High-Performance Concurrent Memory Allocation

by

Mubeen Zulfiqar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Memory management takes a sequence of program generated allocation/deallocation requests and attempts to satisfy them within a fixed-sized block of memory while minimizing the total amount of memory used. A general-purpose dynamic-allocation algorithm cannot anticipate future allocation requests so its output is rarely optimal. However, memory allocators do take advantage of regularities in allocation patterns for typical programs to produce excellent results, both in time and space (similar to LRU paging). In general, allocators use a number of similar techniques, each optimizing specific allocation patterns. Nevertheless, memory allocators are a series of compromises, occasionally with some static or dynamic tuning parameters to optimize specific program-request patterns.

The goal of this thesis is to build a low-latency memory allocator for both kernel and user multi-threaded systems, which is competitive with the best current memory allocators, while extending the feature set of existing and new allocator routines. A new llheap memory-allocator is created that achieves all of these goals, while maintaining and managing sticky allocation properties for zero-filled and aligned allocations without a performance loss. Hence, it becomes possible to use realloc frequently as a safe operation, rather than just occasionally, because it preserves sticky properties when enlarging storage requests. Furthermore, the ability to query sticky properties and information allows programmers to write safer programs, as it is possible to dynamically match allocation styles from unknown library routines that return allocations. The C allocation API is also extended with resize, advanced realloc, aalloc, amemalign, and cmemalign so programmers do not make mistakes writing theses useful allocation operations. llheap is embedded into the μC++ and C$\forall$ runtime systems, both of which have user-level threading. The ability to use C$\forall$'s advanced type-system (and possibly C++'s too) to combine advanced memory operations into one allocation routine using named arguments shows how far the allocation API can be pushed, which increases safety and greatly simplifies programmer's use of dynamic allocation.

The llheap allocator also provides comprehensive statistics for all allocation operations, which are invaluable in understanding and debugging a program's dynamic behaviour. No other memory allocator examined in the thesis provides such comprehensive statistics gathering. As well, llheap provides a debugging mode where allocations are checked with internal pre/post conditions and invariants. It is extremely useful, especially for students. While not as powerful as the valgrind interpreter, a large number of allocations mistakes are detected. Finally, contention-free statistics gathering and debugging have a low enough cost to be used in production code.

A micro-benchmark test-suite is started for comparing allocators, rather than relying on a suite of arbitrary programs. It has been an interesting challenge. These micro-benchmarks have adjustment knobs to simulate allocation patterns hard-coded into arbitrary test programs. Existing memory allocators, glibc, dlmalloc, hoard, jemalloc, ptmalloc3, rpmalloc, tbmalloc, and the new allocator llheap are all compared using the new micro-benchmark test-suite.

Acknowledgements

I would like to thank all the people who made this thesis possible.

I would like to acknowledge Peter A. Buhr for his assistance and support throughout the process. It would have been impossible without him.

I would like to acknowledge Gregor Richards and Trevor Brown for reading my thesis quickly and giving me great feedback on my work.

Also, I would say thanks to my team members at PLG especially Thierry, Michael, and Andrew for their input.

Finally, a special thank you to Huawei Canada for funding this work.

Table of Contents

List of Figures

Chapter 1

Introduction

Memory management takes a sequence of program generated allocation/deallocation requests and attempts to satisfy them within a fixed-sized block of memory while minimizing the total amount of memory used. A general-purpose dynamic-allocation algorithm cannot anticipate future allocation requests so its output is rarely optimal. However, memory allocators do take advantage of regularities in allocation patterns for typical programs to produce excellent results, both in time and space (similar to LRU paging). In general, allocators use a number of similar techniques, each optimizing specific allocation patterns. Nevertheless, memory allocators are a series of compromises, occasionally with some static or dynamic tuning parameters to optimize specific program-request patterns.

1.1 Memory Structure

Figure 1.1 shows the typical layout of a program's address space divided into the following zones (right to left): static code/data, dynamic allocation, dynamic code/data, and stack, with free memory surrounding the dynamic code/data [39]. Static code and data are placed into memory at load time from the executable and are fixed-sized at runtime. Dynamic-allocation memory starts empty and grows/shrinks as the program dynamically creates/deletes variables with independent lifetime. The programming-language's runtime manages this area, where management complexity is a function of the mechanism for deleting variables. Dynamic code/data memory is managed by the dynamic loader for libraries loaded at runtime, which is complex especially in a multi-threaded program [21]. However, changes to the dynamic code/data space are typically infrequent, many occurring at program startup, and are largely outside of a program's control. Stack memory is managed by the program call-mechanism using a simple LIFO technique, which works well for sequential programs. For multi-threaded programs (and coroutines), a new stack is created for each thread; these thread stacks are commonly created in dynamic-allocation memory. This thesis focuses on management of the dynamic-allocation memory.

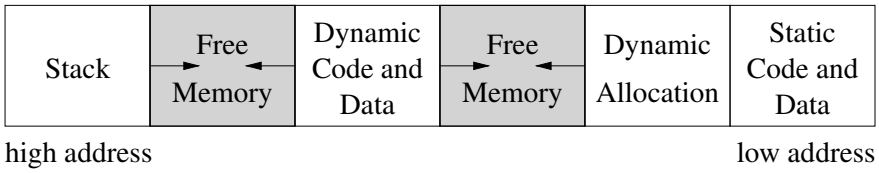


Figure 1.1: Program Address Space Divided into Zones

1.2 Dynamic Memory-Management

Modern programming languages manage dynamic-allocation memory in different ways. Some languages, such as Lisp [42], Java [17], Haskell [22], Go [18], provide explicit allocation but *implicit* deallocation of data through garbage collection [46]. In general, garbage collection supports memory compaction, where dynamic (live) data is moved during runtime to better utilize space. However, moving data requires finding pointers to it and updating them to reflect new data locations. Programming languages such as C [26], C++ [11], and Rust [38] provide the programmer with explicit allocation *and* deallocation of data. These languages cannot find and subsequently move live data because pointers can be created to any storage zone, including internal components of allocated objects, and may contain temporary invalid values generated by pointer arithmetic. Attempts have been made to perform quasi garbage collection in C/C++ [6], but it is a compromise. This thesis only examines dynamic memory-management with *explicit* deallocation. While garbage collection and compaction are not part this work, many of the work's results are applicable to the allocation phase in any memory-management approach.

Most programs use a general-purpose allocator, often the one provided implicitly by the programming-language's runtime. When this allocator proves inadequate, programmers often write specialize allocators for specific needs. C and C++ allow easy replacement of the default memory allocator with an alternative specialized or general-purpose memory-allocator. Jikes RVM MMTk [5] provides a similar generalization for the Java virtual machine. However, high-performance memory-allocators for kernel and user multi-threaded programs are still being designed and improved. For this reason, several alternative general-purpose allocators have been written for C/C++ with the goal of scaling in a multi-threaded program [2, 33, 40, 13]. This thesis examines the design of high-performance allocators for use by kernel and user multi-threaded applications written in C/C++.

1.3 Contributions

This work provides the following contributions in the area of concurrent dynamic allocation:

1. Implementation of a new stand-alone concurrent low-latency memory-allocator ($\approx$1,200 lines of code) for C/C++ programs using kernel threads (1:1 threading), and specialized versions of the allocator for the programming languages μC++ and CFA using user-level threads running over multiple kernel threads (M:N threading).
2. Extend the standard C heap functionality by preserving with each allocation:
   - its request size plus the amount allocated,
   - whether an allocation is zero fill,
   - and allocation alignment.
3. Use the preserved zero fill and alignment as *sticky* properties for realloc to zero-fill and align when storage is extended or copied. Without this extension, it is unsafe to realloc storage initially allocated with zero-fill/alignment as these properties are not preserved when copying. This silent generation of a problem is unintuitive to programmers and difficult to locate because it is transient.

4. Provide additional heap operations to complete programmer expectation with respect to accessing different allocation properties.
    - resize(oaddr, size) re-purpose an old allocation for a new type *without* preserving fill or alignment.
    - resize(oaddr, alignment, size) re-purpose an old allocation with new alignment but *without* preserving fill.
    - realloc(oaddr, alignment, size) same as realloc but adding or changing alignment.
    - aalloc(dim, elemSize) same as calloc except memory is *not* zero filled.
    - amemalign(alignment, dim, elemSize) same as aalloc with memory alignment.
    - cmemalign(alignment, dim, elemSize) same as calloc with memory alignment.
5. Provide additional heap wrapper functions in C∀ creating a more usable set of allocation operations and properties.
6. Provide additional query operations to access information about an allocation:
    - malloc_alignment(addr) returns the alignment of the allocation pointed-to by addr. If the allocation is not aligned or addr is the NULL, the minimal alignment is returned.
    - malloc_zero_fill(addr) returns a boolean result indicating if the memory pointed-to by addr is allocated with zero fill, e.g., by calloc/cmemalign.
    - malloc_size(addr) returns the size of the memory allocation pointed-to by addr.
    - malloc_usable_size(addr) returns the usable (total) size of the memory pointed-to by addr, i.e., the bin size containing the allocation, where malloc_size(addr) $\leq$ malloc_usable_size(addr).
7. Provide complete, fast, and contention-free allocation statistics to help understand allocation behaviour:
    - malloc_stats() print memory-allocation statistics on the file-descriptor set by malloc_stats_fd.
    - malloc_info(options, stream) print memory-allocation statistics as an XML string on the specified file-descriptor set by malloc_stats_fd.
    - malloc_stats_fd(fd) set file-descriptor number for printing memory-allocation statistics (default STDERR_FILENO). This file descriptor is used implicitly by malloc_stats and malloc_info.
8. Provide extensive runtime checks to validate allocation operations and identify the amount of unfreed storage at program termination.
9. Build 4 different versions of the allocator:
    - static or dynamic linking
    - statistic/debugging (testing) or no statistic/debugging (performance)

    A program may link to any of these 4 versions of the allocator often without recompilation. (It is possible to separate statistics and debugging, giving 8 different versions.)
10. A micro-benchmark test-suite for comparing allocators rather than relying on a suite of arbitrary programs. These micro-benchmarks have adjustment knobs to simulate allocation patterns hard-coded into arbitrary test programs

Chapter 2

Background[1]

A program dynamically allocates and deallocates the storage for a variable, referred to as an *object*, through calls such as malloc and free in C, and new and delete in C++. Space for each allocated object comes from the dynamic-allocation zone. A *memory allocator* contains a complex data-structure and code that manages the layout of objects in the dynamic-allocation zone. The management goals are to make allocation/deallocation operations as fast as possible while densely packing objects to make efficient use of memory. Objects in C/C++ cannot be moved to aid the packing process, only adjacent free storage can be *coalesced* into larger free areas. The allocator grows or shrinks the dynamic-allocation zone to obtain storage for objects and reduce memory usage via operating-system calls, such as mmap or sbrk in UNIX.

2.1 Allocator Components

Figure 2.1 shows the two important data components for a memory allocator, management and storage, collectively called the *heap*. The *management data* is a data structure located at a known memory address and contains all information necessary to manage the storage data. The management data starts with fixed-sized information in the static-data memory that references components in the dynamic-allocation memory. The *storage data* is composed of allocated and freed objects, and *reserved memory*. Allocated objects (light grey) are variable sized, and are allocated and maintained by the program; *i.e.*, only the memory allocator knows the location of allocated storage, not the program. Freed objects (white) represent memory deallocated by the program, which are linked into one or more lists facilitating easy location of new allocations. Often the free list is chained internally so it does not consume additional storage, *i.e.*, the link fields are placed at known locations in the unused memory blocks. Reserved memory (dark grey) is one or more blocks of memory obtained from the operating system but not yet allocated to the program; if there are multiple reserved blocks, they are also chained together, usually internally.

In some allocator designs, allocated and freed objects have additional management data embedded within them. Figure 2.2 shows an allocated object with a header, trailer, and alignment padding and spacing around the object. The header contains information about the object, *e.g.*, size, type, etc. The trailer may be used to simplify an allocation implementation, *e.g.*, coalescing, and/or for security purposes to mark the end of an object. An object may be preceded by padding

[1]Part of this chapter draws from similar background work in [45] with many updates.

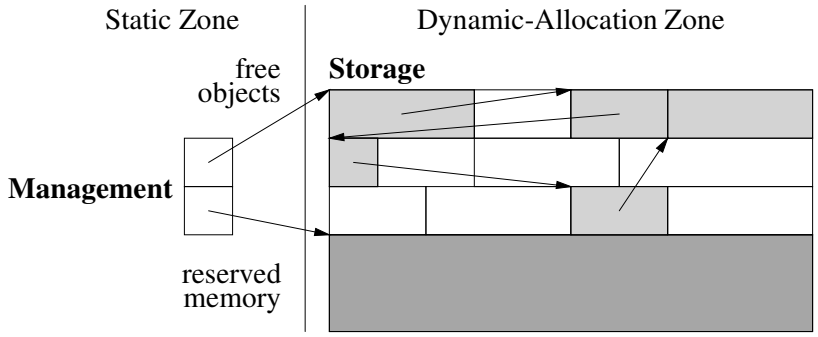


Figure 2.1: Allocator Components (Heap)

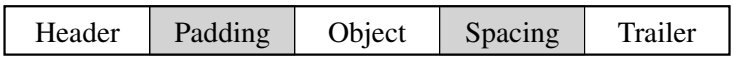


Figure 2.2: Allocated Object

to ensure proper alignment. Some algorithms quantize allocation requests into distinct sizes, called *buckets*, resulting in additional spacing after objects less than the quantized value. (Note, the buckets are often organized as an array of ascending bucket sizes for fast searching, *e.g.*, binary search, and the array is stored in the heap management-area, where each bucket is a top point to the freed objects of that size.) When padding and spacing are necessary, neither can be used to satisfy a future allocation request while the current allocation exists. A free object also contains management data, *e.g.*, size, chaining, etc. The amount of management data for a free node defines the minimum allocation size, *e.g.*, if 16 bytes are needed for a free-list node, any allocation request less than 16 bytes must be rounded up, otherwise the free list cannot use internal chaining. The information in an allocated or freed object is overwritten when it transitions from allocated to freed and vice-versa by new management information and possibly data.

2.2 Single-Threaded Memory-Allocator

A single-threaded memory-allocator does not run any threads itself, but is used by a single-threaded program. Because the memory allocator is only executed by a single thread, concurrency issues do not exist. The primary issues in designing a single-threaded memory-allocator are fragmentation and locality.

2.2.1 Fragmentation

Fragmentation is memory requested from the operating system but not used by the program; hence, allocated objects are not fragmentation. Figure 2.3 shows fragmentation is divided into two forms: internal or external.

