allocator time in 5 tests. The per fetch/return cost is highly variable across all 6 programs on the 2 test machines. Because of the magnitude and variation, it is important for an allocator to provide effect control over fetch and return costs. APF is effective for such control.

Table 2. The total size and maximal live size of memory allocation in the 6 test programs running with 1 thread and with 16 threads. The many differences between the programs show that they cover a wide range of memory allocation behavior.

Benchmarks		Total objects	Max objects in use	Total memory in bytes	Max memory bytes in use	Avg. object size(bytes)	Total / max
MicroServer	**1**	5,407,871,972	550	28,957,915,488,6	3,024,016	5,354.77	9,575,979.59
	16	5,033,013,605	592	26,940,580,225,600	3,118,448	5,352.77	8,639,098.75
Producer-Consumer	**1**	241,710,885	807	1,294,237,246,112	4,550,176	5,354.48	284,436.74
	16	241,749,923	826	1,294,524,983,008	4,582,656	5,354.81	282,483.56
SuperServer	**1**	6,000,000	600,000	192,000,000	192,000,000	32	1
	16	96,000,000	80,352,493	3,072,000,000	2,571,284,032	32	1.19
threadtest	**1**	5,000,000,000	500,000	320,000,000,000	32,000,000	64	10,000
	16	5,000,000,000	422,352	320,000,000,000	31,095,712	64	10,290.81
shbench	**1**	3,231,100,000	153,960,005	183,756,800,000	8,813,425,904	56.87	20.85
	16	3,231,100,016	153,885,012	183,757,717,504	8,717,986,792	56.87	21.08
larson	**1**	145,112,458	1,033	1,279,995,330,856	9,764,080	8,820.71	131,092.26
	16	48,012,675	16,035	423,542,580,744	143,844,592	8,821.47	2,944.45

Table 3. The time of fetch/return operations compared to allocation/free operations. The first three data columns show the ratios in total times of these operations, and the last two columns the ratios of per operation times. Because of the magnitude and variation of costs, it is important for an allocator to limit the frequency of fetch/return in parallel memory allocation.

Benchmarks		$\frac{\text{fetch}}{\text{alloc.+fetch}}$	$\frac{\text{return}}{\text{free+return}}$	$\frac{\text{fetch+return}}{\text{alloc.+fetch+free+return}}$	$\frac{\text{avg.fetch}}{\text{avg.alloc.}}$	$\frac{\text{avg.return}}{\text{avg.free}}$
MicroServer	**Haswell**	0.26%	0.09%	0.18%	182.34	186.65
	Power8	0.16%	0.04%	0.10%	114.29	81.63
ProducerConsumer	**Haswell**	30.77%	36.82%	33.83%	53.03	240.08
	Power8	19.35%	27.80%	23.62%	27.36	152.79
SuperServer	**Haswell**	79.75%	83.31%	81.55%	821.09	2841.39
	Power8	70.56%	76.40%	73.29%	503.46	1831.74
threadtest	**Haswell**	4.87%	5.23%	5.03%	5264.82	7810.76
	Power8	1.72%	1.74%	1.73%	1809.99	2483.60
shbench	**Haswell**	13.74%	16.44%	15.05%	88.59	141.34
	Power8	9.06%	14.48%	11.76%	55.27	120.74
larson	**Haswell**	47.16%	57.71%	52.83%	71.63	342.83
	Power8	24.91%	36.67%	31.07%	31.78	173.86

Table 4. Effect of APF on *MicroServer*, *ProducerConsumer* and *SuperServer* on performance and memory consumption.

