ACDC: Towards a Universal Mutator for Benchmarking Heap Management Systems

Martin Aigner Christoph M. Kirsch
University of Salzburg
firstname.lastname@cs.uni-salzburg.at

Abstract

We present ACDC, an open-source benchmark that may be configured to emulate explicit single- and multi-threaded memory allocation, sharing, access, and deallocation behavior to expose virtually any relevant allocator performance differences. ACDC mimics periodic memory allocation and deallocation (AC) as well as persistent memory (DC). Memory may be allocated thread-locally and shared among multiple threads to study multicore scalability and even false sharing. Memory may be deallocated by threads other than the allocating threads to study blowup memory fragmentation. Memory may be accessed and deallocated sequentially in allocation order or in tree-like traversals to expose allocator deficiencies in exploiting spatial locality. We demonstrate ACDC's capabilities with seven state-of-the-art allocators for C/C++ in an empirical study which also reveals interesting performance differences between the allocators.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Memory management

General Terms Performance, Measurement

Keywords benchmark; explicit heap management; multicore

1. Introduction

ACDC is an open-source benchmark that may be configured to emulate virtually any single- and multi-threaded mutator behavior for measuring allocation, deallocation, and memory access throughput as well as memory consumption and multicore scalability of an allocator. ACDC itself is designed and implemented to introduce negligible temporal and bounded spatial overhead and to scale to large numbers of threads on multicore hardware. In particular, ACDC implements all per-object operations in constant time, pre-allocates all memory for bookkeeping during initialization, and minimizes contention on shared memory for bookkeeping by bulk processing shared objects.

ACDC emulates the lifecycle of dynamically allocated objects which, as shown in Figure 1, begins with the allocation of memory for storing an object on the heap, followed by read and write accesses to the allocated memory, and ends with the deallocation of the allocated memory. The time from allocation to deallocation is called the lifetime of an object. The time from allocation to last access is called the liveness of an object which ACDC, unlike other benchmarking tools, also emulates explicitly by controlling object access. The difference between lifetime and liveness of an object, here called deallocation delay, emulates mutator inefficiencies in identifying dead objects for deallocation which may in turn expose allocator inefficiencies in handling dead memory.

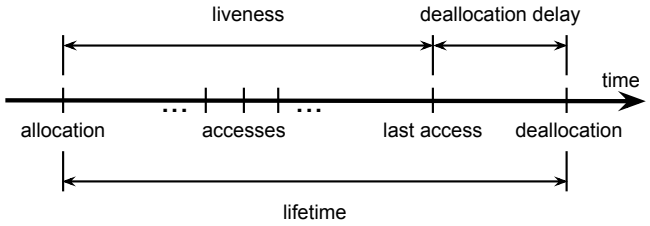


Figure 1: The lifecycle of an object

ACDC allocates objects of different size and may do so periodically at different configurable frequencies for temporary use with finite lifetime (AC) as well as for permanent use with infinite lifetime (DC), hence the name. Size, lifetime, and number of objects are determined according to configurable random distributions that mimic typical behavior observed with allocation-intensive C programs where smaller and short-living objects are more likely to occur than larger and long-living objects [3, 21].

Time in ACDC is logical. Time advances when a configurable amount of memory, called the ACDC time quantum, has been allocated [11]. The allocation and deallocation frequencies are derived from the time quantum. Objects are allocated at the rate of the time quantum and deallocated at the rate of their lifetimes which are multiples of the time quantum.

We present experimental evidence that ACDC is able to reveal the relevant performance characteristics of seven state-of-the-art allocators for C/C++. Our experiments are thus firstly about the capabilities of ACDC and only secondly about the allocators although seeing their relative performance turns out to be interesting and valuable, in particular the time-space trade-offs of scalability versus memory consumption as well as spatial locality versus false sharing. The few unexplained anomalies are to the best of our knowledge artifacts caused by the allocators, not ACDC.

The structure of the paper is as follows. ACDC is described in detail in Section 2. Related work is discussed in Section 3. The allocators and experiments are described in Section 4. Conclusions are in Section 5.

ISMM'13, June 20–21, 2013, Seattle, Washington, USA.

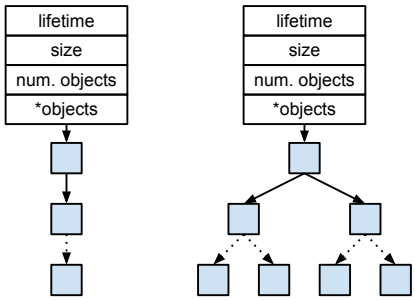


Figure 2: A lifetime-size-class is implemented by either a linked list or a binary tree

2. ACDC

ACDC allocates sets of objects with the same size and lifetime and gathers the objects in so-called lifetime-size-classes. In particular, each ACDC thread first determines the number of objects that are to be allocated next based on size and lifetime values obtained by configurable random distributions and then allocates the objects one by one and stores them in a lifetime-size-class. In order to implement the logical time in ACDC each thread maintains a thread-local clock. After a thread allocated the objects for a lifetime-size-class the thread checks if it has allocated the amount of memory given by the ACDC time quantum since the last time advance, independently of the memory other threads have allocated. If yes, the thread advances the clock and proceeds to share, access, and deallocate objects. If not, the thread allocates objects for another lifetime-size-class until time is advanced. The logical time in ACDC is thus thread-local and approximative since clocks may be late up to the amount of memory allocated for the objects of the largest lifetime-size-class. The drift between clocks is bounded by a configurable amount through a barrier. The lifetime of a shared object ends when it has ended for all threads sharing the object.

As shown in Figure 2, a lifetime-size-class may be configured to be implemented by either a linked list or a binary tree of objects to facilitate subsequent memory access and eventual deallocation either in the exact (list) or mirrored, depth-first (tree) order of allocation. In particular, the tree is constructed in pre-order, left-to-right and subsequently traversed in pre-order, right-to-left. Other choices are possible but remain for future work. Multiple lifetime-size-classes containing objects of different size but all with the same lifetime are gathered in so-called lifetime-classes which are linked lists of such lifetime-size-classes, as shown in Figure 3. Lifetime-classes facilitate constant-time insertion of lifetime-size-classes and deallocation of all objects in a given lifetime-class in time linear in the number of objects. The objects in a lifetime-class

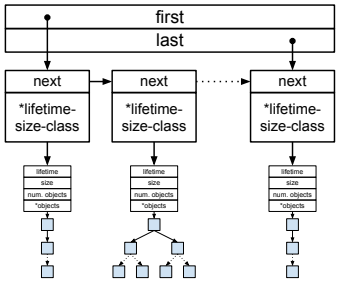


Figure 3: A lifetime-class is a linked list of lifetime-size-classes with the same lifetime

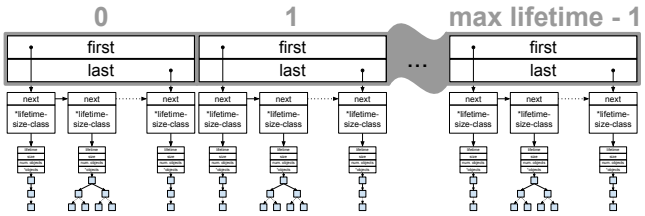


Figure 4: A heap-class is an array of lifetime-classes where the index represents a lifetime

are deallocated when their lifetime ends in which case the lifetime-class is said to have expired.