Internal fragmentation is memory space that is allocated to the program, but is not intended to be accessed by the program, such as headers, trailers, padding, and spacing around an allocated object. This memory is typically used by the allocator for management purposes or required

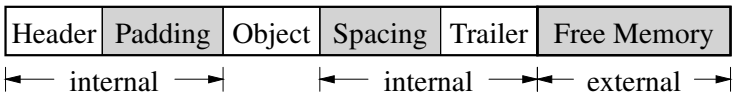


Figure 2.3: Internal and External Fragmentation

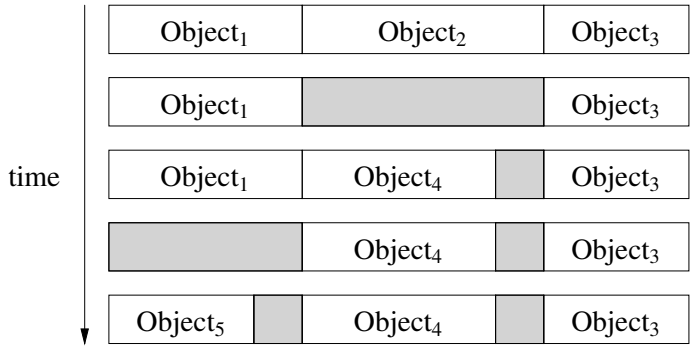
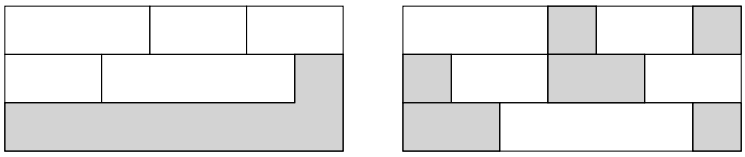


Figure 2.4: Memory Fragmentation



(a) Contiguous (b) Highly Fragmented

Figure 2.5: Fragmentation Quality

by the architecture for correctness, *e.g.*, alignment. Internal fragmentation is problematic when management space is a significant proportion of an allocated object. For example, if internal fragmentation is as large as the object being managed, then the memory usage for that object is doubled. An allocator should strive to keep internal management information to a minimum.

External fragmentation is all memory space reserved from the operating system but not allocated to the program [47, 31, 41], which includes all external management data, freed objects, and reserved memory. This memory is problematic in two ways: heap blowup and highly fragmented memory. *Heap blowup* occurs when memory freed by the program is not reused for future allocations leading to potentially unbounded external fragmentation growth [2]. Heap blowup can occur due to allocator policies that are too restrictive in reusing freed memory and/or no coalescing of free storage. Memory can become *highly fragmented* after multiple allocations and deallocations of objects. Figure 2.4 shows an example of how a small block of memory fragments as objects are allocated and deallocated over time. Blocks of free memory become smaller and non-contiguous making them less useful in serving allocation requests. Memory is highly fragmented when most free blocks are unusable because of their sizes. For example, Figure 2.5(a) and Figure 2.5(b) have the same quantity of external fragmentation, but Figure 2.5(b) is highly fragmented. If there is a request to allocate a large object, Figure 2.5(a) is more likely to be able to satisfy it with existing free memory, while Figure 2.5(b) likely has to request more memory from the operating system.

For a single-threaded memory allocator, three basic approaches for controlling fragmentation are identified [25]. The first approach is a *sequential-fit algorithm* with one list of free objects that is searched for a block large enough to fit a requested object size. Different search policies determine the free object selected, *e.g.*, the first free object large enough or closest to the requested size. Any storage larger than the request can become spacing after the object or be split into a smaller free object. The cost of the search depends on the shape and quality of the free list, *e.g.*, a linear versus a binary-tree free-list, a sorted versus unsorted free-list.

The second approach is a *segregated* or *binning algorithm* with a set of lists for different sized freed objects. When an object is allocated, the requested size is rounded up to the nearest bin-size, often leading to spacing after the object. A binning algorithm is fast at finding free memory of the appropriate size and allocating it, since the first free object on the free list is used. The fewer bin-sizes, the fewer lists need to be searched and maintained; however, the bin sizes are less likely to closely fit the requested object size, leading to more internal fragmentation. The more bin sizes, the longer the search and the less likely free objects are to be reused, leading to more external fragmentation and potentially heap blowup. A variation of the binning algorithm allows objects to be allocated to the requested size, but when an object is freed, it is placed on the free list of the next smallest or equal bin-size. For example, with bin sizes of 8 and 16 bytes, a request for 12 bytes allocates only 12 bytes, but when the object is freed, it is placed on the 8-byte bin-list. For subsequent requests, the bin free-lists contain objects of different sizes, ranging from one bin-size to the next (8-16 in this example), and a sequential-fit algorithm may be used to find an object large enough for the requested size on the associated bin list.

The third approach is *splitting* and *coalescing algorithms*. When an object is allocated, if there are no free objects of the requested size, a larger free object may be split into two smaller objects to satisfy the allocation request without obtaining more memory from the operating system. For example, in the buddy system, a block of free memory is split into two equal chunks, one of those chunks is again split into two equal chunks, and so on until a block just large enough to fit the requested object is created. When an object is deallocated it is coalesced with the objects immediately before and after it in memory, if they are free, turning them into one larger object. Coalescing can be done eagerly at each deallocation or lazily when an allocation cannot be fulfilled. In all cases, coalescing increases allocation latency, hence some allocations can cause unbounded delays during coalescing. While coalescing does not reduce external fragmentation, the coalesced blocks improve fragmentation quality so future allocations are less likely to cause heap blowup. Splitting and coalescing can be used with other algorithms to avoid highly fragmented memory.

2.2.2 Locality

The principle of locality recognizes that programs tend to reference a small set of data, called a working set, for a certain period of time, where a working set is composed of temporal and spatial accesses [7]. Temporal clustering implies a group of objects are accessed repeatedly within a short time period, while spatial clustering implies a group of objects physically close together (nearby addresses) are accessed repeatedly within a short time period. Temporal locality commonly occurs during an iterative computation with a fixed set of disjoint variables, while spatial locality commonly occurs when traversing an array.

Hardware takes advantage of temporal and spatial locality through multiple levels of caching, *i.e.*, memory hierarchy. When an object is accessed, the memory physically located around the object is also cached with the expectation that the current and nearby objects will be referenced within a short period of time. For example, entire cache lines are transferred between memory and cache and entire virtual-memory pages are transferred between disk and memory. A program exhibiting good locality has better performance due to fewer cache misses and page faults[2].

Temporal locality is largely controlled by how a program accesses its variables [12]. Nevertheless, a memory allocator can have some indirect influence on temporal locality and largely dictates spatial locality. For temporal locality, an allocator can return storage for new allocations that was just freed as these memory locations are still *warm* in the memory hierarchy. For spatial locality, an allocator can place objects used together close together in memory, so the working set of the program fits into the fewest possible cache lines and pages. However, usage patterns are different for every program as is the underlying hardware memory architecture; hence, no general-purpose memory-allocator can provide ideal locality for every program on every computer.

There are a number of ways a memory allocator can degrade locality by increasing the working set. For example, a memory allocator may access multiple free objects before finding one to satisfy an allocation request, *e.g.*, sequential-fit algorithm. If there are a (large) number of objects accessed in very different areas of memory, the allocator may perturb the program's memory hierarchy causing multiple cache or page misses [19]. Another way locality can be degraded is by spatially separating related data. For example, in a binning allocator, objects of different sizes are allocated from different bins that may be located in different pages of memory.

2.3 Multi-Threaded Memory-Allocator

A multi-threaded memory-allocator does not run any threads itself, but is used by a multi-threaded program. In addition to single-threaded design issues of fragmentation and locality, a multi-threaded allocator is simultaneously accessed by multiple threads, and hence, must deal with concurrency issues such as mutual exclusion, false sharing, and additional forms of heap blowup.

2.3.1 Mutual Exclusion

Mutual exclusion provides sequential access to the shared management data of the heap. There are two performance issues for mutual exclusion. First is the overhead necessary to perform (at least) a hardware atomic operation every time a shared resource is accessed. Second is when multiple threads contend for a shared resource simultaneously, and hence, some threads must wait until the resource is released. Contention can be reduced in a number of ways:

- using multiple fine-grained locks versus a single lock, spreading the contention across a number of locks;

[2]With the advent of large RAM memory, paging is becoming less of an issue in modern programming.

- using trylock and generating new storage if the lock is busy, yielding a classic space versus time tradeoff;
- using one of the many lock-free approaches for reducing contention on basic data-structure operations [37].

However, all of these approaches have degenerate cases where program contention is high, which occurs outside of the allocator.

2.3.2 False Sharing

False sharing is a dynamic phenomenon leading to cache thrashing. When two or more threads on separate CPUs simultaneously change different objects sharing a cache line, the change invalidates the other thread's associated cache, even though these threads may be uninterested in the other modified object. False sharing can occur in three different ways: program induced, allocator-induced active, and allocator-induced passive; a memory allocator can only affect the latter two.

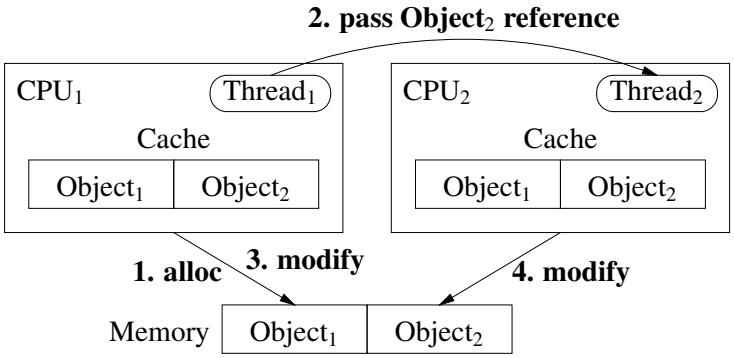
Program-induced false-sharing occurs when one thread passes an object sharing a cache line to another thread, and both threads modify the respective objects. Figure 2.6(a) shows when Thread_1 passes Object_2 to Thread_2, a false-sharing situation forms when Thread_1 modifies Object_1 and Thread_2 modifies Object_2. Changes to Object_1 invalidate CPU_2's cache line, and changes to Object_2 invalidate CPU_1's cache line.

Allocator-induced active false-sharing occurs when objects are allocated within the same cache line but to different threads. For example, in Figure 2.6(b), each thread allocates an object and loads a cache-line of memory into its associated cache. Again, changes to Object_1 invalidate CPU_2's cache line, and changes to Object_2 invalidate CPU_1's cache line.

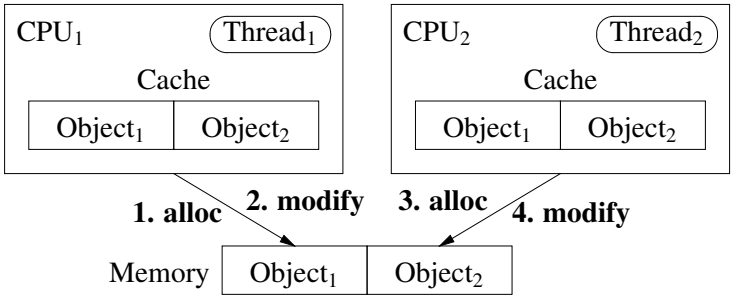
Allocator-induced passive false-sharing is another form of allocator-induced false-sharing caused by program-induced false-sharing. When an object in a program-induced false-sharing situation is deallocated, a future allocation of that object may cause passive false-sharing. For example, in Figure 2.6(c), Thread_1 passes Object_2 to Thread_2, and Thread_2 subsequently deallocates Object_2. Allocator-induced passive false-sharing occurs when Object_2 is reallocated to Thread_2 while Thread_1 is still using Object_1.

2.3.3 Heap Blowup

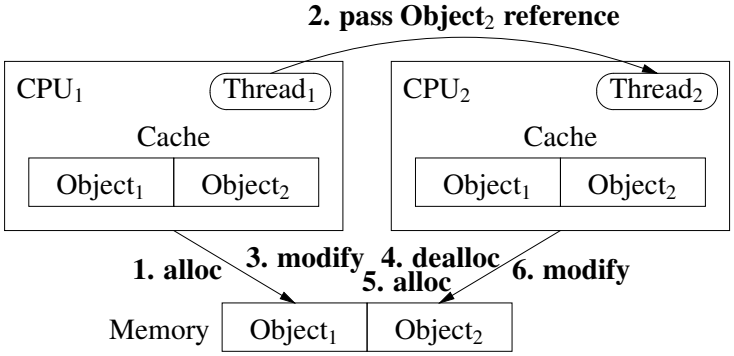
In a multi-threaded program, heap blowup can occur when memory freed by one thread is inaccessible to other threads due to the allocation strategy. Specific examples are presented in later sections.



(a) Program-Induced False-Sharing



(b) Allocator-Induced Active False-Sharing



(c) Allocator-Induced Passive False-Sharing

Figure 2.6: False Sharing

2.4 Multi-Threaded Memory-Allocator Features

The following features are used in the construction of multi-threaded memory-allocators:

1. multiple heaps
   a) with or without a global heap
   b) with or without ownership
2. object containers
   a) with or without ownership
   b) fixed or variable sized

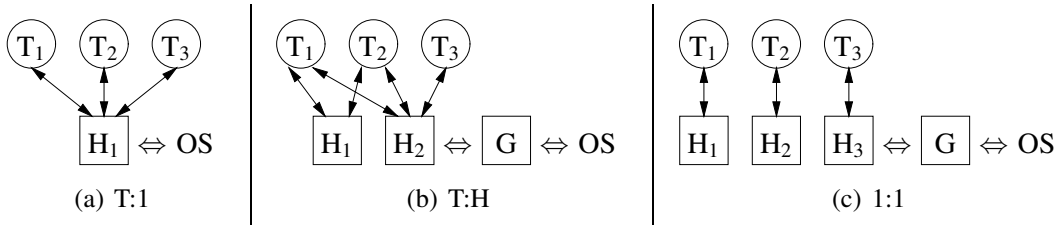


Figure 2.7: Multiple Heaps, Thread:Heap Relationship

   c) global or local free-lists
3. hybrid private/public heap
4. allocation buffer
5. lock-free operations

The first feature, multiple heaps, pertains to different kinds of heaps. The second feature, object containers, pertains to the organization of objects within the storage area. The remaining features apply to different parts of the allocator design or implementation.

2.5 Multiple Heaps

A multi-threaded allocator has potentially multiple threads and heaps. The multiple threads cause complexity, and multiple heaps are a mechanism for dealing with the complexity. The spectrum ranges from multiple threads using a single heap, denoted as T:1 (see Figure 2.7(a)), to multiple threads sharing multiple heaps, denoted as T:H (see Figure 2.7(b)), to one thread per heap, denoted as 1:1 (see Figure 2.7(c)), which is almost back to a single-threaded allocator.

T:1 model where all threads allocate and deallocate objects from one heap. Memory is obtained from the freed objects, or reserved memory in the heap, or from the operating system (OS); the heap may also return freed memory to the operating system. The arrows indicate the direction memory conceptually moves for each kind of operation: allocation moves memory along the path from the heap/operating-system to the user application, while deallocation moves memory along the path from the application back to the heap/operating-system. To safely handle concurrency, a single heap uses locking to provide mutual exclusion. Whether using a single lock for all heap operations or fine-grained locking for different operations, a single heap may be a significant source of contention for programs with a large amount of memory allocation.