Tuning objectives and effects		MicroServer				ProducerConsumer				SuperServer			
		16	**32**	**64**	**128**	**16**	**32**	**64**	**128**	**16**	**32**	**64**	**128**
Maximal performance	**rel. speed**	1.06	1.13	84.53	154.53	2.49	7.97	18.96	29.69	1.37	1.17	1.46	1.46
	rel. mem.	1.21	2.93	2.40	2.49	1.56	2.17	8.09	10.73	0.89	0.94	0.76	0.93
	apf	200	900	300	300	700	400	800	600	500	500	400	500
Memory efficiency	**rel. speed**	1.06	0.94	5.67	24.35	2.29	5.90	12.28	12.37	1.37	1.17	1.38	1.46
	rel. mem.	1.21	0.94	0.97	0.91	0.90	0.95	0.96	1.00	0.89	0.94	0.95	0.93
	apf	200	80	6	5	300	100	30	30	500	500	300	500
Memory saving	**rel. speed**	1.00	1.01	5.11	9.22	1.29	3.03	5.79	7.92	0.96	0.99	1.16	1.19
	rel. mem.	0.51	0.50	0.24	0.24	0.10	0.19	0.44	0.70	0.35	0.37	0.38	0.37
	apf	50	7	4	1	5	5	5	5	200	200	200	200

6.5 Effect of APF Tuning on Synthetic Tests

The performance of synthetic tests is measured by the throughput which is the number of allocations per second. We compare the performance of tcmalloc with and without APF for three tuning objectives[7] for four thread counts: 16, 32, 64, and 128. The total number of tests for six programs is 24. To save space, we show 12 tests for three benchmarks.

Tables 4 shows that when tuned for maximal performance (by using more memory), the throughput is increased between 6% and 154 times. The increase is over 10 times in 9 out of 24 tests. The corresponding memory increase is smaller, over 10 times in 5 out of 24 tests. The most dramatic improvements happen at high thread counts, i.e., 64 and 128 threads. Over half of the tests (7 out of 12) are improved by 10 times or more, and over half of those increases

[7] For the objectives of memory efficiency and memory saving, we choose the *apf* values which have the closest memory cost and speed, respectively.

are over 50. The minimal improvement is 64% in *SuperServer* and the greatest are 154 times in *MicroServer* and 111 times in *larson*.

When APF uses a similar amount of memory as original tcmalloc, it improves performance in 19 of the 24 tests. The improvement ranges from 17% in *SuperServer* at 32 threads to 12.4 times in *ProducerConsumer* at 128 threads. The most improvements happen at 128 threads. Half of the programs are improved by 9 times or more: *shbench* by 9.4 times, *ProducerConsumer* by 11.4 times and *MicroServer* by 23 times.

Except for one program *threadtest* and one case (*larson* at 16 threads), APF can be tuned to have similar performance as original tcmalloc but save memory significantly. The minimal reduction is 30% at 128 threads in *ProducerConsumer*. The next smallest reduction is 49% at 16 threads in *MicroServer*. The remaining 19 memory savings are between 50% and 81%.

The *apf* values for the 12 tests are shown in Table 4. For different programs and tuning purposes, the actual *apf* varies greatly. The performance is sensitive to the most significant digit of the *apf*. The tuning effort is proportional to the logarithm of the *apf* range.

Comparison with* tcmalloc *Tuning In 13 out of 24 tests, tcmalloc can be improved by increasing *overall cache size* from 32MB to 2GB. Figure 8 shows the improvement of tuned tcmalloc and APF over default tcmalloc. Results show that APF improves tcmalloc by 22% on average, always greater than the improvement from tcmalloc tuning.

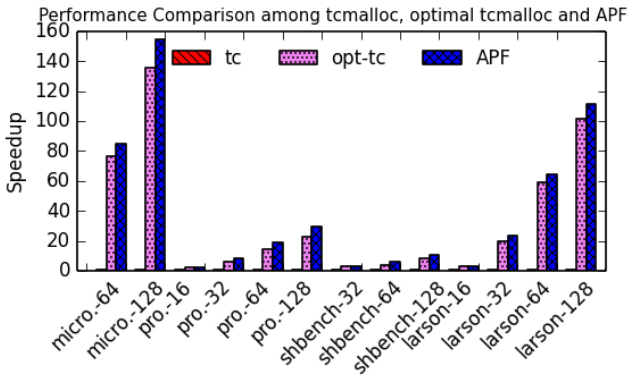


Figure 8. Original, tuned, and APF-based tcmalloc

Comparison with Other Allocators and Optimal Table 5 compares the performance between APF and other allocators on *iibench* on *MongoDB*. jemalloc achieves similar throughput as tcmalloc with 16 and 32 threads. Hoard has similar performance as tcmalloc with 16 threads, but has 25% less throughput than tcmalloc with 32 threads. Synthetic tests show the same ordering of performance among the three allocators (which we omit for lack of space).