ACDC distinguishes a configurable amount of lifetime-classes of which one is dedicated to permanent objects. Each ACDC thread maintains its own lifetime-classes stored in an array, called heap-class, which is indexed by lifetime for constant-time access, as shown in Figure 4.

For benchmarking single-threaded allocation and deallocation throughput and memory consumption ACDC may be configured to emulate a single-threaded mutator that allocates and deallocates objects but never actually accesses the allocated memory, as shown in Figure 5a. For benchmarking memory access throughput to expose differences in memory layout quality ACDC may also be configured to read and write allocated memory between allocation and deallocation. In configurations with a single ACDC thread, as shown in Figure 5b, ACDC may thus expose allocator inefficiencies in accommodating spatial locality.

In multi-threaded configurations, for benchmarking multi-threaded allocation and deallocation throughput as well as memory consumption and multicore scalability, as shown in Figure 5c, ACDC may expose allocator inefficiencies in avoiding contention on allocator data structures (through concurrent allocation and deallocation), blowup memory fragmentation [5] (through deallocation of objects allocated by other threads), and false sharing of allocated objects (through thread-local access of unshared objects). All three types of inefficiencies may prevent multicore scalability.

Sharing objects in multi-threaded ACDC works by having each ACDC thread allocate a configurable number of objects for shared rather than thread-local use. In addition to the thread-local heap-classes each thread also maintains a second, lock-protected heap-class. The lock-protected heap-classes of all threads together serve as distribution pool for shared objects, as shown in Figure 6. Each thread (producer) inserts lifetime-size-classes of objects that are

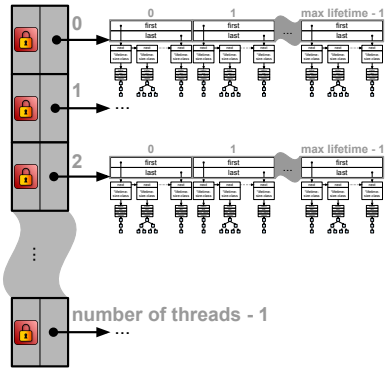
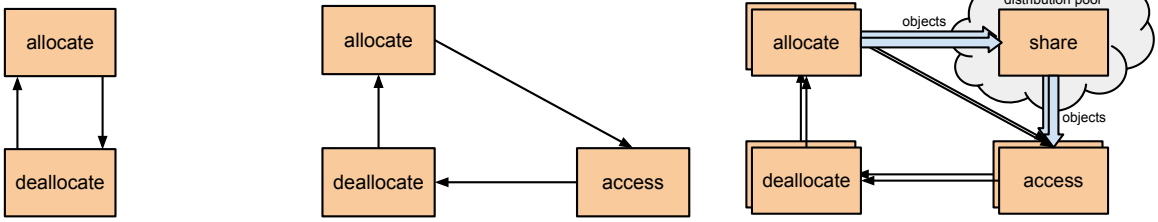


Figure 6: The distribution pool is an array of lock-protected heap-classes, one per thread.



(a) Control-flow view of a single ACDC thread allocating and deallocating objects

(b) A single ACDC thread allocating, accessing, and deallocating objects

(c) Multiple ACDC threads allocating, sharing, accessing, and deallocating objects

Figure 5: ACDC in a nutshell

Parameter	Range	Default
mode	ACDC or false-sharing	ACDC
number of threads	1 to 64	1
number of objects	0 to int-max	0
min. size	8 to int-max	16 B
max. size	min. size to int-max	256 B
min. liveness	1 to benchmark duration	1
max. liveness	min. liveness to int-max	10
deallocation delay	0 to benchmark duration	0
time quantum	1 to int-max	256 KB
benchmark duration	1 to int-max	50
max. time drift	1 to benchmark duration	10
list-based ratio	100% - tree-based ratio	100%
access live objects	TRUE or FALSE	TRUE
write access ratio	0% to 100%	10%
shared objects	TRUE or FALSE	FALSE
shared objects ratio	0% to 100%	100%
retrieving threads ratio	0% to 100%	100%

Table 1: ACDC runtime options and default settings

meant to be shared into the lock-protected heap-classes of (a subset of) all threads and retrieves (consumer) possibly many lifetime-size-classes inserted by others packaged as lifetime-classes from its own lock-protected heap-class. Insertion is linear in the number of threads and retrieval is linear in the number of lifetime-classes. The retrieved lifetime-classes are then inserted into the thread-local heap-class in constant time for further thread-local processing.

2.1 Configuration

ACDC offers extensive configuration of its allocation, sharing, access, and deallocation behavior. Table 1 shows the relevant options along with their default values. We briefly describe each parameter.

The mode parameter switches between ACDC's default mode in which object sizes and lifetimes are determined by random distributions and a mode for exposing allocator inefficiencies due to false sharing. In this mode ACDC only allocates small objects with the same, fixed lifetime.

The number of threads defines how many concurrent threads ACDC creates and runs. The upper bound of 64 threads is explained in Section 2.3. The number of objects defines how many objects ACDC allocates in a lifetime-size-class. With the default of zero ACDC determines the number dynamically based on object size and lifetime. The minimum and maximum size and liveness define the range of the random distributions that determine the size and liveness, respectively, of the objects ACDC allocates. The deallocation delay extends the lifetime of all objects such that ACDC stops accessing the objects when their liveness ended but deallocates them only after their lifetime ended.

The time quantum defines the amount of memory that each thread needs to allocate to advance its thread-local clock by one unit of time. The benchmark duration defines the time to terminate the benchmark by how often each thread needs to advance its clock before termination. The maximum difference between the clock value of a given thread and the value of any other thread's clock is bounded by the maximum time drift.

ACDC gathers allocated objects with the same size and lifetime in lifetime-size-classes that may be implemented by a linked list or a binary tree. The data structure is chosen randomly according to the configured list-based and tree-based ratio.

By default ACDC accesses live objects in between allocation and deallocation as shown in Figure 5b. A behavior where objects are not accessed at all, as shown in Figure 5a, may also be configured by setting access live objects to FALSE. In this case the write access ratio, which controls the fraction of the live heap that will be modified upon access, is ignored. Otherwise, 10% of the live memory are by default written during access.