T:H model where each thread allocates storage from several heaps depending on certain criteria, with the goal of reducing contention by spreading allocations/deallocations across the heaps. The decision on when to create a new heap and which heap a thread allocates from depends on the allocator design. The performance goal is to reduce the ratio of heaps to threads. In general, locking is required, since more than one thread may concurrently access a heap during its lifetime, but contention is reduced because fewer threads access a specific heap.

For example, multiple heaps are managed in a pool, starting with a single or a fixed number of heaps that increase/decrease depending on contention/space issues. At creation, a thread is

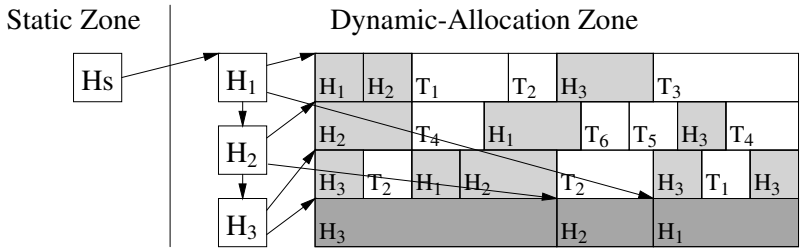


Figure 2.8: Multiple-Heap Storage

associated with a heap from the pool. In some implementations of this model, when the thread attempts an allocation and its associated heap is locked (contention), it scans for an unlocked heap in the pool. If an unlocked heap is found, the thread changes its association and uses that heap. If all heaps are locked, the thread may create a new heap, use it, and then place the new heap into the pool; or the thread can block waiting for a heap to become available. While the heap-pool approach often minimizes the number of extant heaps, the worse case can result in more heaps than threads; *e.g.*, if the number of threads is large at startup with many allocations creating a large number of heaps and then the number of threads reduces.

Threads using multiple heaps need to determine the specific heap to access for an allocation/deallocation, *i.e.*, association of thread to heap. A number of techniques are used to establish this association. The simplest approach is for each thread to have a pointer to its associated heap (or to administrative information that points to the heap), and this pointer changes if the association changes. For threading systems with thread-local storage, the heap pointer is created using this mechanism; otherwise, the heap routines must simulate thread-local storage using approaches like hashing the thread's stack-pointer or thread-id to find its associated heap.

The storage management for multiple heaps is more complex than for a single heap (see Figure 2.1, p. 5). Figure 2.8 illustrates the general storage layout for multiple heaps. Allocated and free objects are labelled by the thread or heap they are associated with. (Links between free objects are removed for simplicity.) The management information in the static zone must be able to locate all heaps in the dynamic zone. The management information for the heaps must reside in the dynamic-allocation zone if there are a variable number. Each heap in the dynamic zone is composed of a list of free objects and a pointer to its reserved memory. An alternative implementation is for all heaps to share one reserved memory, which requires a separate lock for the reserved storage to ensure mutual exclusion when acquiring new memory. Because multiple threads can allocate/free/reallocate adjacent storage, all forms of false sharing may occur. Other storage-management options are to use mmap to set aside (large) areas of virtual memory for each heap and suballocate each heap's storage within that area, pushing part of the storage management complexity back to the operating system.

Multiple heaps increase external fragmentation as the ratio of heaps to threads increases, which can lead to heap blowup. The external fragmentation experienced by a program with a single heap is now multiplied by the number of heaps, since each heap manages its own free storage and allocates its own reserved memory. Additionally, objects freed by one heap cannot be reused by other threads without increasing the cost of the memory operations, except indirectly by returning free memory to the operating system, which can be expensive. Depending

on how the operating system provides dynamic storage to an application, returning storage may be difficult or impossible, *e.g.*, the contiguous sbrk area in Unix. In the worst case, a program in which objects are allocated from one heap but deallocated to another heap means these freed objects are never reused.

Adding a *global heap* (G) attempts to reduce the cost of obtaining/returning memory among heaps (sharing) by buffering storage within the application address-space. Now, each heap obtains and returns storage to/from the global heap rather than the operating system. Storage is obtained from the global heap only when a heap allocation cannot be fulfilled, and returned to the global heap when a heap's free memory exceeds some threshold. Similarly, the global heap buffers this memory, obtaining and returning storage to/from the operating system as necessary. The global heap does not have its own thread and makes no internal allocation requests; instead, it uses the application thread, which called one of the multiple heaps and then the global heap, to perform operations. Hence, the worst-case cost of a memory operation includes all these steps. With respect to heap blowup, the global heap provides an indirect mechanism to move free memory among heaps, which usually has a much lower cost than interacting with the operating system to achieve the same goal and is independent of the mechanism used by the operating system to present dynamic memory to an address space.

However, since any thread may indirectly perform a memory operation on the global heap, it is a shared resource that requires locking. A single lock can be used to protect the global heap or fine-grained locking can be used to reduce contention. In general, the cost is minimal since the majority of memory operations are completed without the use of the global heap.

1:1 model (thread heaps) where each thread has its own heap eliminating most contention and locking because threads seldom access another thread's heap (see ownership in Section 2.5.2). An additional benefit of thread heaps is improved locality due to better memory layout. As each thread only allocates from its heap, all objects for a thread are consolidated in the storage area for that heap, better utilizing each CPUs cache and accessing fewer pages. In contrast, the T:H model spreads each thread's objects over a larger area in different heaps. Thread heaps can also eliminate allocator-induced active false-sharing, if memory is acquired so it does not overlap at crucial boundaries with memory for another thread's heap. For example, assume page boundaries coincide with cache line boundaries, if a thread heap always acquires pages of memory then no two threads share a page or cache line unless pointers are passed among them. Hence, allocator-induced active false-sharing in Figure 2.6(b), p. 10 cannot occur because the memory for thread heaps never overlaps.

When a thread terminates, there are two options for handling its thread heap. First is to free all objects in the thread heap to the global heap and destroy the thread heap. Second is to place the thread heap on a list of available heaps and reuse it for a new thread in the future. Destroying the thread heap immediately may reduce external fragmentation sooner, since all free objects are freed to the global heap and may be reused by other threads. Alternatively, reusing thread heaps may improve performance if the inheriting thread makes similar allocation requests as the thread that previously held the thread heap because any unfreed storage is immediately accessible.

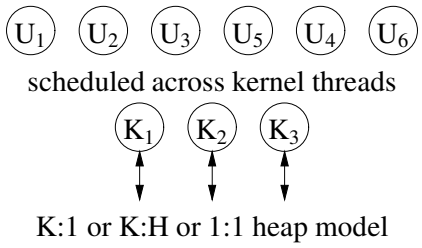


Figure 2.9: User-Level Kernel Heaps

2.5.1 User-Level Threading

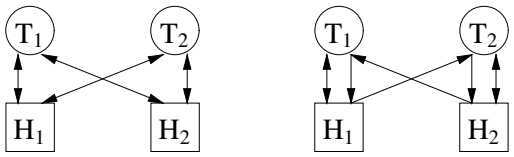
It is possible to use any of the heap models with user-level (M:N) threading. However, an important goal of user-level threading is for fast operations (creation/termination/context-switching) by not interacting with the operating system, which allows the ability to create large numbers of high-performance interacting threads ($>$ 10,000). It is difficult to retain this goal, if the user-threading model is directly involved with the heap model. Figure 2.9 shows that virtually all user-level threading systems use whatever kernel-level heap-model is provided by the language runtime. Hence, a user thread allocates/deallocates from/to the heap of the kernel thread on which it is currently executing.

Adopting this model results in a subtle problem with shared heaps. With kernel threading, an operation that is started by a kernel thread is always completed by that thread. For example, if a kernel thread starts an allocation/deallocation on a shared heap, it always completes that operation with that heap even if preempted, *i.e.*, any locking correctness associated with the shared heap is preserved across preemption.

However, this correctness property is not preserved for user-level threading. A user thread can start an allocation/deallocation on one kernel thread, be preempted (time slice), and continue running on a different kernel thread to complete the operation [10]. When the user thread continues on the new kernel thread, it may have pointers into the previous kernel-thread's heap and hold locks associated with it. To get the same kernel-thread safety, time slicing must be disabled/enabled around these operations, so the user thread cannot jump to another kernel thread. However, eagerly disabling/enabling time-slicing on the allocation/deallocation fast path is expensive, because preemption does not happen that frequently. Instead, techniques exist to lazily detect this case in the interrupt handler, abort the preemption, and return to the operation so it can complete atomically. Occasionally ignoring a preemption should be benign, but a persistent lack of preemption can result in both short and long term starvation.

2.5.2 Ownership

Ownership defines which heap an object is returned-to on deallocation. If a thread returns an object to the heap it was originally allocated from, a heap has ownership of its objects. Alternatively, a thread can return an object to the heap it is currently associated with, which can be any heap accessible during a thread's lifetime. Figure 2.10 shows an example of multiple heaps (minus the global heap) with and without ownership. Again, the arrows indicate the direction



(a) Ownership (b) No Ownership

Figure 2.10: Heap Ownership

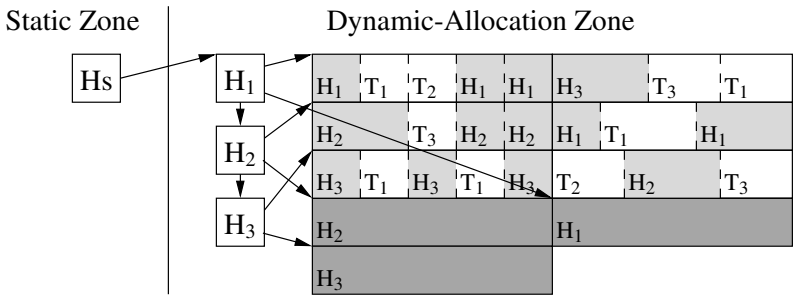
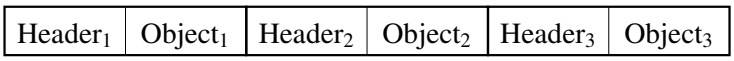


Figure 2.11: Multiple-Heap Storage with Ownership

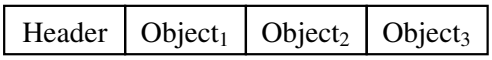
memory conceptually moves for each kind of operation. For the 1:1 thread:heap relationship, a thread only allocates from its own heap, and without ownership, a thread only frees objects to its own heap, which means the heap is private to its owner thread and does not require any locking, called a *private heap*. For the T:1/T:H models with or without ownership or the 1:1 model with ownership, a thread may free objects to different heaps, which makes each heap publicly accessible to all threads, called a *public heap*.

Figure 2.11 shows the effect of ownership on storage layout. (For simplicity, assume the heaps all use the same size of reserves storage.) In contrast to Figure 2.8, p. 12, each reserved area used by a heap only contains free storage for that particular heap because threads must return free objects back to the owner heap. Again, because multiple threads can allocate/free/reallocate adjacent storage in the same heap, all forms of false sharing may occur. The exception is for the 1:1 model if reserved memory does not overlap a cache-line because all allocated storage within a used area is associated with a single thread. In this case, there is no allocator-induced active false-sharing (see Figure 2.6(b), p. 10) because two adjacent allocated objects used by different threads cannot share a cache-line. As well, there is no allocator-induced passive false-sharing (see Figure 2.6(b), p. 10) because two adjacent allocated objects used by different threads cannot occur because free objects are returned to the owner heap.

The main advantage of ownership is preventing heap blowup by returning storage for reuse by the owner heap. Ownership prevents the classical problem where one thread performs allocations from one heap, passes the object to another thread, and the receiving thread deallocates the object to another heap, hence draining the initial heap of storage. As well, allocator-induced passive false-sharing is eliminated because returning an object to its owner heap means it can never be allocated to another thread. For example, in Figure 2.6(c), p. 10, the deallocation by Thread_2 returns Object_2 back to Thread_1's heap; hence a subsequent allocation by Thread_2 cannot return this storage. The disadvantage of ownership is deallocating to another thread's heap so heaps are



(a) Object Headers



(b) Object Container

Figure 2.12: Header Placement

no longer private and require locks to provide safe concurrent access.

Object ownership can be immediate or delayed, meaning free objects may be batched on a separate free list either by the returning or receiving thread. While the returning thread can batch objects, batching across multiple heaps is complex and there is no obvious time when to push back to the owner heap. It is better for returning threads to immediately return to the receiving thread's batch list as the receiving thread has better knowledge when to incorporate the batch list into its free pool. Batching leverages the fact that most allocation patterns use the contention-free fast-path, so locking on the batch list is rare for both the returning and receiving threads.

It is possible for heaps to steal objects rather than return them and then reallocate these objects again when storage runs out on a heap. However, stealing can result in passive false-sharing. For example, in Figure 2.6(c), p. 10, Object_2 may be deallocated to Thread_2's heap initially. If Thread_2 reallocates Object_2 before it is returned to its owner heap, then passive false-sharing may occur.

2.6 Object Containers

Bracketing every allocation with headers/trailers can result in significant internal fragmentation, as shown in Figure 2.12(a). Especially if the headers contain redundant management information, then storing that information is a waste of storage, *e.g.*, object size may be the same for many objects because programs only allocate a small set of object sizes. As well, it can result in poor cache usage, since only a portion of the cache line is holding useful information from the program's perspective. Spatial locality can also be negatively affected leading to poor cache locality [12]: while the header and object are together in memory, they are generally not accessed together; *e.g.*, the object is accessed by the program when it is allocated, while the header is accessed by the allocator when the object is free.

An alternative approach factors common header/trailer information to a separate location in memory and organizes associated free storage into blocks called *object containers* (*superblocks* in [2]), as in Figure 2.12(b). The header for the container holds information necessary for all objects in the container; a trailer may also be used at the end of the container. Similar to the approach described for thread heaps in Section 2.5, p. 11, if container boundaries do not overlap with memory of another container at crucial boundaries and all objects in a container are allocated to the same thread, allocator-induced active false-sharing is avoided.

The difficulty with object containers lies in finding the object header/trailer given only the object address, since that is normally the only information passed to the deallocation operation.

One way to do this is to start containers on aligned addresses in memory, then truncate the lower bits of the object address to obtain the header address (or round up and subtract the trailer size to obtain the trailer address). For example, if an object at address 0xFC28 EF08 is freed and containers are aligned on 64 KB (0x0001 0000) addresses, then the container header is at 0xFC28 0000.