Figure 9 compares APF with the optimal solution. In all cases, APF has the same trend as optimal. It is very close to the optimal solution and near identical in two programs.

Table 6 compares APF-based tcmalloc with three other memory allocators, ptmalloc, Hoard, and jemalloc. Compared to original tcmalloc, APF has 4.19× speedup as opposed to 0.26× of ptmalloc, 1.22× of Hoard, and 2.57× of jemalloc. APF is consistently

Table 5. The throughput speedup of 4 allocators, normalized to original tcmalloc on *iibench* running with 16 and 32 threads.

Workloads	APF	Hoard	ptmalloc	jemlloc
iibench (equal,16)	1.29×	0.99×	0.52×	1.01×
iibench (varied,16)	1.25×	0.97×	0.74×	1.02×
iibench (equal,32)	1.43×	0.72×	0.55×	0.94×
iibench (varied,32)	1.19×	0.76×	0.59×	1.01×

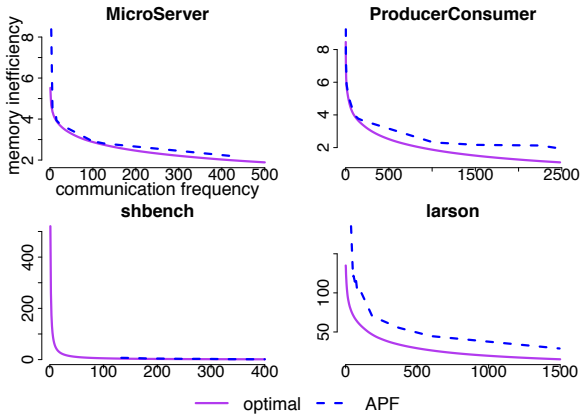


Figure 9. APF versus optimal. Due to the cubic time complexity of optimization, the comparison is shown for a single size class. The x-axis, *communication frequency*, counts both fetches and returns of APF.

Table 6. The throughput speedup of 4 allocators, all running with 64 threads. Three of ptmalloc's tests took too long time, whose speedups are marked 0×.

Programs	APF	Hoard	ptmalloc	jemlloc
MicroServer	5.11×	1.40×	0.66×	3.14×
ProducerConsumer	12.28×	1.92×	0.90×	8.04×
SuperServer	1.25×	0.42×	0.00×	1.07×
threadtest	2.21×	1.46×	0.02×	0.88×
shbench	2.89×	0.54×	0.00×	0.74×
larson	1.42×	1.59×	0.00×	1.33×
average	4.19×	1.22×	0.26×	2.57×

good. Hoard behaves better in four programs, but worse in the other two, compared to tcmalloc. The comparison is unfair in that the other allocators are not tuned for these tests. However, for this paper, the comparison is valid, because our goal is to show the advantage of tuning. In all allocators, a default configuration does not work well for all programs. In high concurrency, it has pathologically bad performance for all programs. For applications that spend much time inside memory allocators, e.g., *ProducerConsumer*, tuning is extremely beneficial.

6.6 Summary of Experimental Findings

First, to fully control allocator parallelism and memory efficiency, it is necessary to individually control per-thread, per-size-class reserve size, and this control should be tailored for each workload. The precise and complete control can significantly improve the

throughput and reduce the tail latency especially for programs with many parallel threads.

Second, online timescale theory is efficient and effective. Its algorithms take a constant number of operations per allocation. It achieves near optimal control when compared with the optimal offline algorithm.

Third, APF tuning is not overly onerous. APF monotonicity ensures that the allocator parallelism increases with *apf*. Manual tuning can cover the full APF effects by trying a handful of *apf* values. In "tuned" results reported in this section, the *apf* values are all round numbers (only the first one or two digits are not zero). The tuning effort is proportional to the logarithm of the *apf* range.

Finally, APF tuning is flexible and effective. It enables a user to select the desirable tradeoff, for example, maximal throughput, efficient memory use, or minimal memory. It can be used for legacy applications, replacing the original allocator with one that has APF tuning.