The behavior illustrated in Figure 5c can be configured by setting shared objects to TRUE. In this case the shared objects ratio defines the number of allocated lifetime-size-classes that will be shared with other threads. The receiving threads ratio controls the number of threads that will receive references to the shared objects through the distribution pool.

2.2 Metrics and Probes

ACDC explores the basic performance dimensions time and space. Time is reported in normalized total allocation time, normalized total deallocation time, and normalized total access time in milliseconds (also consistently in all experiments). Note that unlike the logical time of ACDC, temporal performance is reported in real time. By allocation time we mean the time an allocator spends in allocating objects. Total allocation time is the sum of the allocation time the allocator benchmarked by ACDC spends in all threads. The normalized total allocation time is the total allocation time divided by the number of threads. Total allocation time is thus equal to normalized total allocation time in single-threaded configurations. Allocation time is measured in CPU cycles by reading the CPU time stamp counter before and after allocating a lifetime-size-class of objects. The CPU cycles are scaled to milliseconds during normalization. Normalized total deallocation time and normalized total access time are defined similarly. Here the time stamp counter is read before and after deallocating expired lifetime-classes and accessing objects, respectively. The time spent in sharing lifetime-size-classes, controlling the time drift, and other bookkeeping is thus not considered.

Space is reported in normalized average memory consumption in megabytes (again also consistently in all experiments). By memory consumption we mean the memory consumption of ACDC without the overhead of the bookkeeping data structures. For measuring memory consumption we have implemented an interface to the Linux proc filesystem that provides the resident set size, i.e., the number of virtual memory pages ACDC maintains in real memory. ACDC samples memory consumption at each time advance starting after a warm-up phase of twice the maximum liveness. The average memory consumption is the arithmetic mean of the samples without the bookkeeping overhead. The normalized average memory consumption is the average memory consumption again divided by the number of threads.

For simplicity, we do not allocate permanent objects in our experiments so that the arithmetic mean is a useful metric and the memory required for bookkeeping overhead is bounded in ACDC's configuration parameters and can thus be pre-allocated. Although the required amount may be estimated for each configuration we determined an amount that works for all experiments (500 MB) and then only used that amount. We employ the brk system call for pre-allocating the memory for bookkeeping to avoid performance impacts on the allocators under test. All bookkeeping objects are allocated by ACDC's own memory management in the pre-allocated space and aligned and padded to cache lines to avoid false sharing.

2.3 Implementation details

ACDC constructs lifetime-size-classes by approximating empirical findings that suggest objects allocated by real applications are more likely to be small and short-living than large and long-living [3, 21]. ACDC determines size, lifetime, and number of objects of a lifetime-size-class in constant time in three steps. First, ACDC selects the size from a uniformly distributed, discrete interval $[2^r, 2^{r+1}]$ where r is selected from a uniformly distributed, discrete interval $[log_2(\text{min. size}), log_2(\text{max. size}))$. Next, ACDC randomly selects the liveness from a uniformly distributed, discrete interval [min. liveness, max. liveness] and adds the deallocation delay to obtain the lifetime. In the last step, ACDC calculates the number of objects based on the selected size and liveness with the following formula:

$$\begin{aligned}\text{number of objects} = &(log_2(\text{max. size}) - log_2(\text{selected size}) + 1)^2 * \\ &(\text{max. liveness} - \text{selected liveness} + 1)^2\end{aligned}$$

The formula yields a large number of objects if the distance of the selected size to the maximum size is large (the objects are small) or if the distance of the selected liveness to the maximum liveness is large (the objects are short-living).

ACDC may run up to 64 threads. The upper bound enables lock-free deallocation of shared objects through atomic operations on 64-bit words storing 64-bit bitmaps with one bit for each thread. Each lifetime-size-class maintains such a bitmap where the thread that allocates the lifetime-size-class sets the bits that are assigned to the threads that are selected to share the lifetime-size-class. Each sharing thread resets its assigned bit when the lifetime-class that contains the shared lifetime-size-class expires without deallocating the objects unless the thread determines the word storing the bitmap to be zero. On platforms that support atomic operations on 128-bit words ACDC may run up to 128 concurrent threads.

3. Related Work

3.1 Empirical studies

Empirical studies on memory allocators are typically performed as part of the literature on new allocators. To the best of our knowledge there is no recent academic study of the relative performance of explicit memory allocators. The Oracle Technology Network contains a recent article on the performance impact of memory allocation in multi-threaded applications [20]. However, it only compares three allocators namely Hoard [5], libumem [6] (based on the SunOS slab allocator), and mtmalloc (part of Oracle Solaris 10). The study concludes that in terms of latency and scalability mtmalloc outperforms the other two allocators and in terms of memory efficiency libumen does best. A different benchmark performed by Berger [4] on the same allocators showed speedups of 2.74 and 3.13 comparing Hoard to libumem and mtmalloc, respectively.

3.2 Benchmarks

The literature on memory management systems contains a large and diverse set of benchmarking programs for evaluating the performance of allocators. We focus our discussion on synthetic programs that appeared in well-known allocator papers, e.g. Hoard [5], LKmalloc [13], Streamflow [18], and the allocator by Michael [17].

The Larson benchmark [13] aims at simulating the behavior of a server responding to a client request. A worker thread in the Larson benchmark receives a set of objects from another thread, performs random deallocations and allocations on this set and writes two words in each newly allocated object. Then it passes the set of objects to a new thread performing the same routine and terminates. The benchmark may run multiple workers in parallel. The threads allocate objects of different size which are uniformly distributed in a configured range. This may be an unrealistic assumption according to previous results on object lifetime characteristics [3, 21]. Also, this benchmark does not allow to control the lifetime of the allocated objects and does not implement heap access.

Sh8bench is the latest version of a synthetic benchmark by MicroQuill [2]. It computes a simple object size distribution based on a statically predefined enumeration of only 12 different sizes ranging from 8 to 168524 bytes. In each round, the mutator deallocates a portion of the objects allocated in the previous round and also allocates new objects. The mutator does not involve access to the requested dynamic memory. Like the Larson benchmark, sh8bench offers no control over object lifetimes.

Lever and Boreham presented a performance study of ptmalloc [15] using three benchmarking programs as part of the Linux scalability project. The first program, called malloc-test, runs multiple threads that all allocate a fixed number of objects of fixed size and deallocate each object right after allocation. The goal is to examine malloc and free latency for an increasing number of threads. The second program is a simplified version of the Larson benchmark using only a fixed object size. The goal of this benchmark is to expose blowup memory fragmentation [5], i.e., increasing memory consumption when deallocating threads are different from allocating threads. Finally, the third program tests for false sharing effects where a set of objects is allocated and each object is accessed by a different thread. However, only malloc-test is available for download on the Linux scalability project webpage [1].