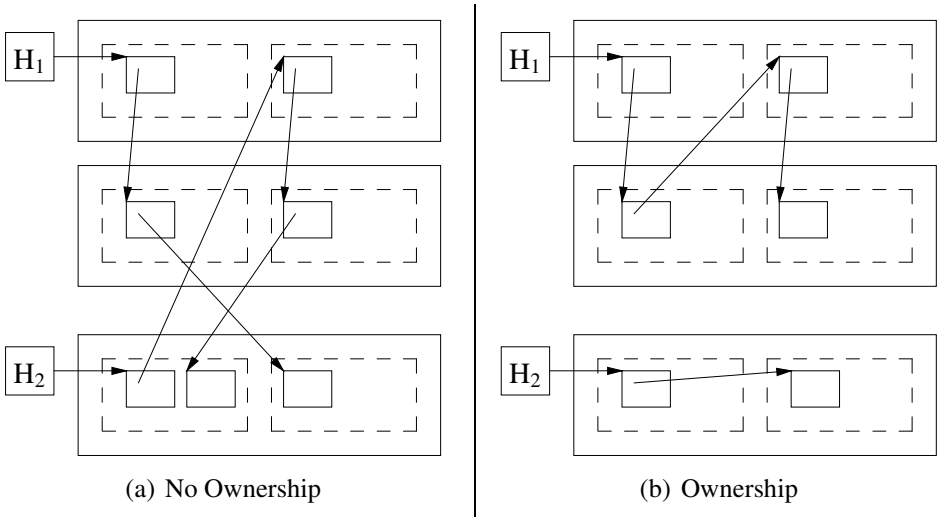
Normally, a container has homogeneous objects of fixed size, with fixed information in the header that applies to all container objects (*e.g.*, object size and ownership). This approach greatly reduces internal fragmentation since far fewer headers are required, and potentially increases spatial locality as a cache line or page holds more objects since the objects are closer together due to the lack of headers. However, although similar objects are close spatially within the same container, different sized objects are further apart in separate containers. Depending on the program, this may or may not improve locality. If the program uses several objects from a small number of containers in its working set, then locality is improved since fewer cache lines and pages are required. If the program uses many containers, there is poor locality, as both caching and paging increase. Another drawback is that external fragmentation may be increased since containers reserve space for objects that may never be allocated by the program, *i.e.*, there are often multiple containers for each size only partially full. However, external fragmentation can be reduced by using small containers.

Containers with heterogeneous objects implies different headers describing them, which complicates the problem of locating a specific header solely by an address. A couple of solutions can be used to implement containers with heterogeneous objects. However, the problem with allowing objects of different sizes is that the number of objects, and therefore headers, in a single container is unpredictable. One solution allocates headers at one end of the container, while allocating objects from the other end of the container; when the headers meet the objects, the container is full. Freed objects cannot be split or coalesced since this causes the number of headers to change. The difficulty in this strategy remains in finding the header for a specific object; in general, a search is necessary to find the object's header among the container headers. A second solution combines the use of container headers and individual object headers. Each object header stores the object's heterogeneous information, such as its size, while the container header stores the homogeneous information, such as the owner when using ownership. This approach allows containers to hold different types of objects, but does not completely separate headers from objects. The benefit of the container in this case is to reduce some redundant information that is factored into the container header.

In summary, object containers trade off internal fragmentation for external fragmentation by isolating common administration information to remove/reduce internal fragmentation, but at the cost of external fragmentation as some portion of a container may not be used and this portion is unusable for other kinds of allocations. A consequence of this tradeoff is its effect on spatial locality, which can produce positive or negative results depending on program access-patterns.

2.6.1 Container Ownership

Without ownership, objects in a container are deallocated to the heap currently associated with the thread that frees the object. Thus, different objects in a container may be on different heap



(a) No Ownership (b) Ownership

Figure 2.13: Free-list Structure with Container Ownership

free-lists (see Figure 2.13(a)). With ownership, all objects in a container belong to the same heap (see Figure 2.13(b)), so ownership of an object is determined by the container owner. If multiple threads can allocate/free/reallocate adjacent storage in the same heap, all forms of false sharing may occur. Only with the 1:1 model and ownership is active and passive false-sharing avoided (see Section 2.5.2, p. 14). Passive false-sharing may still occur, if delayed ownership is used. Finally, a completely free container can become reserved storage and be reset to allocate objects of a new size or freed to the global heap.

When a container changes ownership, the ownership of all objects within it change as well. Moving a container involves moving all objects on the heap's free-list in that container to the new owner. This approach can reduce contention for the global heap, since each request for objects from the global heap returns a container rather than individual objects.

Additional restrictions may be applied to the movement of containers to prevent active false-sharing. For example, in Figure 2.14(a), a container being used by Thread_1 changes ownership, through the global heap. In Figure 2.14(b), when Thread_2 allocates an object from the newly acquired container it is actively false-sharing even though no objects are passed among threads. Note, once the object is freed by Thread_1, no more false sharing can occur until the container changes ownership again. To prevent this form of false sharing, container movement may be restricted to when all objects in the container are free. One implementation approach that increases the freedom to return a free container to the operating system involves allocating containers using a call like mmap, which allows memory at an arbitrary address to be returned versus only storage at the end of the contiguous sbrk area, again pushing storage management complexity back to the operating system.

Using containers with ownership increases external fragmentation since a new container for a requested object size must be allocated separately for each thread requesting it. In Figure 2.15, using object ownership allocates 80% more space than without ownership.

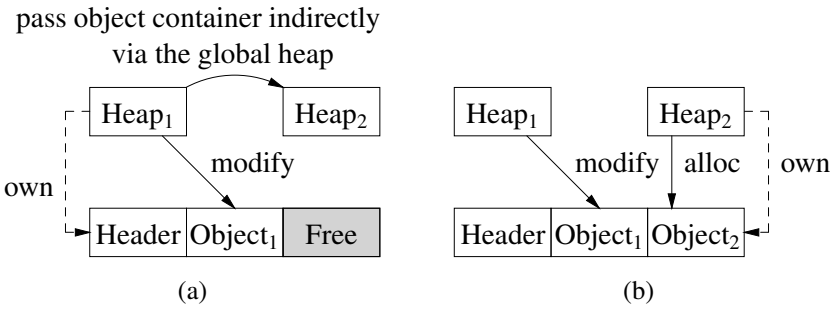


Figure 2.14: Active False-Sharing using Containers

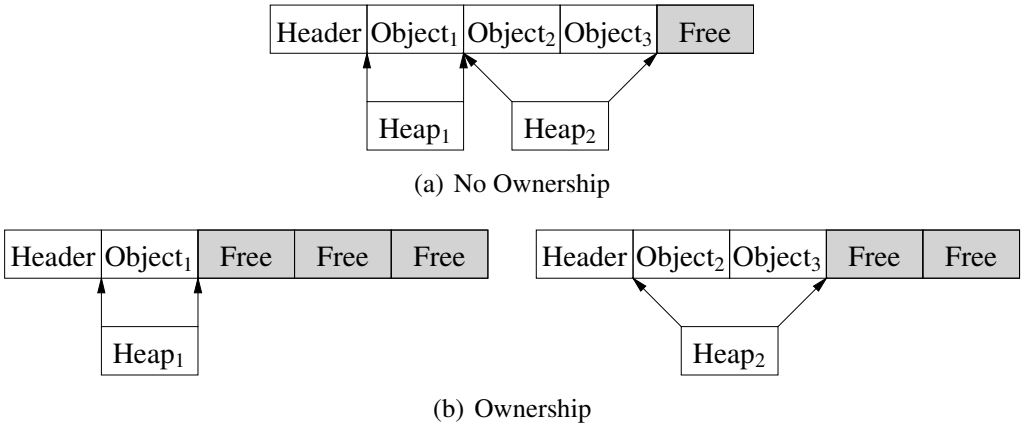


Figure 2.15: External Fragmentation with Container Ownership

2.6.2 Container Size

One way to control the external fragmentation caused by allocating a large container for a small number of requested objects is to vary the size of the container. As described earlier, container boundaries need to be aligned on addresses that are a power of two to allow easy location of the header (by truncating lower bits). Aligning containers in this manner also determines the size of the container. However, the size of the container has different implications for the allocator.

The larger the container, the fewer containers are needed, and hence, the fewer headers need to be maintained in memory, improving both internal fragmentation and potentially performance. However, with more objects in a container, there may be more objects that are unallocated, increasing external fragmentation. With smaller containers, not only are there more containers, but a second new problem arises where objects are larger than the container. In general, large objects, *e.g.*, greater than 64 KB, are allocated directly from the operating system and are returned immediately to the operating system to reduce long-term external fragmentation. If the container size is small, *e.g.*, 1 KB, then a 1.5 KB object is treated as a large object, which is likely to be inappropriate. Ideally, it is best to use smaller containers for smaller objects, and larger containers for medium objects, which leads to the issue of locating the container header.

In order to find the container header when using different sized containers, a super container is used (see Figure 2.16). The super container spans several containers, contains a header with information for finding each container header, and starts on an aligned address. Super-container

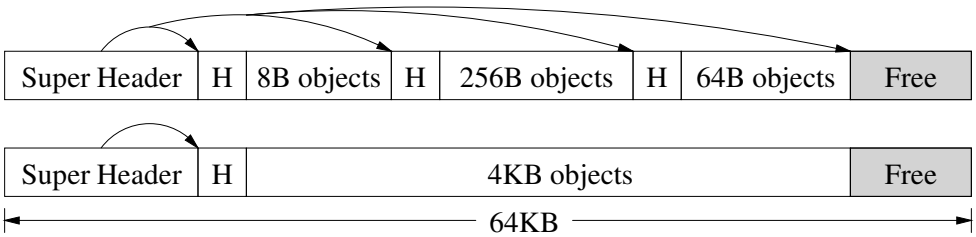


Figure 2.16: Super Containers

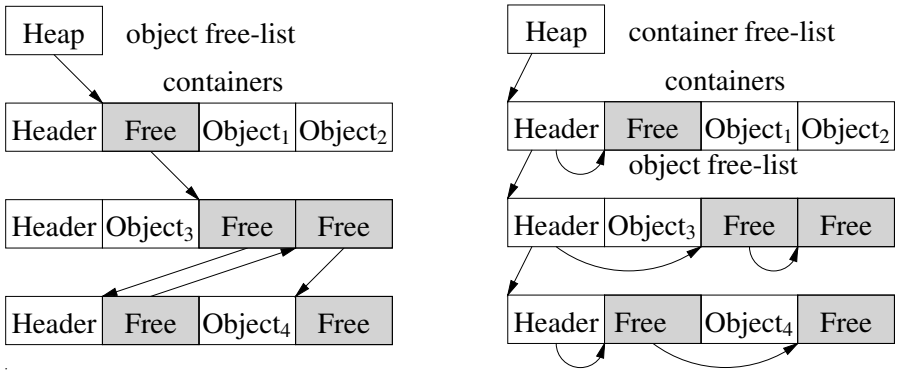
headers are found using the same method used to find container headers by dropping the lower bits of an object address. The containers within a super container may be different sizes or all the same size. If the containers in the super container are different sizes, then the super-container header must be searched to determine the specific container for an object given its address. If all containers in the super container are the same size, *e.g.*, 16KB, then a specific container header can be found by a simple calculation. The free space at the end of a super container is used to allocate new containers.

Minimal internal and external fragmentation is achieved by having as few containers as possible, each being as full as possible. It is also possible to achieve additional benefit by using larger containers for popular small sizes, as it reduces the number of containers with associated headers. However, this approach assumes it is possible for an allocator to determine in advance which sizes are popular. Keeping statistics on requested sizes allows the allocator to make a dynamic decision about which sizes are popular. For example, after receiving a number of allocation requests for a particular size, that size is considered a popular request size and larger containers are allocated for that size. If the decision is incorrect, larger containers than necessary are allocated that remain mostly unused. A programmer may be able to inform the allocator about popular object sizes, using a mechanism like mallopt, in order to select an appropriate container size for each object size.

2.6.3 Container Free-Lists

The container header allows an alternate approach for managing the heap's free-list. Rather than maintain a global free-list throughout the heap (see Figure 2.17(a)), the containers are linked through their headers and only the local free objects within a container are linked together (see Figure 2.17(b)). Note, maintaining free lists within a container assumes all free objects in the container are associated with the same heap; thus, this approach only applies to containers with ownership.

This alternate free-list approach can greatly reduce the complexity of moving all freed objects belonging to a container to another heap. To move a container using a global free-list, as in Figure 2.17(a), the free list is first searched to find all objects within the container. Each object is then removed from the free list and linked together to form a local free-list for the move to the new heap. With local free-lists in containers, as in Figure 2.17(b), the container is simply removed from one heap's free list and placed on the new heap's free list. Thus, when using local free-lists, the operation of moving containers is reduced from $O(N)$ to $O(1)$. However, there is



(a) Global Free-List Among Containers (b) Local Free-List Within Containers

Figure 2.17: Container Free-List Structure

the additional storage cost in the header, which increases the header size, and therefore internal fragmentation.

When all objects in the container are the same size, a single free-list is sufficient. However, when objects in the container are different size, the header needs a free list for each size class when using a binning allocation algorithm, which can be a significant increase in the container-header size. The alternative is to use a different allocation algorithm with a single free-list, such as a sequential-fit allocation-algorithm.

2.6.4 Hybrid Private/Public Heap

Section 2.5.2, p. 14 discusses advantages and disadvantages of public heaps (T:H model and with ownership) and private heaps (thread heaps with ownership). For thread heaps with ownership, it is possible to combine these approaches into a hybrid approach with both private and public heaps (see Figure 2.18). The main goal of the hybrid approach is to eliminate locking on thread-local allocation/deallocation, while providing ownership to prevent heap blowup. In the hybrid approach, a thread first allocates from its private heap and second from its public heap if no free memory exists in the private heap. Similarly, a thread first deallocates an object to its private heap, and second to the public heap. Both private and public heaps can allocate/deallocate to/from the global heap if there is no free memory or excess free memory, although an implementation may choose to funnel all interaction with the global heap through one of the heaps. Note, deallocation from the private to the public (dashed line) is unlikely because there is no obvious advantages unless the public heap provides the only interface to the global heap. Finally, when a thread frees an object it does not own, the object is either freed immediately to its owner's public heap or put in the freeing thread's private heap for delayed ownership, which allows the freeing thread to temporarily reuse an object before returning it to its owner or batch objects for an owner heap into a single return.

As mentioned, an implementation may have only one heap interact with the global heap, so the other heap can be simplified. For example, if only the private heap interacts with the global heap, the public heap can be reduced to a lock-protected free-list of objects deallocated by other threads due to ownership, called a *remote free-list*. To avoid heap blowup, the private heap

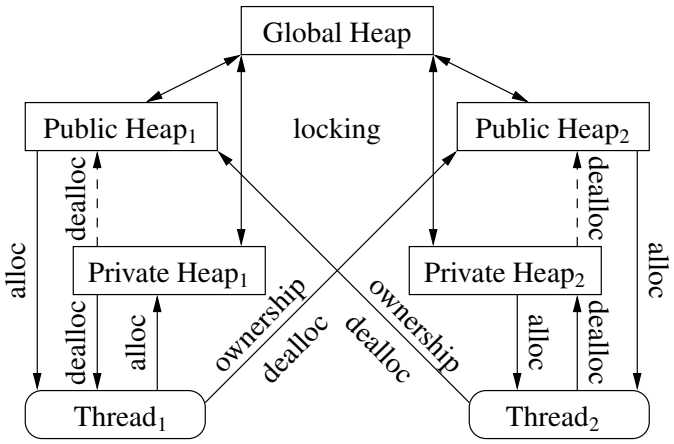


Figure 2.18: Hybrid Private/Public Heap for Per-thread Heaps

allocates from the remote free-list when it reaches some threshold or it has no free storage. Since the remote free-list is occasionally cleared during an allocation, this adds to that cost. Clearing the remote free-list is $O(1)$ if the list can simply be added to the end of the private-heap's free-list, or $O(N)$ if some action must be performed for each freed object.