The alternative to tuning is automatic optimization, which is unlikely effective for three reasons. First, the allocator is only one component of the host application. APF monotonicity does not mean monotone improvement. Second, APF effect curves are never linear and all have different shapes, even for the same application and the same input. Third, a user may have different objectives in the trade-off between performance and memory consumption.

In comparison, tcmalloc and other allocators use the same global parameters and thresholds, e.g. the batch size, for all threads and size classes. They lack the granularity and precision of APF. In addition, we found that tcmalloc spends significant time in running its heuristics, especially at high thread counts.

7 Related Work

Scalable allocators such as Hoard and the subsequent solutions [5, 10, 14, 16, 49, 51] adopt a layered organization with thread-local caches and global heaps. A most recent allocator called scalloc [1] uses a flat cache hierarchy.

Thread-local caches and their size classes cannot reuse each other's memory directly. To ameliorate, tcmalloc [16] and Hoard [5] both fetch or return a fixed number of objects on-demand. The difference is that tcmalloc uses many collaborated heuristics to tune when to communicate, while Hoard delegates the decision to users. jemalloc [10] periodically adjusts the number of memory chunks in a fetch and return at run-time. SuperMalloc [31] prefetches a fixed amount of objects on-demand. Instead of global policies, APF uses per-thread, per-class reserve control enabled by the online timescale theory.

tcmalloc was known for a frequent "slowpoke" in garbage collection *Scavenge()*. Lee et al. found the reason [35]. When there is lack of memory, memory stealing may trigger a chain reaction where a victim thread has to steal the memory back and creates a time-consuming loop. In contrast, the monitoring and control in APF have the same cost across all thread counts, which improves both throughput and tail latency.

Previous allocators use multiple heuristics and setting multiple thresholds for all threads. The interaction makes it difficult for a user to understand and tune performance. Using the new theory, APF lets a user tune the size of all reserves using a single parameter.

In addition to increasing speed, it is beneficial to reduce the variability of heap operations. *TintMalloc* controls heap data placement to balance the speed of thread execution, which reduces their total idle time waiting for barriers [50]. As we have shown, allocation cost may be uneven especially with current allocators in high concurrency. APF equalizes the synchronization cost across threads and can reduce the cost as much as a user wants.

A related problem is predicability. For interactive services, the first goal is to reduce the number of requests exceeding their target latency. An effective solution is a recent work-stealing policy called *tail-control* [36]. It assumes a constant work distribution, which for online workloads of this paper, means that allocator cost must not change with system load. APF makes this guarantee for the synchronization cost.

APF builds on the HOTM and liveness theories [8, 38–40]. The liveness theory does not model memory reuse, so it cannot measure memory demand. HOTM defines the average reuse but gives no solution for either offline or online measurement and no remedy if the actual memory demand deviates from the average (Section 3.7). There are many similar works using timescale functions [9, 37, 41, 43–48].

There are other considerations concerning time and space, such as size class partition [10, 31], locality [11, 29, 33, 51, 56], and metadata management [5, 10, 11, 16, 51]. The existing techniques are efficient and mature, thus we respect the their designs. There are also efforts exploiting object demographics, such as lifetimes, reference and object sizes [4, 13, 18, 20, 53], to improve the performance of memory management. In a different vein, application specific optimizations of memory allocators are studied for Web applications [26, 32], transactional workloads [3], and many-core applications [52].

8 Summary

We have designed a new technique APF memory reservation for parallel memory allocation. APF precisely controls the per-thread, per-class reserve size using a single parameter. We have presented a new theory of timescale memory demand, linear-time offline and online algorithms to measure two timescale functions, liveness and reuse, an online algorithm to achieve the target *apf*, and an offline algorithm to give the upper-bound performance. We have evaluated APF on *MongoDB*, *Olio* Web server and synthetic benchmarks. Results demonstrate that APF is highly effective in improving the throughput and reducing the memory waste as well as the worst-case time cost of parallel memory allocation.