These commonly used benchmarking programs achieve similar goals in different ways but, unlike ACDC, do not allow to evaluate all relevant allocator performance criteria in isolation. The Larson benchmark only provides a throughput metric counting the number of allocations and deallocations. The malloc-test and Sh8bench benchmarks accumulate all performance information in total execution time. Effects related to spatial locality cannot be explicitly studied with any of these tools. Moreover, the liveness of objects is not modeled explicitly. In contrast, ACDC is an attempt to enable emulation of mutator behavior that reveals all relevant allocator performance characteristics.

For garbage-collected programming languages like Java there exist standardized evaluation suites like SPECjvm2008 [19]. However, ACDC currently does not support benchmarking implicit memory management systems.

4. Experiments

All experiments ran on a server machine with four 6-core 2.1 GHz AMD Opteron 8425 processors, 64 KB L1 and 512 KB L2 data cache per core, 6 MB shared cache per processor, 110 GB of main memory, and Linux kernel version 3.2.0.

The allocators were compiled using their default Makefile (all with compiler optimizations enabled) except ptmalloc2 and tcmalloc which came pre-compiled with Ubuntu LTS 12.04. We obtained ptmalloc2 from the GNU C library version 2.15 and tcmalloc from the Google perftools package version 1.7.

Unless stated otherwise, we use the metrics defined in Section 2.2 and repeated each experiment five times. The graphs show the arithmetic mean and the sample standard deviation. The benchmark duration is set to multiples of maximum liveness where we observed that the relative performance differences between allocators do not change anymore by extending the benchmark duration.

4.1 Allocators

We employ seven state-of-the-art multi-threaded allocators for C/C++ that worked for us out of the box without any modifications. In the following we briefly discuss the key features of these allocators.

The ptmalloc2 allocator by Wolfram Gloger is based on Doug Lea's allocator [14] and shipped as part of the GNU C library (glibc) in most Linux distributions. Objects allocated through ptmalloc2 are 16-byte-aligned and have an 8-byte header. Requests for objects smaller than 64 bytes are served from so-called fast bins, i.e., caching pools of recently freed objects. Objects larger than 512 bytes are managed in a best-fit fashion. The ptmalloc3 allocator is the latest version of Wolfram Gloger's allocator implementation [9]. It uses a POSIX mutex for all public calls to the allocator. The algorithms are also based on Doug Lea's allocator.

The jemalloc allocator [7, 8] written by Jason Evans aims at multicore scalability. It is the default allocator in FreeBSD, NetBSD, and some versions of Mozilla Firefox. The allocator divides the heap into independent sub-heaps called arenas that can be processed in parallel. In addition, each thread maintains a cache that can be accessed without locking. Freed objects are always returned to the arena they were allocated from to control blowup memory fragmentation [5].

A similar approach is taken by the tcmalloc allocator [10] from Google. The allocator serves requests for small objects from thread-local caches. When a request cannot be served, a bunch of objects is fetched from the central heap. Large objects are directly served by the central heap. In contrast to jemalloc, small freed objects are not put in the thread cache of the allocating thread but in the cache of the deallocating thread. When a thread cache exceeds a size of 2MB a garbage collector moves unused objects to the central heap, again to control blowup memory fragmentation.

Hoard by Berger et al. [5] was the first allocator designed for multicore scalability using per-CPU heaps that addressed the blowup fragmentation problem. Objects are allocated in size classes which are organized in so-called superblocks, i.e., contiguous memory allocated from the operating system in multiples of the system page size. Freed objects are returned to the per-CPU heap from which they were allocated. Superblocks that become less utilized than a given empty fraction may be moved to a shared central heap where the available memory can be re-used by another per-CPU heap thus balancing free memory among threads and limiting blowup fragmentation.

The tbb Scalable Allocator [12] by Intel uses thread-local caches and a global free list when a request cannot be served from the object caches. The global free list is protected by fine-grained locks. As with jemalloc, freed objects are returned to the heap they were allocated from. To reduce synchronization, tbb uses two separate free lists for objects returned by the owner and other threads.

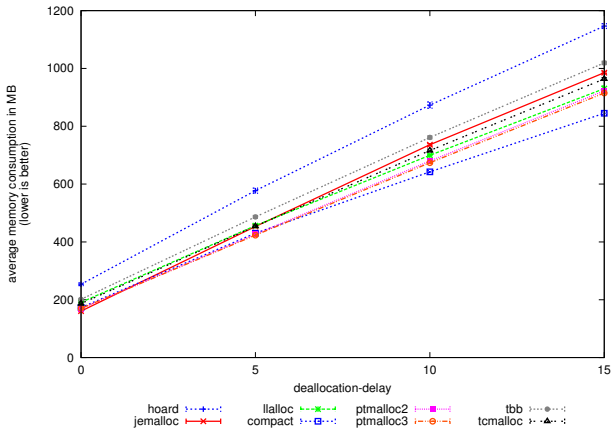


Figure 7: Average memory consumption for an increasing deallocation delay

The Lockless Memory Allocator (llalloc) [16] also uses different algorithms for different size classes. Small objects up to 512 bytes are managed by a slab allocator that uses slabs for each size class which are dynamically allocated for each thread. Empty slabs are traded between threads using per-CPU locking. Objects larger than 512 bytes are served by a best-fit allocation strategy which is extended by a per-size object cache that serves objects in LIFO order. Synchronization in llalloc is performed by lock-free queues, one queue per thread. A thread that frees an object allocated by another thread places the object in that thread's queue. Eventually, the allocating thread will empty its queue and reuse the object thereby controlling blowup memory fragmentation.

ACDC itself provides two mutator-aware allocators as baseline, in particular for benchmarking spatial locality and false-sharing performance. The first allocator, called compact, reserves a contiguous area of memory and arranges it as array of objects to store the objects of a lifetime-size-class without any space in between the objects. In list-based lifetime-size-classes the first object is stored at index 0 and the successor of an object stored at index i is located at index $i + 1$. In tree-based lifetime-size-classes the root object resides at position 0 and the left and right child of an object stored at position i is located at index $2i + 1$ and $2i + 2$, respectively. The second allocator, called align, aligns an object to cache line boundaries and adds padding space to occupy the rest of the cache line such that no other object is stored in the same cache line. The align allocator avoids false sharing at the expense of wasted memory and cache space. Both, the compact and the align allocators allocate and deallocate in constant time modulo the underlying allocator. Both are built on top of ptmalloc2 and therefore share its temporal and spatial performance characteristics. We point that out in the relevant parts of the experimental evaluation.

4.2 Capabilities of ACDC: allocation and deallocation time

We measure allocation and deallocation time for an increasing heap size without accessing any objects. The heap size is increased by increasing the deallocation delay without changing any other parameters which may affect allocation and deallocation time, e.g., size, liveness and number of objects. The data in Figure 7 confirms that the deallocation delay does indeed translate nearly linearly into heap size for all allocators. The non-default portion of the ACDC configuration for this experiment is in Table 2.