If only the public heap interacts with other threads and the global heap, the private heap can handle thread-local allocations and deallocations without locking. In this scenario, the private heap must deallocate storage after reaching a certain threshold to the public heap (and then eventually to the global heap from the public heap) or heap blowup can occur. If the public heap does the major management, the private heap can be simplified to provide high-performance thread-local allocations and deallocations.

The main disadvantage of each thread having both a private and public heap is the complexity of managing two heaps and their interactions in an allocator. Interestingly, heap implementations often focus on either a private or public heap, giving the impression a single versus a hybrid approach is being used. In many case, the hybrid approach is actually being used, but the simpler heap is just folded into the complex heap, even though the operations logically belong in separate heaps. For example, a remote free-list is actually a simple public-heap, but may be implemented as an integral component of the complex private-heap in an allocator, masking the presence of a hybrid approach.

2.7 Allocation Buffer

An allocation buffer is reserved memory (see Section 2.1, p. 4) not yet allocated to the program, and is used for allocating objects when the free list is empty. That is, rather than requesting new storage for a single object, an entire buffer is requested from which multiple objects are allocated later. Any heap may use an allocation buffer, resulting in allocation from the buffer before requesting objects (containers) from the global heap or operating system, respectively. The allocation buffer reduces contention and the number of global/operating-system calls. For coalescing, a buffer is split into smaller objects by allocations, and recomposed into larger buffer areas during deallocations.

Allocation buffers are useful initially when there are no freed objects in a heap because many allocations usually occur when a thread starts (simple bump allocation). Furthermore, to prevent heap blowup, objects should be reused before allocating a new allocation buffer. Thus, allocation buffers are often allocated more frequently at program/thread start, and then allocations often diminish.

Using an allocation buffer with a thread heap avoids active false-sharing, since all objects in the allocation buffer are allocated to the same thread. For example, if all objects sharing a cache line come from the same allocation buffer, then these objects are allocated to the same thread, avoiding active false-sharing. Active false-sharing may still occur if objects are freed to the global heap and reused by another heap.

Allocation buffers may increase external fragmentation, since some memory in the allocation buffer may never be allocated. A smaller allocation buffer reduces the amount of external fragmentation, but increases the number of calls to the global heap or operating system. The allocation buffer also slightly increases internal fragmentation, since a pointer is necessary to locate the next free object in the buffer.

The unused part of a container, neither allocated or freed, is an allocation buffer. For example, when a container is created, rather than placing all objects within the container on the free list, the objects form an allocation buffer and are allocated from the buffer as allocation requests are made. This lazy method of constructing objects is beneficial in terms of paging and caching. For example, although an entire container, possibly spanning several pages, is allocated from the operating system, only a small part of the container is used in the working set of the allocator, reducing the number of pages and cache lines that are brought into higher levels of cache.

2.8 Lock-Free Operations

A *lock-free algorithm* guarantees safe concurrent-access to a data structure, so that at least one thread makes progress, but an individual thread has no execution bound and may starve [20, pp. 745–746]. (A *wait-free algorithm* puts a bound on the number of steps any thread takes to complete an operation to prevent starvation.) Lock-free operations can be used in an allocator to reduce or eliminate the use of locks. While locks and lock-free data-structures often have equal performance, lock-free has the advantage of not holding a lock across preemption so other threads can continue to make progress. With respect to the heap, these situations are unlikely unless all threads make extremely high use of dynamic-memory allocation, which can be an indication of poor design. Nevertheless, lock-free algorithms can reduce the number of context switches, since a thread does not yield/block while waiting for a lock; on the other hand, a thread may busy-wait for an unbounded period holding a processor. Finally, lock-free implementations have greater complexity and hardware dependency. Lock-free algorithms can be applied most easily to simple free-lists, *e.g.*, remote free-list, to allow lock-free insertion and removal from the head of a stack. Implementing lock-free operations for more complex data-structures (queue [44]/deque [43]) is correspondingly more complex. Michael [32] and Gidenstam *et al.* [14] have created lock-free variations of the Hoard allocator.

Chapter 3

Allocator

This chapter presents a new stand-alone concurrent low-latency memory-allocator (≈1,200 lines of code), called llheap (low-latency heap), for C/C++ programs using kernel threads (1:1 threading), and specialized versions of the allocator for the programming languages μC++ and C$\forall$ using user-level threads running over multiple kernel threads (M:N threading). The new allocator fulfills the GNU C Library allocator API [16].

3.1 llheap

The primary design objective for llheap is low-latency across all allocator calls independent of application access-patterns and/or number of threads, *i.e.*, very seldom does the allocator have a delay during an allocator call. (Large allocations requiring initialization, *e.g.*, zero fill, and/or copying are not covered by the low-latency objective.) A direct consequence of this objective is very simple or no storage coalescing; hence, llheap's design is willing to use more storage to lower latency. This objective is apropos because systems research and industrial applications are striving for low latency and computers have huge amounts of RAM memory. Finally, llheap's performance should be comparable with the current best allocators (see performance comparison in Chapter 5, p. 54).

3.2 Design Choices

llheap's design was reviewed and changed multiple times throughout the thesis. Some of the rejected designs are discussed because they show the path to the final design (see discussion in Section 2.5, p. 11). Note, a few simple tests for a design choice were compared with the current best allocators to determine the viability of a design.

3.2.1 Allocation Fastpath

These designs look at the allocation/free *fastpath*, *i.e.*, when an allocation can immediately return free storage or returned storage is not coalesced.

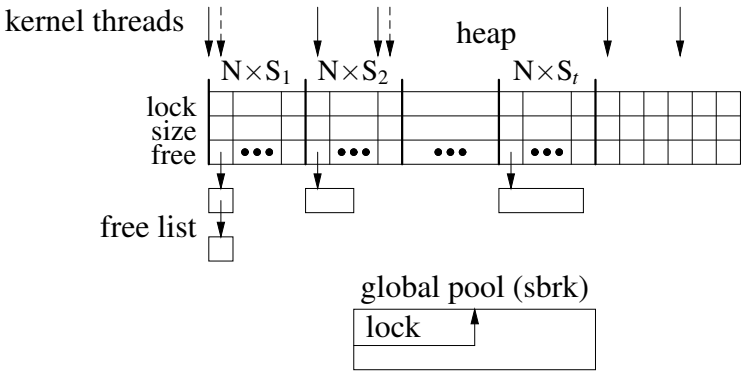


Figure 3.1: T:1 with Shared Buckets

T:1 model Figure 3.1 shows one heap accessed by multiple kernel threads (KTs) using a bucket array, where smaller bucket sizes are shared among N KTs. This design leverages the fact that usually the allocation requests are less than 1024 bytes and there are only a few different request sizes. When KTs $\leq$ N, the common bucket sizes are uncontented; when KTs $>$ N, the free buckets are contented and latency increases significantly. In all cases, a KT must acquire/release a lock, contented or uncontented, along the fast allocation path because a bucket is shared. Therefore, while threads are contending for a small number of buckets sizes, the buckets are distributed among them to reduce contention, which lowers latency; however, picking N is workload specific.

Problems:

- Need to know when a KT is created/destroyed to assign/unassign a shared bucket-number from the memory allocator.
- When no thread is assigned a bucket number, its free storage is unavailable.
- All KTs contend for the global-pool lock for initial allocations, before free-lists get populated.

Tests showed having locks along the allocation fast-path produced a significant increase in allocation costs and any contention among KTs produces a significant spike in latency.

T:H model Figure 3.2 shows a fixed number of heaps (N), each a local free pool, where the heaps are sharded (distributed) across the KTs. A KT can point directly to its assigned heap or indirectly through the corresponding heap bucket. When KT $\leq$ N, the heaps might be uncontented; when KTs $>$ N, the heaps are contented. In all cases, a KT must acquire/release a lock, contented or uncontented along the fast allocation path because a heap is shared. By increasing N, this approach reduces contention but increases storage (time versus space); however, picking N is workload specific.

Problems:

- Need to know when a KT is created/destroyed to assign/unassign a heap from the memory allocator.
- When no thread is assigned to a heap, its free storage is unavailable.
- Ownership issues arise (see Section 2.5.2, p. 14).

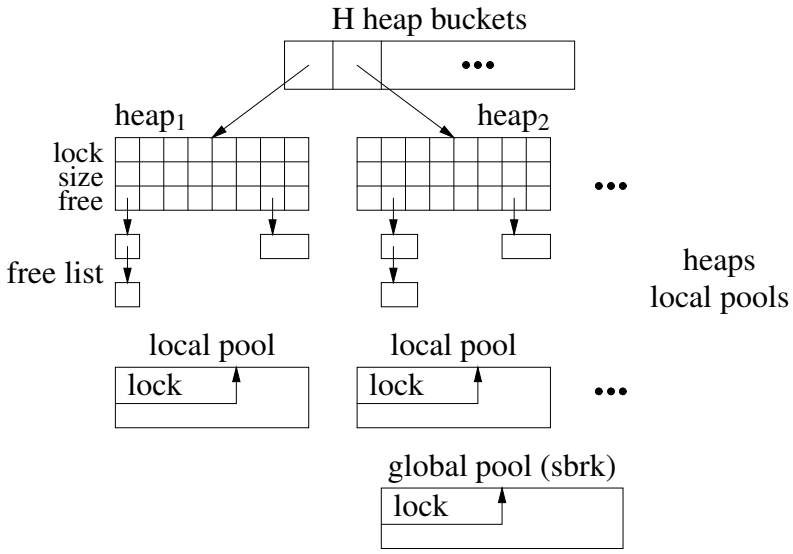


Figure 3.2: T:H with Shared Heaps

- All KTs contend for the local/global-pool lock for initial allocations, before free-lists get populated.

Tests showed having locks along the allocation fast-path produced a significant increase in allocation costs and any contention among KTs produces a significant spike in latency.

T:H model, H = number of CPUs This design is the T:H model but H is set to the number of CPUs on the computer or the number restricted to an application, *e.g.*, via taskset. (See Figure 3.2 but with a heap bucket per CPU.) Hence, each CPU logically has its own private heap and local pool. A memory operation is serviced from the heap associated with the CPU executing the operation. This approach removes fastpath locking and contention, regardless of the number of KTs mapped across the CPUs, because only one KT is running on each CPU at a time (modulo operations on the global pool and ownership). This approach is essentially an M:N approach where M is the number if KTs and N is the number of CPUs.

Problems:

- Need to know when a CPU is added/removed from the taskset.
- Need a fast way to determine the CPU a KT is executing on to access the appropriate heap.
- Need to prevent preemption during a dynamic memory operation because of the *serially-reusable problem*.

> A sequence of code that is guaranteed to run to completion before being invoked to accept another input is called serially-reusable code. [23]

If a KT is preempted during an allocation operation, the operating system can schedule another KT on the same CPU, which can begin an allocation operation before the previous operation associated with this CPU has completed, invalidating heap correctness. Note, the serially-reusable problem can occur in sequential programs with preemption, if the signal

handler calls the preempted function, unless the function is serially reusable. Essentially, the serially-reusable problem is a race condition on an unprotected critical section, where the operating system is providing the second thread via the signal handler.

Library librseq [8] was used to perform a fast determination of the CPU and to ensure all memory operations complete on one CPU using librseq's restartable sequences, which restart the critical section after undoing its writes, if the critical section is preempted.

Tests showed that librseq can determine the particular CPU quickly but setting up the restartable critical-section along the allocation fast-path produced a significant increase in allocation costs. Also, the number of undoable writes in librseq is limited and restartable sequences cannot deal with user-level thread (UT) migration across KTs. For example, UT_1 is executing a memory operation by KT_1 on CPU_1 and a time-slice preemption occurs. The signal handler context switches UT_1 onto the user-level ready-queue and starts running UT_2 on KT_1, which immediately calls a memory operation. Since KT_1 is still executing on CPU_1, librseq takes no action because it assumes KT_1 is still executing the same critical section. Then UT_1 is scheduled onto KT_2 by the user-level scheduler, and its memory operation continues in parallel with UT_2 using references into the heap associated with CPU_1, which corrupts CPU_1's heap. If librseq had an rseq_abort which:

1. Marked the current restartable critical-section as cancelled so it restarts when attempting to commit.
2. Do nothing if there is no current restartable critical section in progress.

Then rseq_abort could be called on the backside of a user-level context-switching. A feature similar to this idea might exist for hardware transactional-memory. A significant effort was made to make this approach work but its complexity, lack of robustness, and performance costs resulted in its rejection.

1:1 model This design is the T:H model with T = H, where there is one thread-local heap for each KT. (See Figure 3.2 but with a heap bucket per KT and no bucket or local-pool lock.) Hence, immediately after a KT starts, its heap is created and just before a KT terminates, its heap is (logically) deleted. Heaps are uncontended for a KTs memory operations as every KT has its own thread-local heap, modulo operations on the global pool and ownership.

Problems:

- Need to know when a KT starts/terminates to create/delete its heap.
  It is possible to leverage constructors/destructors for thread-local objects to get a general handle on when a KT starts/terminates.
- There is a classic *memory-reclamation* problem for ownership because storage passed to another thread can be returned to a terminated heap.
  The classic solution only deletes a heap after all referents are returned, which is complex. The cheap alternative is for heaps to persist for program duration to handle outstanding referent frees. If old referents return storage to a terminated heap, it is handled in the same way as an active heap. To prevent heap blowup, terminated heaps can be reused by new KTs, where a reused heap may be populated with free storage from a prior KT (external fragmentation). In

most cases, heap blowup is not a problem because programs have a small allocation set-size, so the free storage from a prior KT is apropos for a new KT.

- There can be significant external fragmentation as the number of KTs increases.
  In many concurrent applications, good performance is achieved with the number of KTs proportional to the number of CPUs. Since the number of CPUs is relatively small, and a heap is also relatively small, ≈10K bytes (not including any associated freed storage), the worst-case external fragmentation is still small compared to the RAM available on large servers with many CPUs.
- There is the same serially-reusable problem with UTs migrating across KTs.

Tests showed this design produced the closest performance match with the best current allocators, and code inspection showed most of these allocators use different variations of this approach.