Acknowledgments

The authors wish to acknowledge the collaboration with Lingxiang Xiang on optimal memory reservation and thank ISMM reviewers for their careful evaluation and helpful suggestions and for Tongping Liu for shepherding. The presentation has also been improved thanks to the comments by Jacob Brock, Wentao Cai, Dong Chen, John Criswell, Colin Pronovost, Zhizhou Zhang, and Peng Zhao. The research is supported in part by the National Science Foundation (Contract No. CCF-1717877, CCF-1629376, CNS-1319617, CCF-1116104) and an IBM CAS Faculty Fellowship.

References

[1] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, multicore-scalable, low-fragmentation memory allocation through large

virtual memory and global data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*.

[2] Matthew Arnold and Barbara G. Ryder. 2001. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, Utah, 168–179.

[3] Alexandro Baldassin, Edson Borin, and Guido Araujo. 2015. Performance Implications of Dynamic Memory Allocators on Transactional Memory Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[4] David A. Barrett and Benjamin G. Zorn. 1993. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*.

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 117–128. https://doi.org/10.1145/356989.357000

[6] T. M. Chilimbi and M. Hirzel. 2002. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 199–209.

[7] David Detlefs, Al Dosser, and Benjamin Zorn. 1994. Memory Allocation Costs in Large C and C++ Programs. *Softw. Pract. Exper.* (1994).

[8] Chen Ding and Pengcheng Li. 2014. Cache-Conscious Memory Management. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*.

[9] Chen Ding and Pengcheng Li. 2016. Timescale Stream Statisitcs for Hierarchical Management. In *STREAM workshop*.

[10] Jason Evans. 2005. jemalloc: a memory allocator. (2005). http://www.canonware.com/jemalloc/.

[11] Yi Feng and Emery D. Berger. 2005. A Locality-improving Dynamic Memory Allocator. In *Proceedings of the 2005 Workshop on Memory System Performance*.

[12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[13] David Gay and Alex Aiken. 1998. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*.

[14] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. 2005. Allocating memory in a lock-free manner. *Algorithms - ESA* (2005), 329–342.

[15] Wolfram Gloger. 2006. ptmalloc: a memory allocator. (2006). http://www.malloc.de/en/.

[16] Google. 2004. Tcmalloc: a memory allocator. (2004). http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[17] Google groups: MongoDB discussion [n. d.]. ([n. d.]). https://groups.google.com/forum/?fromgroups#!search/tcmal-loc/mongodb-dev/8o_B58q0T3s/iM4IHM2rGgAJ.

[18] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*.

[19] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*.

[20] D. R. Hanson. 1990. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Softw. Pract. Exper.* (1990).

[21] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[22] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 156–164. https://doi.org/10.1145/1024393.1024412

[23] iibench. 2015. A case study of the iiBench benchmark. (2015). http://smalldatum.blogspot.com/2015/06/insert-benchmark-for-mongodb-memory.html.

[24] iibench. 2015. A MongoDB performance issue. (2015). https://jira.mongodb.org/browse/SERVER-20306.

[25] MongoDB Inc. 2009. The MongoDB database. (2009). https://www.mongodb.org/.

[26] Hiroshi Inoue, Hideaki Komatsu, and Toshio Nakatani. 2009. A Study of Memory Management for Web-based Applications on Multicore Processors. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[27] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. 2016. TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 129–141. https://doi.org/10.1145/2872362.2872370

[28] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2014. Predictive parallelization: taming tail latencies in web search. In *The International ACM SIGIR Conference*. 253–262. https://doi.org/10.1145/2600428.2609572

[29] Alin Jula and Lawrence Rauchwerger. 2009. Two Memory Allocators That Use Hints to Improve Locality. In *Proceedings of the 2009 International Symposium on Memory Management*.

[30] Murali R. Krishnan. 1999. Heap: Pleasures and pains. *Microsoft Developer Newsletter* (1999).

[31] Bradley C. Kuszmaul. 2015. SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*.

[32] PerAAke Larson and Murali Krishnan. 1998. Memory Allocation for Long-running Server Applications. In *Proceedings of the 1st International Symposium on Memory Management*.

[33] Chris Lattner and Vikram S. Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 129–142.

[34] Doug Lea. 1987. A memory allocator. (1987). http://gee.cs.oswego.edu/dl/html/malloc.html.