Parameter	Value
min. size	8 B
max. size	8 KB
deallocation delay	increasing from 0 to 15
time quantum	50 MB
access live objects	FALSE

Table 2: ACDC configuration for the allocation and deallocation time experiment (only non-default values are shown)

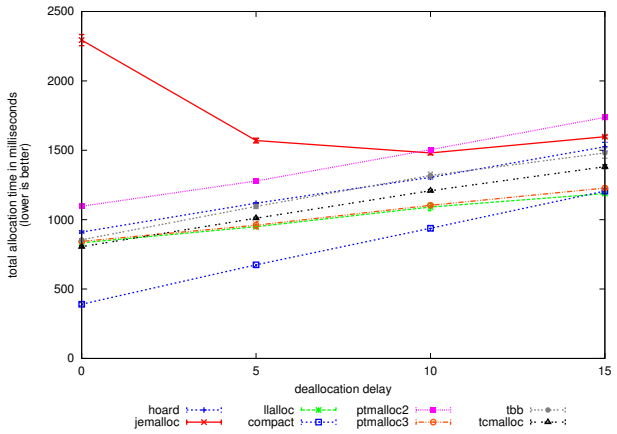


Figure 8: Total allocation time for an increasing deallocation delay

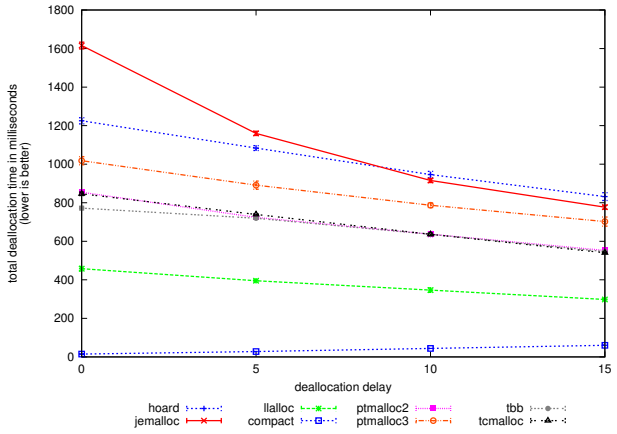


Figure 9: Total deallocation time for an increasing deallocation delay

Figure 8 depicts the total allocation time for an increasing deallocation delay. With the exception of jemalloc, the allocation time of all allocators increases close to linearly with the deallocation delay. However, the slope of the graphs differ by a constant factor where llalloc is less affected by the heap size than the others. The only anomaly in this experiment relative to the other allocators is the behavior of jemalloc.

Figure 9 shows, again for an increasing deallocation delay, the total deallocation time which, unlike the total allocation time, decreases rather than increases. In this case, however, the slope of the graphs is nearly the same for all allocators, again except for jemalloc. We included the compact allocator in this and other experiments below to show experimentally a performance baseline which

Parameter	Value
min. size	increasing from 8 to 1024 B
max. size	2 * min. size
max. liveness	1
time quantum	10 MB
benchmark duration	20
access live objects	FALSE

Table 3: ACDC configuration for the memory consumption experiment (only non-default values are shown)

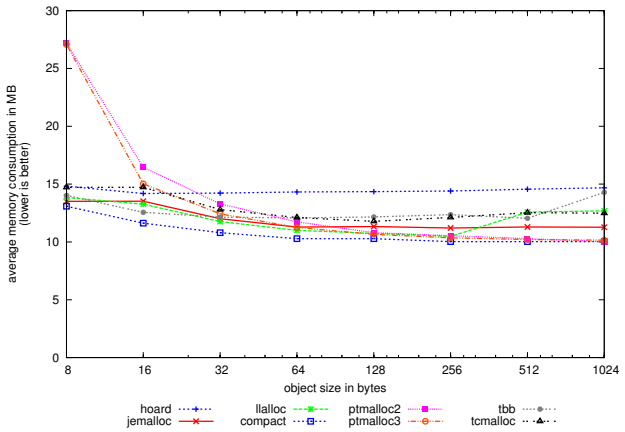


Figure 10: Memory consumption for increasing object sizes. On the x-axis a value for x denotes the range of object sizes from x to $2x$.

may only be reached by mutator-aware allocators. The compact allocator allocates and deallocates whole lifetime-size-classes rather than single objects with a single malloc and free call, respectively.

4.3 Capabilities of ACDC: memory consumption

We are interested in the space overhead introduced by allocators depending on the size of allocated objects. For this purpose, we measure average memory consumption for increasing object sizes (by increasing minimum and maximum sizes in ACDC). Table 3 summarizes the non-default portion of the ACDC configuration for this experiment.

Figure 10 shows the average memory consumption for increasing object sizes. Note that on the x-axis a value for x actually denotes the range of object sizes from x to $2x$. ACDC selects actual sizes randomly from this range. The ptmalloc2 and ptmalloc3 allocators introduce significant space overhead for small objects, possibly caused by the minimum 16-byte alignment. In contrast, both allocators introduce up to 20% less overhead for larger objects than the other allocators in this experiment.

4.4 Capabilities of ACDC: spatial locality

We measure total memory access time for an increasing ratio of list-based rather than tree-based lifetime-size-classes where objects are accessed (and deallocated) increasingly in the order in which they were allocated, i.e., with increasing spatial locality up to sequential locality. The non-default portion of the ACDC configuration for this experiment is in Table 4. The results are shown in Figure 11.

The compact allocator provides the best memory layout in terms of spatial locality because no memory is wasted between objects and the distance between successively accessed objects in memory is minimal. A higher ratio of list-based lifetime-size-classes increases spatial locality even more because the chances for the next

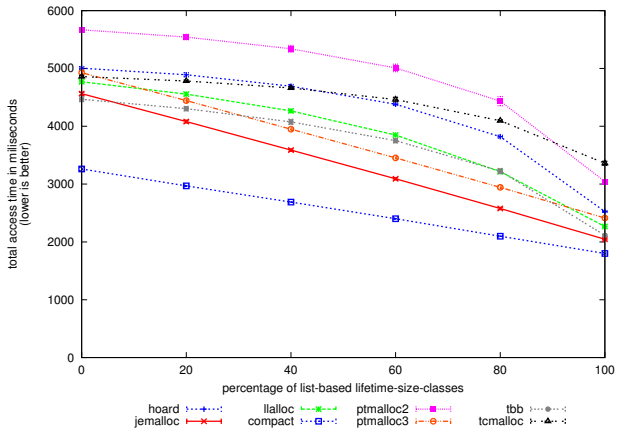


Figure 11: Total memory access time for an increasing ratio of list-based lifetime-size-classes