The conclusion from this design exercise is: any atomic fence, atomic instruction (lock free), or lock along the allocation fastpath produces significant slowdown. For the T:1 and T:H models, locking must exist along the allocation fastpath because the buckets or heaps might be shared by multiple threads, even when KTs $\leq$ N. For the T:H=CPU and 1:1 models, locking is eliminated along the allocation fastpath. However, T:H=CPU has poor operating-system support to determine the CPU id (heap id) and prevent the serially-reusable problem for KTs. More operating system support is required to make this model viable, but there is still the serially-reusable problem with user-level threading. So the 1:1 model had no atomic actions along the fastpath and no special operating-system support requirements. The 1:1 model still has the serially-reusable problem with user-level threading, which is addressed in Section 3.5, p. 36, and the greatest potential for heap blowup for certain allocation patterns.

3.2.2 Allocation Latency

A primary goal of llheap is low latency. Two forms of latency are internal and external. Internal latency is the time to perform an allocation, while external latency is time to obtain/return storage from/to the operating system. Ideally latency is $O(1)$ with a small constant.

To obtain $O(1)$ internal latency means no searching on the allocation fastpath and largely prohibits coalescing, which leads to external fragmentation. The mitigating factor is that most programs have well behaved allocation patterns, where the majority of allocation operations can be $O(1)$, and heap blowup does not occur without coalescing (although the allocation footprint may be slightly larger).

To obtain $O(1)$ external latency means obtaining one large storage area from the operating system and subdividing it across all program allocations, which requires a good guess at the program storage high-watermark and potential large external fragmentation. Excluding real-time operating-systems, operating-system operations are unbounded, and hence some external latency is unavoidable. The mitigating factor is that operating-system calls can often be reduced if a programmer has a sense of the storage high-watermark and the allocator is capable of using this information (see malloc_expansion page 42). Furthermore, while operating-system calls are unbounded, many are now reasonably fast, so their latency is tolerable and infrequent.

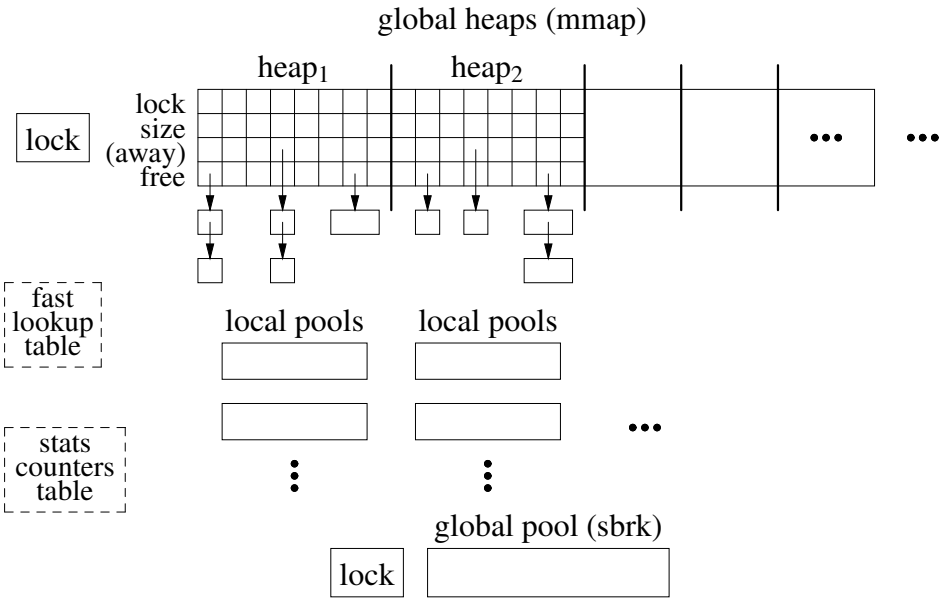


Figure 3.3: llheap Structure

3.3 llheap Structure

Figure 3.3 shows the design of llheap, which uses the following features:

- 1:1 multiple-heap model to minimize the fastpath,
- can be built with or without heap ownership,
- headers per allocation versus containers,
- no coalescing to minimize latency,
- global heap memory (pool) obtained from the operating system using mmap to create and reuse heaps needed by threads,
- local reserved memory (pool) per heap obtained from global pool,
- global reserved memory (pool) obtained from the operating system using sbrk call,
- optional fast-lookup table for converting allocation requests into bucket sizes,
- optional statistic-counters table for accumulating counts of allocation operations.

llheap starts by creating an array of N global heaps from storage obtained using mmap, where N is the number of computer cores, that persists for program duration. There is a global bump-pointer to the next free heap in the array. When this array is exhausted, another array of heaps is allocated. There is a global top pointer for a intrusive linked-list to chain free heaps from terminated threads. When statistics are turned on, there is a global top pointer for a intrusive linked-list to chain *all* the heaps, which is traversed to accumulate statistics counters across heaps using malloc_stats.

When a KT starts, a heap is allocated from the current array for exclusive use by the KT. When a KT terminates, its heap is chained onto the heap free-list for reuse by a new KT, which prevents unbounded growth of number of heaps. The free heaps are stored on stack so hot storage

is reused first. Preserving all heaps, created during the program lifetime, solves the storage lifetime problem when ownership is used. This approach wastes storage if a large number of KTs are created/terminated at program start and then the program continues sequentially. llheap can be configured with object ownership, where an object is freed to the heap from which it is allocated, or object no-ownership, where an object is freed to the KT's current heap.

Each heap uses segregated free-buckets that have free objects distributed across 91 different sizes from 16 to 4M. All objects in a bucket are of the same size. The number of buckets used is determined dynamically depending on the crossover point from sbrk to mmap allocation using mallopt(M_MMAP_THRESHOLD), *i.e.*, small objects managed by the program and large objects managed by the operating system. Each free bucket of a specific size has the following two lists:

- A free stack used solely by the KT heap-owner, so push/pop operations do not require locking. The free objects are a stack so hot storage is reused first.
- For ownership, a shared away-stack for KTs to return storage allocated by other KTs, so push/pop operations require locking. When the free stack is empty, the entire ownership stack is removed and becomes the head of the corresponding free stack.

Algorithm 1 shows the allocation outline for an object of size S. First, the allocation is divided into small (sbrk) or large (mmap). For large allocations, the storage is mapped directly from the operating system. For small allocations, S is quantized into a bucket size. Quantizing is performed using a binary search over the ordered bucket array. An optional optimization is fast lookup $O(1)$ for sizes < 64K from a 64K array of type **char**, where each element has an index to the corresponding bucket. The **char** type restricts the number of bucket sizes to 256. For $S > 64$K, a binary search is used. Then, the allocation storage is obtained from the following locations (in order), with increasing latency.

1. bucket's free stack,
2. bucket's away stack,
3. heap's local pool
4. global pool
5. operating system (sbrk)

Algorithm 2 shows the de-allocation (free) outline for an object at address A with ownership. First, the address is divided into small (sbrk) or large (mmap). For large allocations, the storage is unmapped back to the operating system. For small allocations, the bucket associated with the request size is retrieved. If the bucket is local to the thread, the allocation is pushed onto the thread's associated bucket. If the bucket is not local to the thread, the allocation is pushed onto the owning thread's associated away stack.

Algorithm 3 shows the de-allocation (free) outline for an object at address A without ownership. The algorithm is the same as for ownership except if the bucket is not local to the thread. Then the corresponding bucket of the owner thread is computed for the deallocating thread, and the allocation is pushed onto the deallocating thread's bucket.

Finally, the llheap design funnels all allocation/deallocation operations through the malloc and free routines, which are the only routines to directly access and manage the internal data structures of the heap. Other allocation operations, *e.g.*, calloc, memalign, and realloc, are composed of calls to malloc and possibly free, and may manipulate header information after

Algorithm 1 Dynamic object allocation of size S

```
1: O ← NULL
2: if S >= mmap-threshhold then
3:     O ← allocate dynamic memory using system call mmap with size S
4: else
5:     B ← smallest free-bucket ≥ S
6:     if B's free-list is empty then
7:         if B's away-list is empty then
8:             if heap's allocation buffer < S then
9:                 get allocation from global pool (which might call sbrk)
10:            end if
11:            O ← bump allocate an object of size S from allocation buffer
12:        else
13:            merge B's away-list into free-list
14:            O ← pop an object from B's free-list
15:        end if
16:    else
17:        O ← pop an object from B's free-list
18:    end if
19:    O's owner ← B
20: end if
21: return O
```

Algorithm 2 Dynamic object free at address A with object ownership

```
1: if A mapped allocation then
2:     return A's dynamic memory to system using system call munmap
3: else
4:     B ← O's owner
5:     if B is thread-local heap's bucket then
6:         push A to B's free-list
7:     else
8:         push A to B's away-list
9:     end if
10: end if
```

Algorithm 3 Dynamic object free at address A without object ownership

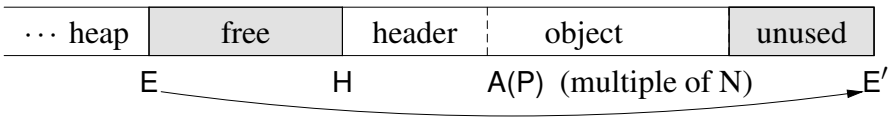
```
1: if A mapped allocation then
2:     return A's dynamic memory to system using system call munmap
3: else
4:     B ← O's owner
5:     if B is thread-local heap's bucket then
6:         push A to B's free-list
7:     else
8:         C ← thread local heap's bucket with same size as B
9:         push A to C's free-list
10:    end if
11: end if
```

storage is allocated. This design simplifies heap-management code during development and maintenance.

3.3.1 Alignment

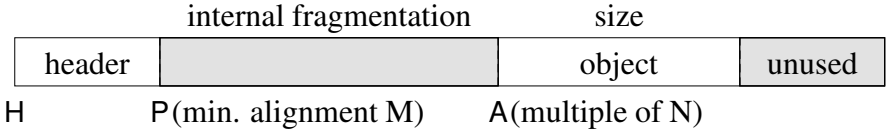
Most dynamic memory allocations have a minimum storage alignment for the contained object(s). Often the minimum memory alignment, M, is the bus width (32 or 64-bit) or the largest register (double, long double) or largest atomic instruction (DCAS) or vector data (MMMX). In general, the minimum storage alignment is 8/16-byte boundary on 32/64-bit computers. For consistency, the object header is normally aligned at this same boundary. Larger alignments must be a power of 2, such as page alignment (4/8K). Any alignment request, N, $\leq$ the minimum alignment is handled as a normal allocation with minimal alignment.

For alignments greater than the minimum, the obvious approach for aligning to address A is: compute the next address that is a multiple of N after the current end of the heap, E, plus room for the header before A and the size of the allocation after A, moving the end of the heap to E′.



The storage between E and H is chained onto the appropriate free list for future allocations. The same approach is used for sufficiently large free blocks, where E is the start of the free block, and any unused storage before H or after the allocated object becomes free storage. In this approach, the aligned address A is the same as the allocated storage address P, *i.e.*, P = A for all allocation routines, which simplifies deallocation. However, if there are a large number of aligned requests, this approach leads to memory fragmentation from the small free areas around the aligned object. As well, it does not work for large allocations, where many memory allocators switch from program sbrk to operating-system mmap. The reason is that mmap only starts on a page boundary, and it is difficult to reuse the storage before the alignment boundary for other requests. Finally, this approach is incompatible with allocator designs that funnel allocation requests through malloc as it directly manipulates management information within the allocator to optimize the space/time of a request.

Instead, llheap alignment is accomplished by making a *pessimistic* allocation request for sufficient storage to ensure that *both* the alignment and size request are satisfied, *e.g.*:

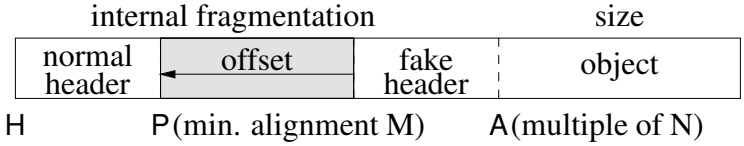


The amount of storage necessary is alignment – M + size, which ensures there is an address, A, after the storage returned from malloc, P, that is a multiple of alignment followed by sufficient storage for the data object. The approach is pessimistic because if P already has the correct alignment N, the initial allocation has already requested sufficient space to move to the next

multiple of N. For this special case, there is alignment – M bytes of unused storage after the data object, which subsequently can be used by realloc.

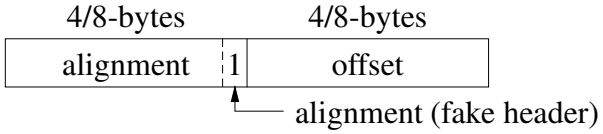
Note, the address returned is A, which is subsequently returned to free. However, to correctly free the allocated object, the value P must be computable, since that is the value generated by malloc and returned within memalign. Hence, there must be a mechanism to detect when P $\neq$ A and how to compute P from A.

The llheap approach uses two headers: the *original* header associated with a memory allocation from malloc, and a *fake* header within this storage before the alignment boundary A, which is returned from memalign, e.g.:



Since malloc has a minimum alignment of M, P $\neq$ A only holds for alignments greater than M. When P $\neq$ A, the minimum distance between P and A is M bytes, due to the pessimistic storage-allocation. Therefore, there is always room for an M-byte fake header before A.

The fake header must supply an indicator to distinguish it from a normal header and the location of address P generated by malloc. This information is encoded as an offset from A to P and the initialize alignment (discussed in Section 3.3.2). To distinguish a fake header from a normal header, the least-significant bit of the alignment is used because the offset participates in multiple calculations, while the alignment is just remembered data.



3.3.2 realloc and Sticky Properties

The allocation routine realloc provides a memory-management pattern for shrinking/enlarging an existing allocation, while maintaining some or all of the object data, rather than performing the following steps manually.

realloc pattern	**manually**
`T * naddr = realloc( oaddr, newSize );`	`T * naddr = (T *)malloc( newSize );` *// new storage* `memcpy( naddr, addr, oldSize );` *// copy old bytes* `free( addr );` *// free old storage* `addr = naddr;` *// change pointer*

The realloc pattern leverages available storage at the end of an allocation due to bucket sizes, possibly eliminating a new allocation and copying. This pattern is not used enough to reduce

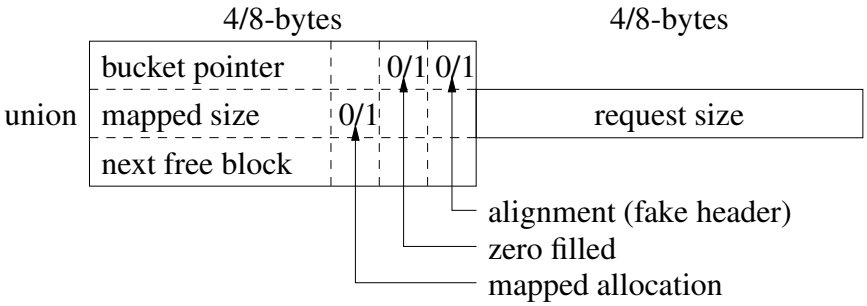


Figure 3.4: llheap Normal Header

storage management costs. In fact, if oaddr is nullptr, realloc does a malloc, so even the initial malloc can be a realloc for consistency in the allocation pattern.