[35] Sangho Lee, Teresa Johnson, and Easwaran Raman. 2014. Feedback Directed Optimization of TCMalloc. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*.

[36] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. 2016. Work stealing for interactive services to meet target latency. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 14:1–14:13. https://doi.org/10.1145/2851141.2851151

[37] Pengcheng Li, Dhruva R. Chakrabarti, Chen Ding, and Liang Yuan. 2017. Adaptive Software Caching for Efficient NVRAM Data Persistence. In *Proceedings of the International Parallel and Distributed Processing Symposium*. 112–122. https://doi.org/10.1109/IPDPS.2017.83

[38] Pengcheng Li and Chen Ding. 2013. All-window Data Liveness. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.

[39] Pengcheng Li, Chen Ding, and Hao Luo. 2014. Modeling Heap Data Growth Using Average Liveness. In *Proceedings of the International Symposium on Memory Management*.

[40] Pengcheng Li, Hao Luo, and Chen Ding. 2016. Rethinking a heap hierarchy as a cache hierarchy: a higher-order theory of memory demand (HOTM). In *Proceedings of the International Symposium on Memory Management*. 111–121. https://doi.org/10.1145/2926697.2926708

[41] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. 2019. Beating OPT with Statistical Clairvoyance and Variable Size Caching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[42] Lockless Inc. 2012. A lockless memory allocator. (2012). http://locklessinc.com/.

[43] Hao Luo, Jacob Brock, Chencheng Ye, Pengcheng Li, and Chen Ding. 2016. Compositional model of Coherence and NUMA Effects for Optimizing Thread and Data Placement. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*.

[44] Hao Luo, Guoyang Chen, Pengcheng Li, Chen Ding, and Xipeng Shen. 2016. Data-centric Combinatorial Optimization of Parallel Code. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. poster paper*.

[45] Hao Luo, Guoyang Chen, Fangzhou Liu, Pengcheng Li, Chen Ding, and Xipeng Shen. 2018. Footprint Modeling of Cache Associativity and Granularity. In *Proceedings of the International Symposium on Memory Systems*. ACM, New York, NY, USA, 232–242. https://doi.org/10.1145/3240302.3240419

[46] Hao Luo, Chen Ding, and Pengcheng Li. 2014. Optimal thread-to-core mapping for pipeline programs. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*.

[47] Hao Luo, Pengcheng Li, and Chen Ding. 2015. *Parallel Data Sharing in Cache: Theory, Measurement and Analysis*. Technical Report URCS #994. Department of Computer Science, University of Rochester.

[48] Hao Luo, Pengcheng Li, and Chen Ding. 2017. Thread Data Sharing in Cache: Theory and Measurement. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 103–115. http://dl.acm.org/citation.cfm?id=3018759

[49] Maged M. Michael. 2004. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA, 35–46.

[50] Xing Pan, Yasaswini Jyothi Gownivaripalli, and Frank Mueller. 2016. TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring. In *Proceedings of the International Parallel and Distributed Processing Symposium*. 363–372. https://doi.org/10.1109/IPDPS.2016.26

[51] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the International Symposium on Memory Management*. 84–94.

[52] Sangmin Seo, Junghyun Kim, and Jaejin Lee. 2011. SFMalloc: A Lock-Free and Mostly Synchronization-Free Dynamic Memory Allocator for Manycores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*.

[53] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. 2002. Exploiting Prolific Types for Memory Management and Optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[54] Devesh Tiwari, Sanghoon Lee, James Tuck, and Yan Solihin. 2010. MMT: Exploiting fine-grained parallelism in dynamic memory management. In *Proceedings of the International Parallel and Distributed Processing Symposium*.

[55] David B. Wagner. 1995. An introductory article on dynamic programming in Mathematica. (1995).

[56] Zhenjiang Wang, Chenggang Wu, and Pen-Chung Yew. 2010. On Improving Heap Memory Layout by Dynamic Pool Allocation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*.

[57] Rickey C. Weisner. 2012. How memory allocation affects performance in multithreaded programs. *System News for Sun Users* (2012).

[58] YCSB 2015. The Yahoo! Cloud Serving Benchmark. (2015). https://github.com/brianfrankcooper/YCSB/wiki.