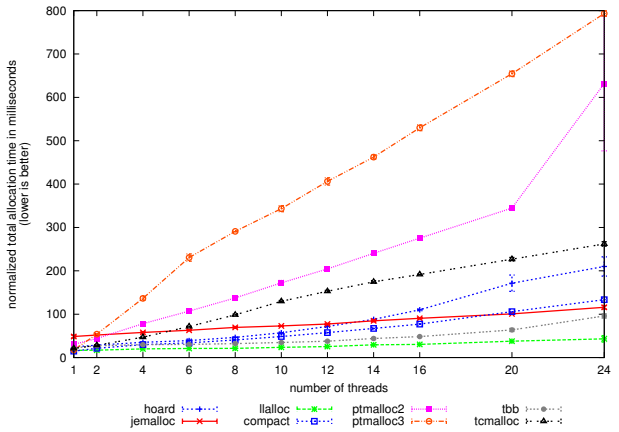


Figure 12: Normalized total allocation time for an increasing number of threads running the thread-local configuration without memory access

Parameter	Value
min. size	16 B
max. size	32 B
time quantum	500 KB
list-based ratio	increasing from 0% to 100%
write access ratio	0%

Table 4: ACDC configuration for the spatial locality experiment (only non-default values are shown)

Parameter	Value
mode	ACDC
number of threads	increasing from 1 to 24
min. size	8 B
max. size	2 KB
min. liveness	1
max. liveness	5
time quantum	1 MB
max. time drift	5
access live objects	FALSE (without memory access) or TRUE (with memory access)
shared objects	FALSE (thread-local configuration) or TRUE (shared-objects configuration)

Table 5: ACDC configuration for all multicore scalability experiments (only non-default values are shown)

object to already reside in the same cache line is higher for compact list-based than for compact tree-based lifetime-size-classes.

For the other allocators, the quality of the memory layout in terms of spatial locality also increases as the access order approaches the allocation order. However, jemaloc, llalloc, and tbb create a memory layout that benefits from spatial locality even more than ptmalloc2 and tcmalloc.

4.5 Capabilities of ACDC: multicore scalability

We are interested in exposing multicore scalability of allocators in terms of allocation and deallocation time as well as memory consumption. We benchmark thread-local and shared-objects configurations with and without accessing objects. The non-default portion

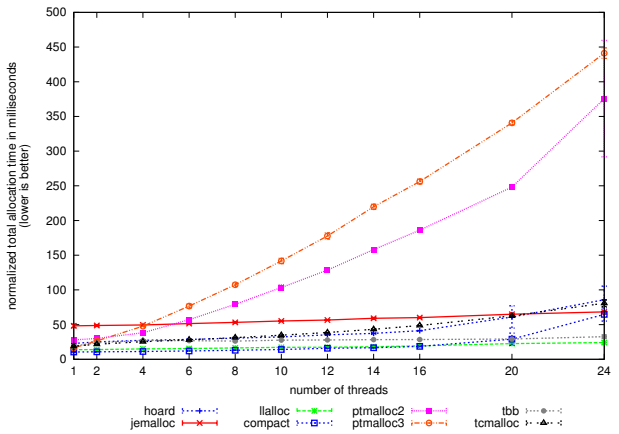


Figure 13: Normalized total allocation time for an increasing number of threads running the thread-local configuration with memory access

of the ACDC configuration for all four experiments are summarized in Table 5.

Figure 12 depicts the normalized total allocation time for an increasing number of threads in the thread-local configuration without memory access. This configuration puts high pressure on the allocator because ACDC performs no other operations than allocating and deallocating objects. The allocation time increases for all allocators but the slopes differ significantly. The llalloc allocator performs best in this experiment showing nearly perfect scalability (constant normalized allocation time). Also jemalloc scales well, however showing higher absolute allocation time. On the other hand, ptmalloc3 seems to suffer from contention on its locks. The situation with ptmalloc2 is similar, however less dramatic.

Figure 13 shows the data when ACDC performs memory access in between allocation and deallocation which reduces the pressure on the allocators. However, both ptmalloc2 and ptmalloc3 still do not scale but their absolute allocation times are much better, especially with ptmalloc3. The other allocators all scale well. The compact allocator, which is built on top of ptmalloc2, shows for more than 20 threads an increasing allocation time because the

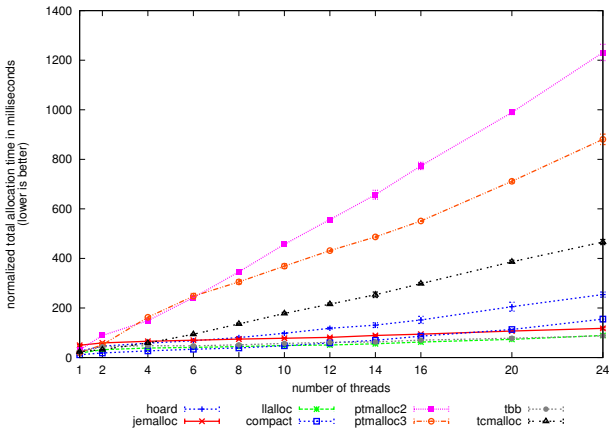


Figure 14: Normalized total allocation time for an increasing number of threads running the shared-objects configuration without memory access

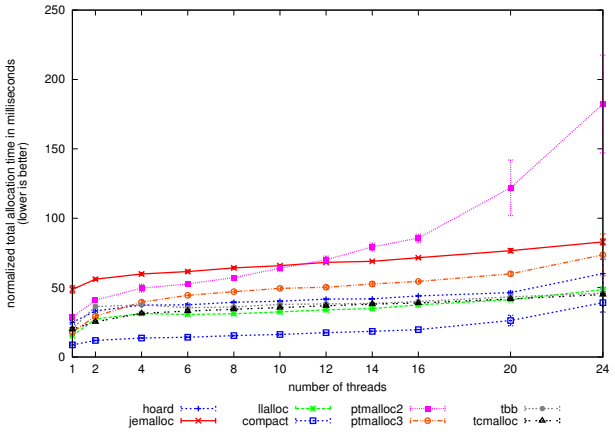


Figure 15: Normalized total allocation time for an increasing number of threads running the shared-objects configuration with memory access

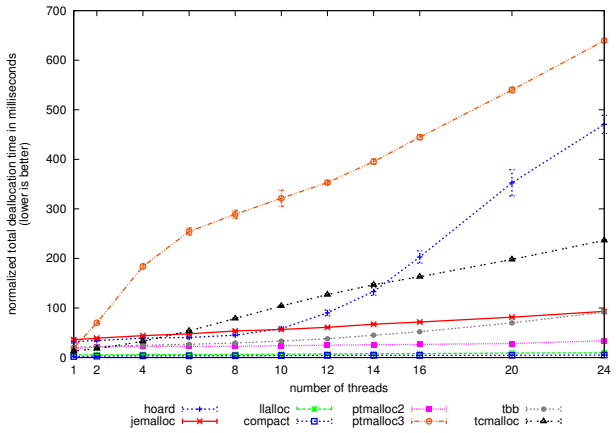


Figure 16: Normalized total deallocation time for an increasing number of threads running the thread-local configuration without memory access