The hidden problem for this pattern is the effect of zero fill and alignment with respect to reallocation. Are these properties transient or persistent ("sticky")? For example, when memory is initially allocated by calloc or memalign with zero fill or alignment properties, respectively, what happens when those allocations are given to realloc to change size? That is, if realloc logically extends storage into unused bucket space or allocates new storage to satisfy a size change, are initial allocation properties preserved? Currently, allocation properties are not preserved, so subsequent use of realloc storage may cause inefficient execution or errors due to lack of zero fill or alignment. This silent problem is unintuitive to programmers and difficult to locate because it is transient. To prevent these problems, llheap preserves initial allocation properties for the lifetime of an allocation and the semantics of realloc are augmented to preserve these properties, with additional query routines. This change makes the realloc pattern efficient and safe.

3.3.3 Header

To preserve allocation properties requires storing additional information with an allocation, The best available option is the header, where Figure 3.4 shows the llheap storage layout. The header has two data field sized appropriately for 32/64-bit alignment requirements. The first field is a union of three values:

bucket pointer is for allocated storage and points back to the bucket associated with this storage requests (see Figure 3.3, p. 29 for the fields accessible in a bucket).

mapped size is for mapped storage and is the storage size for use in unmapping.

next free block is for free storage and is an intrusive pointer chaining same-size free blocks onto a bucket's free stack.

The second field remembers the request size versus the allocation (bucket) size, *e.g.*, request 42 bytes which is rounded up to 64 bytes. Since programmers think in request sizes rather than allocation sizes, the request size allows better generation of statistics or errors and also helps in memory management.

The low-order 3-bits of the first field are *unused* for any stored values as these values are 16-byte aligned by default, whereas the second field may use all of its bits. The 3 unused bits are

```
Heap statistics: (storage request / allocation)
  malloc >0 calls 2,766; 0 calls 2,064; storage 12,715 / 13,367 bytes
  aalloc >0 calls 0; 0 calls 0; storage 0 / 0 bytes
  calloc >0 calls 6; 0 calls 0; storage 1,008 / 1,104 bytes
  memalign >0 calls 0; 0 calls 0; storage 0 / 0 bytes
  amemalign >0 calls 0; 0 calls 0; storage 0 / 0 bytes
  cmemalign >0 calls 0; 0 calls 0; storage 0 / 0 bytes
  resize >0 calls 0; 0 calls 0; storage 0 / 0 bytes
  realloc >0 calls 0; 0 calls 0; storage 0 / 0 bytes
  free !null calls 2,766; null calls 4,064; storage 12,715 / 13,367 bytes
  away pulls 0; pushes 0; storage 0 / 0 bytes
  sbrk calls 1; storage 10,485,760 bytes
  mmap calls 10,000; storage 10,000 / 10,035 bytes
  munmap calls 10,000; storage 10,000 / 10,035 bytes
  threads started 4; exited 3
  heaps new 4; reused 0
```

Figure 3.5: Statistics Output

used to represent mapped allocation, zero filled, and alignment, respectively. Note, the alignment bit is not used in the normal header and the zero-filled/mapped bits are not used in the fake header. This implementation allows a fast test if any of the lower 3-bits are on (& and compare). If no bits are on, it implies a basic allocation, which is handled quickly; otherwise, the bits are analysed and appropriate actions are taken for the complex cases. Since most allocations are basic, they will take significantly less time as the memory operations will be done along the allocation and free fastpath.

3.4 Statistics and Debugging

llheap can be built to accumulate fast and largely contention-free allocation statistics to help understand allocation behaviour. Incrementing statistic counters must appear on the allocation fastpath. As noted, any atomic operation along the fastpath produces a significant increase in allocation costs. To make statistics performant enough for use on running systems, each heap has its own set of statistic counters, so heap operations do not require atomic operations.

To locate all statistic counters, heaps are linked together in statistics mode, and this list is locked and traversed to sum all counters across heaps. Note, the list is locked to prevent errors traversing an active list; the statistics counters are not locked and can flicker during accumulation. Figure 3.5 shows an example of statistics output, which covers all allocation operations and information about deallocating storage not owned by a thread. No other memory allocator studied provides as comprehensive statistical information. Finally, these statistics were invaluable during the development of this thesis for debugging and verifying correctness and should be equally valuable to application developers.

llheap can also be built with debug checking, which inserts many asserts along all allocation paths. These assertions detect incorrect allocation usage, like double frees, unfreed storage, or memory corruptions because internal values (like header fields) are overwritten. These checks

are best effort as opposed to complete allocation checking as in valgrind. Nevertheless, the checks detect many allocation problems. There is an unfortunate problem in detecting unfreed storage because some library routines assume their allocations have life-time duration, and hence, do not free their storage. For example, printf allocates a 1024-byte buffer on the first call and never deletes this buffer. To prevent a false positive for unfreed storage, it is possible to specify an amount of storage that is never freed (see malloc_unfreed page 42), and it is subtracted from the total allocate/free difference. Determining the amount of never-freed storage is annoying, but once done, any warnings of unfreed storage are application related.

Tests indicate only a 30% performance decrease when statistics *and* debugging are enabled, and the latency cost for accumulating statistic is mitigated by limited calls, often only one at the end of the program.

3.5 User-level Threading Support

The serially-reusable problem (see page 26) occurs for kernel threads in the "T:H model, H = number of CPUs" model and for user threads in the "1:1" model, where llheap uses the "1:1" model. The solution is to prevent interrupts that can result in a CPU or KT change during operations that are logically critical sections such as starting a memory operation on one KT and completing it on another. Locking these critical sections negates any attempt for a quick fastpath and results in high contention. For user-level threading, the serially-reusable problem appears with time slicing for preemptable scheduling, as the signal handler context switches to another user-level thread. Without time slicing, a user thread performing a long computation can prevent the execution of (starve) other threads. To prevent starvation for a memory-allocation-intensive thread, *i.e.*, the time slice always triggers in an allocation critical-section for one thread so the thread never gets time sliced, a thread-local *rollforward* flag is set in the signal handler when it aborts a time slice. The rollforward flag is tested at the end of each allocation funnel routine (see page 30), and if set, it is reset and a volunteer yield (context switch) is performed to allow other threads to execute.

llheap uses two techniques to detect when execution is in an allocation operation or routine called from allocation operation, to abort any time slice during this period. On the slowpath when executing expensive operations, like sbrk or mmap, interrupts are disabled/enabled by setting kernel-thread-local flags so the signal handler aborts immediately. On the fastpath, disabling/enabling interrupts is too expensive as accessing kernel-thread-local storage can be expensive and not user-thread-safe. For example, the ARM processor stores the thread-local pointer in a coprocessor register that cannot perform atomic base-displacement addressing. Hence, there is a window between loading the kernel-thread-local pointer from the coprocessor register into a normal register and adding the displacement when a time slice can move a thread.

The fast technique (with lower run time cost) is to define a special code section and places all non-interruptible routines in this section. The linker places all code in this section into a contiguous block of memory, but the order of routines within the block is unspecified. Then, the signal handler compares the program counter at the point of interrupt with the the start and end address of the non-interruptible section, and aborts if executing within this section and sets the rollforward flag. This technique is fragile because any calls in the non-interruptible code outside

of the non-interruptible section (like sbrk) must be bracketed with disable/enable interrupts and these calls must be along the slowpath. Hence, for correctness, this approach requires inspection of generated assembler code for routines placed in the non-interruptible section. This issue is mitigated by the llheap funnel design so only funnel routines and a few statistics routines are placed in the non-interruptible section and their assembler code examined. These techniques are used in both the μC++ and C$\forall$ versions of llheap as both of these systems have user-level threading.

3.6 Bootstrapping

There are problems bootstrapping a memory allocator.

1. Programs can be statically or dynamically linked.
2. The order in which the linker schedules startup code is poorly supported so it cannot be controlled entirely.
3. Knowing a KT's start and end independently from the KT code is difficult.

For static linking, the allocator is loaded with the program. Hence, allocation calls immediately invoke the allocator operation defined by the loaded allocation library and there is only one memory allocator used in the program. This approach allows allocator substitution by placing an allocation library before any other in the linked/load path.

Allocator substitution is similar for dynamic linking, but the problem is that the dynamic loader starts first and needs to perform dynamic allocations *before* the substitution allocator is loaded. As a result, the dynamic loader uses a default allocator until the substitution allocator is loaded, after which all allocation operations are handled by the substitution allocator, including from the dynamic loader. Hence, some part of the sbrk area may be used by the default allocator and statistics about allocation operations cannot be correct. Furthermore, dynamic linking goes through trampolines, so there is an additional cost along the allocator fastpath for all allocation operations. Testing showed up to a 5% performance decrease with dynamic linking as compared to static linking, even when using tls_model("initial-exec") so the dynamic loader can obtain tighter binding.

All allocator libraries need to perform startup code to initialize data structures, such as the heap array for llheap. The problem is getting initialization done before the first allocator call. However, there does not seem to be mechanism to tell either the static or dynamic loader to first perform initialization code before any calls to a loaded library. Also, initialization code of other libraries and the run-time environment may call memory allocation routines such as malloc. This compounds the situation as there is no mechanism to tell either the static or dynamic loader to first perform the initialization code of the memory allocator before any other initialization that may involve a dynamic memory allocation call. As a result, calls to allocation routines occur without initialization. To deal with this problem, it is necessary to put a conditional initialization check along the allocation fastpath to trigger initialization (singleton pattern).

Two other important execution points are program startup and termination, which include prologue or epilogue code to bootstrap a program, which programmers are unaware of. For

example, dynamic-memory allocations before/after the application starts should not be considered in statistics because the application does not make these calls. llheap establishes these two points using routines:

```
__attribute__(( constructor( 100 ) )) static void startup( void ) {
    // clear statistic counters
    // reset allocUnfreed counter
}
__attribute__(( destructor( 100 ) )) static void shutdown( void ) {
    // sum allocUnfreed for all heaps
    // subtract global unfreed storage
    // if allocUnfreed > 0 then print warning message
}
```

which use global constructor/destructor priority 100, where the linker calls these routines at program prologue/epilogue in increasing/decreasing order of priority. Application programs may only use global constructor/destructor priorities greater than 100. Hence, startup is called after the program prologue but before the application starts, and shutdown is called after the program terminates but before the program epilogue. By resetting counters in startup, prologue allocations are ignored, and checking unfreed storage in shutdown checks only application memory management, ignoring the program epilogue.

While startup/shutdown apply to the program KT, a concurrent program creates additional KTs that do not trigger these routines. However, it is essential for the allocator to know when each KT is started/terminated. One approach is to create a thread-local object with a construct/destructor, which is triggered after a new KT starts and before it terminates, respectively.

```
struct ThreadManager {
    volatile bool pgm_thread;
    ThreadManager() {} // unusable
    ~ThreadManager() { if ( pgm_thread ) heapManagerDtor(); }
};
static thread_local ThreadManager threadManager;
```

Unfortunately, thread-local variables are created lazily, *i.e.*, on the first dereference of threadManager, which then triggers its constructor. Therefore, the constructor is useless for knowing when a KT starts because the KT must reference it, and the allocator does not control the application KT. Fortunately, the singleton pattern needed for initializing the program KT also triggers KT allocator initialization, which can then reference pgm_thread to call threadManager's constructor, otherwise its destructor is not called. Now when a KT terminates, ~ThreadManager is called to chain it onto the global-heap free-stack, where pgm_thread is set to true only for the program KT. The conditional destructor call prevents closing down the program heap, which must remain available because epilogue code may free more storage.

Finally, there is a recursive problem when the singleton pattern dereferences pgm_thread to initialize the thread-local object, because its initialization calls atExit, which immediately calls malloc to obtain storage. This recursion is handled with another thread-local flag to prevent double initialization. A similar problem exists when the KT terminates and calls member ~ThreadManager, because immediately afterwards, the terminating KT calls free to deallocate the storage obtained from the atExit. In the meantime, the terminated heap has been put on the

global-heap free-stack, and may be active by a new KT, so the atExit free is handled as a free to another heap and put onto the away list using locking.

For user threading systems, the KTs are controlled by the runtime, and hence, start/end pointers are known and interact directly with the llheap allocator for μC++ and C$\forall$, which eliminates or simplifies several of these problems. The following API was created to provide interaction between the language runtime and the allocator.

```
void startThread();                 // KT starts
void finishThread();                // KT ends
void startup();                     // when application code starts
void shutdown();                    // when application code ends
bool traceHeap();                   // enable allocation/free printing for debugging
bool traceHeapOn();                 // start printing allocation/free calls
bool traceHeapOff();                // stop printing allocation/free calls
```

This kind of API is necessary to allow concurrent runtime systems to interact with different memory allocators in a consistent way.

3.7 Added Features and Methods

The C dynamic-allocation API (see Figure 3.6) is neither orthogonal nor complete. For example,

- It is possible to zero fill or align an allocation but not both.
- It is *only* possible to zero fill an array allocation.
- It is not possible to resize a memory allocation without data copying.
- realloc does not preserve initial allocation properties.

As a result, programmers must provide these options, which is error prone, resulting in blaming the entire programming language for a poor dynamic-allocation API. Furthermore, newer programming languages have better type systems that can provide safer and more powerful APIs for memory allocation.

The following presents design and API changes for C, C++ (μC++), and C$\forall$, all of which are implemented in llheap.

3.7.1 Out of Memory

Most allocators use nullptr to indicate an allocation failure, specifically out of memory; hence the need to return an alternate value for a zero-sized allocation. A different approach allowed by C API is to abort a program when out of memory and return nullptr for a zero-sized allocation. In theory, notifying the programmer of memory failure allows recovery; in practice, it is almost impossible to gracefully recover when out of memory. Hence, the cheaper approach of returning nullptr for a zero-sized allocation is chosen because no pseudo allocation is necessary.

```
void * malloc( size_t size );
void * calloc( size_t nmemb, size_t size );
void * realloc( void * ptr, size_t size );
void * reallocarray( void * ptr, size_t nmemb, size_t size );
void free( void * ptr );
void * memalign( size_t alignment, size_t size );
void * aligned_alloc( size_t alignment, size_t size );
int posix_memalign( void ** memptr, size_t alignment, size_t size );
void * valloc( size_t size );
void * pvalloc( size_t size );

struct mallinfo mallinfo( void );
int mallopt( int param, int val );
int malloc_trim( size_t pad );
size_t malloc_usable_size( void * ptr );
void malloc_stats( void );
int malloc_info( int options, FILE * fp );
```

Figure 3.6: C Dynamic-Allocation API

3.7.2 C Interface

For C, it is possible to increase functionality and orthogonality of the dynamic-memory API to make allocation better for programmers.

For existing C allocation routines:

- calloc sets the sticky zero-fill property.
- memalign, aligned_alloc, posix_memalign, valloc and pvalloc set the sticky alignment property.
- realloc and reallocarray preserve sticky properties.