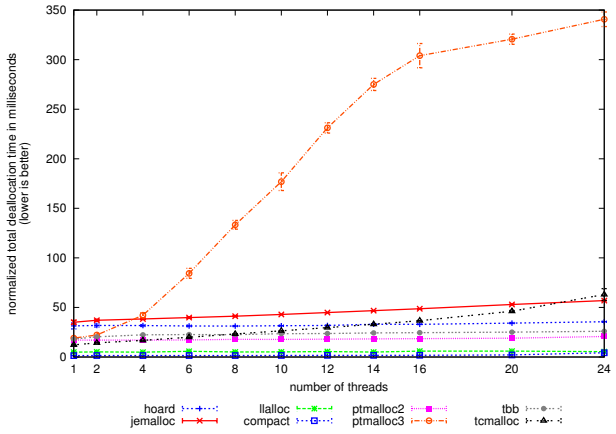


Figure 17: Normalized total deallocation time for an increasing number of threads running the thread-local configuration with memory access

scalability deficiencies of ptmalloc2 dominate the allocation time of the compact allocator.

Figure 14 depicts the data for the shared-objects configuration without memory access. Now, threads have to deallocate objects that were allocated by another thread. The result is higher total allocation time for those allocators that fail to scale in this experiment, namely ptmalloc2 and ptmalloc3. However, now ptmalloc3 performs better than ptmalloc2. The scalable allocators perform similar to the thread-local configuration except tcmalloc which takes about 80% longer to handle allocations in the shared-object configuration than in the thread-local configuration.

When we run ACDC with memory access in the shared-objects configuration the pressure on the allocators drops again. Figure 15 depicts the data for that configuration. Here, the absolute values are much better than in the other three configurations because access to shared objects takes more time than access to unshared objects. This effect decreases the pressure on the allocators even more resulting in less contention and lower allocation times.

The normalized total deallocation time of the thread-local configuration without memory access is shown in Figure 16. In this experiment, llalloc is again the fastest and most scalable allocator. Note, however, that ptmalloc2 scales nearly perfect in terms of deallocation time. This shows the advantage of ACDC over the benchmarks discussed in Section 3. A benchmark that does not separate allocation and deallocation time is unable to show this phenomenon. The ptmalloc3 allocator does not scale in this experiment and also tcmalloc and Hoard show a significant increase in deallocation time for more than six and ten threads, respectively.

For Figure 17 ACDC again performs memory access in between allocation and deallocation. Reducing the pressure on their deallocation routines, all allocators perform well except ptmalloc3 which, however, produces much better absolute deallocation times than in the experiment without memory access.

Figure 18 depicts the normalized total deallocation time of the shared-objects configuration without memory access. Here, ptmalloc3 and ptmalloc2 show slight scalability deficiencies. However, ptmalloc 3 gives much better absolute values than in the thread-local configuration while ptmalloc2 takes much longer to deallocate shared objects. Still, the overall results are better than for the thread-local configuration.

Adding memory access to the shared-objects configuration yields the normalized total deallocation time shown in Figure 19.

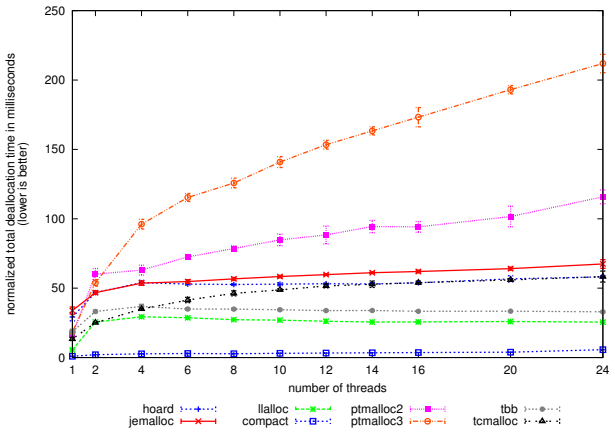


Figure 18: Normalized total deallocation time for an increasing number of threads running the shared-objects configuration without memory access

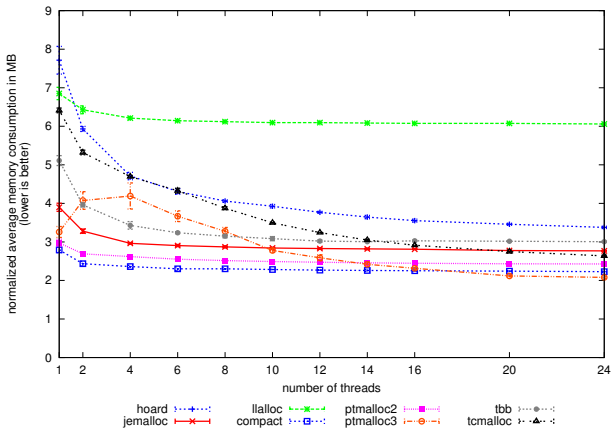


Figure 20: Normalized average memory consumption for an increasing number of threads running the thread-local configuration

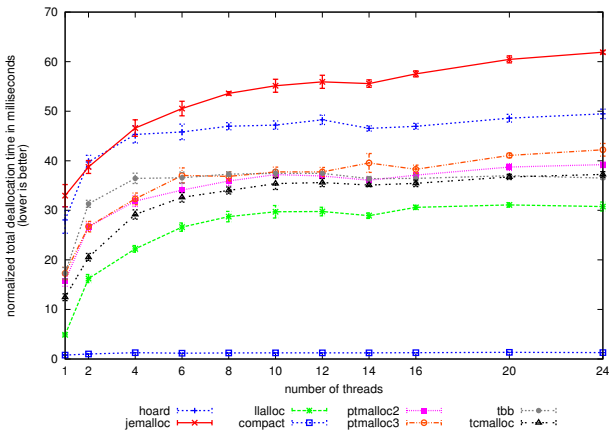


Figure 19: Normalized total deallocation time for an increasing number of threads running the shared-objects configuration with memory access

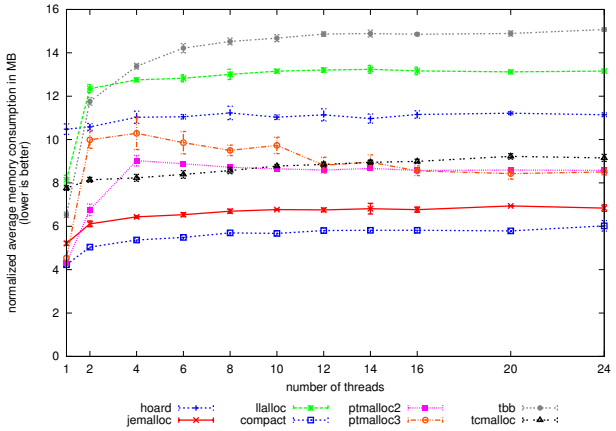


Figure 21: Normalized average memory consumption for an increasing number of threads running the shared-objects configuration

We observe a similar situation as with the allocation time for the shared-objects configuration with memory access. Relaxing the pressure on the deallocation routines results in fast and scalable deallocation of shared objects even for an increasing number of threads.

The normalized average memory consumption of this experiment running the thread-local configuration is presented in Figure 20. The result illustrates the time-space trade-off that the different allocators implement. The llalloc allocator, the fastest and most scalable allocator in this evaluation, shows the highest per-thread memory consumption while ptmalloc2 and ptmalloc3, which did not scale in terms of allocation time, are the most space efficient allocators in this experiment. The tbb, tcmalloc, and jemalloc allocators implement a more balanced trade-off between time and space.

For the shared-objects configuration, Figure 21 shows the normalized average memory consumption. We observe an increasing per-thread memory consumption in this experiment. Deallocating objects that were allocated by a different thread can cause blowup memory fragmentation. This experiment illustrates how effectively the allocators handle this problem. Apparently, none of the allocators create unbounded memory consumption. However, the differences in the absolute space demands is significant. The tbb allocator, for example, consumes twice the amount of memory the jemalloc allocator consumes.

Parameter	Value
mode	false-sharing
number of threads	increasing from 1 to 24
min. size	10 B
max. size	10 B
min. liveness	1
max. liveness	1
benchmark duration	30
time quantum	1 million read and write accesses

Table 6: ACDC configuration for the false-sharing experiment (only non-default values are shown)

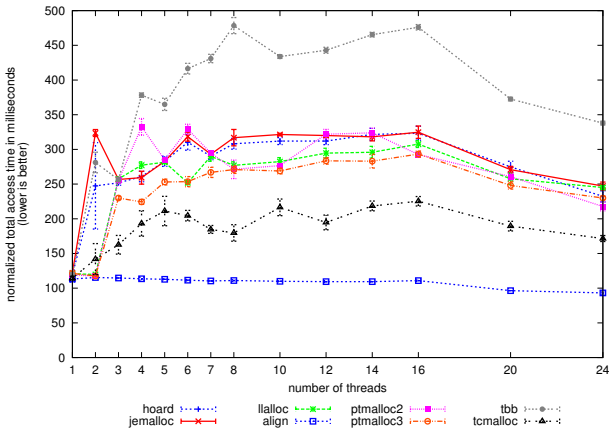


Figure 22: Normalized total access time for an increasing number of threads

4.6 Capabilities of ACDC: false sharing

ACDC allows to run a special mode to expose memory layouts which are prone to false sharing. In this mode one thread allocates as many objects as there are threads, one for each thread including itself. The allocating thread (single producer) passes each object to a different thread (multiple consumers) and then all threads perform a large number of read and write accesses to their object until its lifetime ends. Note that the time quantum in this mode is given in read and write accesses, i.e., after a given number of accesses to the objects the thread clocks advance and a new object is allocated for each thread. Table 6 summarizes the non-default portion of the ACDC configuration for this experiment.

Figure 22 shows the normalized total access time for an increasing number of threads running ACDC in false-sharing mode. The baseline is here the align allocator where each object is allocated in its own cache line avoiding false sharing altogether. For only one thread, of course, there is no false sharing and all allocators yield the same access time. For an increasing number of threads, however, the access time increases up to a factor of five. This experiment in combination with the findings from Figure 11 illustrate yet another time-space trade-off. The allocators which performed best in terms of spacial locality, namely tbb, jemalloc, and llalloc are more prone to introduce false sharing.

5. Conclusion

We have presented ACDC, an open-source benchmark for measuring allocator performance by emulating realistic single- and multi-threaded mutators. We have presented the basic modes of operation of ACDC including allocation and deallocation of objects, emulation of heap access patterns, and sharing objects among multiple threads. In an empirical study involving seven state-of-the-art allocators we showed that ACDC is able to expose differences in their performance in terms of allocation, deallocation, and memory access throughput as well as memory consumption and multicore scalability and we also illustrated the time-space trade-offs implemented by the allocators. As part of future work we plan to extend ACDC for benchmarking managed languages.

Acknowledgments

This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23). We thank the anonymous referees for their constructive and inspiring comments and suggestions.

References

[1] CITI Projects: Linux scalability, 1999. URL `http://www.citi.umich.edu/projects/linux-scalability/`.

[2] MicroQuill Inc., 2013. URL `http://www.microquill.com/`.

[3] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 187–196, New York, USA, 1993. ACM.

[4] E. D. Berger. The Hoard Memory Allocator Documentation: Frequently Asked Questions, 2004. URL `http://people.cs.umass.edu/~emery/hoard/hoard-documentation.html#IDAOZYP`.

[5] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not.*, 35(11):117–128, Nov. 2000.

[6] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.

[7] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD. In *The Technical BSD Conference*, BSDCan '06, Apr. 2006.

[8] J. Evans. Scalable memory allocation using jemalloc, 2011. URL `https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`.

[9] W. Gloger. Wolfram Gloger's malloc homepage, 2006. URL `http://www.malloc.de/en/`.

[10] Google, Inc. Google Performance Tools, 2012. URL `http://code.google.com/p/gperftools/wiki/GooglePerformanceTools`.

[11] B. Hayes. Using key object opportunism to collect old objects. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 33–46, New York, USA, 1991. ACM.

[12] Intel, Inc. Thread Building Blocks, 2013. URL `http://threadingbuildingblocks.org/`.

[13] P.-A. Larson and M. Krishnan. Memory allocation for long-running server applications. In *Proceedings of the 1st international symposium on Memory management*, ISMM '98, pages 176–185, New York, USA, 1998. ACM.

[14] D. Lea. A Memory Allocator, 2000. URL `http://g.oswego.edu/dl/html/malloc.html`.

[15] C. Lever and D. Boreham. malloc() performance in a multithreaded Linux environment. In *Proceedings of the FREENIX Track of the 2000 USENIX Annual Technical Conference*, June 2001.

[16] Lockless, Inc. The Lockless Memory Allocator, 2013. URL `http://locklessinc.com/`.

[17] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 35–46, New York, USA, 2004. ACM.

[18] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 84–94, New York, USA, 2006. ACM.

[19] Standard Performance Evaluation Corporation. Specjvm2008, 2013. URL `http://www.spec.org/jvm2008/`.

[20] R. C. Weisner. How Memory Allocation Affects Performance in Multithreaded Programs, 2012. URL `http://www.oracle.com/technetwork/articles/servers-storage-dev/mem-alloc-1557798.html`.

[21] B. Zorn and D. Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Not.*, 27(12):71–80, Dec. 1992.