The C dynamic-memory API is extended with the following routines:

void * aalloc(size_t dim, size_t elemSize) extends calloc for allocating a dynamic array of objects without calculating the total size of array explicitly but *without* zero-filling the memory. aalloc is significantly faster than calloc, which is the only alternative given by the standard memory-allocation routines.

Usage aalloc takes two parameters.

- dim: number of array objects
- elemSize: size of array object

It returns the address of the dynamic array or NULL if either dim or elemSize are zero.

void * resize(**void** * oaddr, size_t size) extends realloc for resizing an existing allocation *without* copying previous data into the new allocation or preserving sticky properties. resize is significantly faster than realloc, which is the only alternative.

Usage resize takes two parameters.

- oaddr: address to be resized
- size: new allocation size (smaller or larger than previous)

It returns the address of the old or new storage with the specified new size or NULL if size is zero.

void * amemalign(size_t alignment, size_t dim, size_t elemSize) extends aalloc and memalign for allocating an aligned dynamic array of objects. Sets sticky alignment property.

Usage amemalign takes three parameters.

- alignment: alignment requirement
- dim: number of array objects
- elemSize: size of array object

It returns the address of the aligned dynamic-array or NULL if either dim or elemSize are zero.

void * cmemalign(size_t alignment, size_t dim, size_t elemSize) extends amemalign with zero fill and has the same usage as amemalign. Sets sticky zero-fill and alignment property. It returns the address of the aligned, zero-filled dynamic-array or NULL if either dim or elemSize are zero.

size_t malloc_alignment(**void** * addr) returns the alignment of the dynamic object for use in aligning similar allocations.

Usage malloc_alignment takes one parameter.

- addr: address of an allocated object.

It returns the alignment of the given object, where objects not allocated with alignment return the minimal allocation alignment.

bool malloc_zero_fill(**void** * addr) returns true if the object has the zero-fill sticky property for use in zero filling similar allocations.

Usage malloc_zero_fill takes one parameters.

- addr: address of an allocated object.

It returns true if the zero-fill sticky property is set and false otherwise.

size_t malloc_size(**void** * addr) returns the request size of the dynamic object (updated when an object is resized) for use in similar allocations. See also malloc_usable_size.

Usage malloc_size takes one parameters.

- addr: address of an allocated object.

It returns the request size or zero if addr is NULL.

int malloc_stats_fd(**int** fd) changes the file descriptor where malloc_stats writes statistics (default stdout).

Usage malloc_stats_fd takes one parameters.

- fd: file descriptor.

It returns the previous file descriptor.

size_t malloc_expansion() set the amount (bytes) to extend the heap when there is insufficient free storage to service an allocation request. It returns the heap extension size used throughout a program when requesting more memory from the system using sbrk system-call, *i.e.*, called once at heap initialization.

size_t malloc_mmap_start() set the crossover between allocations occurring in the sbrk area or separately mapped. It returns the crossover point used throughout a program, *i.e.*, called once at heap initialization.

size_t malloc_unfreed() amount subtracted to adjust for unfreed program storage (debug only). It returns the new subtraction amount and called by malloc_stats.

3.7.3 C++ Interface

The following extensions take advantage of overload polymorphism in the C++ type-system.

void * resize(**void** * oaddr, size_t nalign, size_t size) extends resize with an alignment requirement.

Usage takes three parameters.

- oaddr: address to be resized
- nalign: alignment requirement
- size: new allocation size (smaller or larger than previous)

It returns the address of the old or new storage with the specified new size and alignment, or NULL if size is zero.

void * realloc(**void** * oaddr, size_t nalign, size_t size) extends realloc with an alignment requirement and has the same usage as aligned resize.

3.7.4 C∀ Interface

The following extensions take advantage of overload polymorphism in the C∀ type-system. The key safety advantage of the C∀ type system is using the return type to select overloads; hence, a polymorphic routine knows the returned type and its size. This capability is used to remove the object size parameter and correctly cast the return storage to match the result type. For example, the following is the C∀ wrapper for C malloc:

```
T * malloc( void );
T * aalloc( size_t dim );
T * calloc( size_t dim );
T * resize( T * ptr, size_t size );
T * realloc( T * ptr, size_t size );
T * memalign( size_t align );
T * amemalign( size_t align, size_t dim );
T * cmemalign( size_t align, size_t dim  );
T * aligned_alloc( size_t align );
int posix_memalign( T ** ptr, size_t align );
T * valloc( void );
T * pvalloc( void );
```

Figure 3.7: C∀ C-Style Dynamic-Allocation API

```
forall( T & | sized(T) ) {
    T * malloc( void ) {
        if ( _Alignof(T) <= libAlign() ) return (T *)malloc( sizeof(T) ); // C allocation
        else return (T *)memalign( _Alignof(T), sizeof(T) ); // C allocation
    } // malloc
```

and is used as follows:

```
int * i = malloc();
double * d = malloc();
struct Spinlock { ... } __attribute__(( aligned(128) ));
Spinlock * sl = malloc();
```

where each malloc call provides the return type as T, which is used with **sizeof**, **_Alignof**, and casting the storage to the correct type. This interface removes many of the common allocation errors in C programs. Figure 3.7 show the C∀ wrappers for the equivalent C/C++ allocation routines with same semantic behaviour.

In addition to the C∀ C-style allocator interface, a new allocator interface is provided to further increase orthogonality and usability of dynamic-memory allocation. This interface helps programmers in three ways.

- naming: C∀ regular and **ttype** polymorphism (**ttype** polymorphism in C∀ is similar to C++ variadic templates) is used to encapsulate a wide range of allocation functionality into a single routine name, so programmers do not have to remember multiple routine names for different kinds of dynamic allocations.
- named arguments: individual allocation properties are specified using postfix function call, so the programmers do not have to remember parameter positions in allocation calls.
- object size: like the C∀'s C-interface, programmers do not have to specify object size or cast allocation results.

Note, postfix function call is an alternative call syntax, using backtick `, where the argument appears before the function name, *e.g.*,

```
duration ?`h( int h );      // ? denote the position of the function operand
```

```
duration ?`m( int m );
duration ?`s( int s );
duration dur = 3`h + 42`m + 17`s;
```

T * alloc(...) **or** T * alloc(size_t dim, ...) is overloaded with a variable number of specific allocation operations, or an integer dimension parameter followed by a variable number of specific allocation operations. These allocation operations can be passed as named arguments when calling the alloc routine. A call without parameters returns a dynamically allocated object of type T (malloc). A call with only the dimension (dim) parameter returns a dynamically allocated array of objects of type T (aalloc). The variable number of arguments consist of allocation properties, which can be combined to produce different kinds of allocations. The only restriction is for properties realloc and resize, which cannot be combined.

The allocation property functions are:

T_align ?`align(size_t alignment) to align the allocation. The alignment parameter must be $\geq$ the default alignment (libAlign() in C∀) and a power of two, *e.g.*:

```
int * i0 = alloc( 4096`align );  sout | i0 | nl;
int * i1 = alloc( 3, 4096`align );  sout | i1; for (i; 3 ) sout | &i1[i]; sout | nl;

0x555555572000
0x555555574000 0x555555574000 0x555555574004 0x555555574008
```

returns a dynamic object and object array aligned on a 4096-byte boundary.

S_fill(T) ?`fill (/* *various types* */) to initialize storage. There are three ways to fill storage:

1. A char fills each byte of each object.
2. An object of the returned type fills each object.
3. An object array pointer fills some or all of the corresponding object array.

For example:

```
1  int * i0 = alloc( 0n`fill );  sout | *i0 | nl;  // disambiguate 0
2  int * i1 = alloc( 5`fill );  sout | *i1 | nl;
3  int * i2 = alloc( '\xfe' `fill ); sout | hex( *i2 ) | nl;
4  int * i3 = alloc( 5, 5`fill );  for ( i; 5 ) sout | i3[i]; sout | nl;
5  int * i4 = alloc( 5, 0xdeadbeefN`fill );  for ( i; 5 ) sout | hex( i4[i] ); sout | nl;
6  int * i5 = alloc( 5, i3`fill );  for ( i; 5 ) sout | i5[i]; sout | nl;
7  int * i6 = alloc( 5, [i3, 3]`fill );  for ( i; 5 ) sout | i6[i]; sout | nl;
```

```
1  0
2  5
3  0xfefefefe
4  5 5 5 5 5
5  0xdeadbeef 0xdeadbeef 0xdeadbeef 0xdeadbeef 0xdeadbeef
6  5 5 5 5 5
7  5 5 5 –555819298 –555819298  // two undefined values
```

Examples 1 to 3 fill an object with a value or characters. Examples 4 to 7 fill an array of objects with values, another array, or part of an array.

S_resize(T) ?`resize(**void** * oaddr) used to resize, realign, and fill, where the old object data is not copied to the new object. The old object type may be different from the new object type, since the values are not used. For example:

```
1 int * i = alloc( 5`fill );  sout | i | *i;
2 i = alloc( i`resize, 256`align, 7`fill );  sout | i | *i;
3 double * d = alloc( i`resize, 4096`align, 13.5`fill );  sout | d | *d;
```

```
1 0x55555556d5c0 5
2 0x555555570000 7
3 0x555555571000 13.5
```

Examples 2 to 3 change the alignment, fill, and size for the initial storage of i.

```
1 int * ia = alloc( 5, 5`fill );  for ( i; 5 ) sout | ia[i]; sout | nl;
2 ia = alloc( 10, ia`resize, 7`fill ); for ( i; 10 ) sout | ia[i]; sout | nl;
3 sout | ia; ia = alloc( 5, ia`resize, 512`align, 13`fill ); sout | ia; for ( i; 5 ) sout | ia[i]; sout | nl;;
4 ia = alloc( 3, ia`resize, 4096`align, 2`fill );  sout | ia; for ( i; 3 ) sout | &ia[i] | ia[i]; sout | nl;
```

```
1 5 5 5 5 5
2 7 7 7 7 7 7 7 7 7
3 0x55555556d560 0x555555571a00 13 13 13 13 13
4 0x555555572000 0x555555572000 2 0x555555572004 2 0x555555572008 2
```

Examples 2 to 4 change the array size, alignment and fill for the initial storage of ia.

S_realloc(T) ?`realloc(T * a)) used to resize, realign, and fill, where the old object data is copied to the new object. The old object type must be the same as the new object type, since the value is used. Note, for fill, only the extra space after copying the data from the old object is filled with the given parameter. For example:

```
1 int * i = alloc( 5`fill );  sout | i | *i;
2 i = alloc( i`realloc, 256`align );  sout | i | *i;
3 i = alloc( i`realloc, 4096`align, 13`fill );  sout | i | *i;
```

```
1 0x55555556d5c0 5
2 0x555555570000 5
3 0x555555571000 5
```

Examples 2 to 3 change the alignment for the initial storage of i. The 13`fill in example 3 does nothing because no extra space is added.

```
1 int * ia = alloc( 5, 5`fill );  for ( i; 5 ) sout | ia[i]; sout | nl;
2 ia = alloc( 10, ia`realloc, 7`fill ); for ( i; 10 ) sout | ia[i]; sout | nl;
3 sout | ia; ia = alloc( 1, ia`realloc, 512`align, 13`fill ); sout | ia; for ( i; 1 ) sout | ia[i]; sout | nl;;
4 ia = alloc( 3, ia`realloc, 4096`align, 2`fill );  sout | ia; for ( i; 3 ) sout | &ia[i] | ia[i]; sout | nl;
```

```
1 5 5 5 5 5
2 5 5 5 5 5 7 7 7 7 7
3 0x55555556c560 0x555555570a00 5
4 0x555555571000 0x555555571000 5 0x555555571004 2 0x555555571008 2
```

Examples 2 to 4 change the array size, alignment and fill for the initial storage of ia. The 13`fill in example 3 does nothing because no extra space is added.

These C∀ allocation features are used extensively in the development of the C∀ runtime.

Chapter 4

Benchmarks

There are two basic approaches for evaluating computer software: benchmarks and micro-benchmarks.

Benchmarks are a suite of application programs (SPEC CPU/WEB) that are exercised in a common way (inputs) to find differences among underlying software implementations associated with an application (compiler, memory allocator, web server, *etc*.). The applications are supposed to represent common execution patterns that need to perform well with respect to an underlying software implementation. Benchmarks are often criticized for having overlapping patterns, insufficient patterns, or extraneous code that masks patterns.

Micro-Benchmarks attempt to extract the common execution patterns associated with an application and run the pattern independently. This approach removes any masking from extraneous application code, allows execution pattern to be very precise, and provides an opportunity for the execution pattern to have multiple independent tuning adjustments (knobs). Micro-benchmarks are often criticized for inadequately representing real-world applications.

While some crucial software components have standard benchmarks, no standard benchmark exists for testing and comparing memory allocators. In the past, an assortment of applications have been used for benchmarking allocators [9, 2, 3, 4]: P2C, GS, Espresso/Espresso-2, CFRAC/CFRAC-2, GMake, GCC, Perl/Perl-2, Gawk/Gawk-2, XPDF/XPDF-2, ROBOOP, Lindsay. As well, an assortment of micro-benchmark have been used for benchmarking allocators [27, 2, 40]: threadtest, shbench, Larson, consume, false sharing. Many of these benchmark applications and micro-benchmarks are old and may not reflect current application allocation patterns.

This thesis designs and examines a new set of micro-benchmarks for memory allocators that test a variety of allocation patterns, each with multiple tuning parameters. The aim of the micro-benchmark suite is to create a set of programs that can evaluate a memory allocator based on the key performance metrics such as speed, memory overhead, and cache performance. These programs give details of an allocator's memory overhead and speed under certain allocation patterns. The allocation patterns are configurable (adjustment knobs) to observe an allocator's performance across a spectrum allocation patterns, which is seldom possible with benchmark programs. Each micro-benchmark program has multiple control knobs specified by command-line arguments.

The new micro-benchmark suite measures performance by allocating dynamic objects and measuring specific metrics. An allocator's speed is benchmarked in different ways, as are issues like false sharing.

4.1 Prior Multi-Threaded Micro-Benchmarks

Modern memory allocators, such as llheap, must handle multi-threaded programs at the KT and UT level. The following multi-threaded micro-benchmarks are presented to give a sense of prior work [2] at the KT level. None of the prior work addresses multi-threading at the UT level.

4.1.1 threadtest

This benchmark stresses the ability of the allocator to handle different threads allocating and deallocating independently. There is no interaction among threads, *i.e.*, no object sharing. Each thread repeatedly allocates 100,000 *8-byte* objects then deallocates them in the order they were allocated. The execution time of the benchmark evaluates its efficiency.

4.1.2 shbench

This benchmark is similar to threadtest but each thread randomly allocate and free a number of *random-sized* objects. It is a stress test that also uses runtime to determine efficiency of the allocator.

4.1.3 Larson

This benchmark simulates a server environment. Multiple threads are created where each thread allocates and frees a number of random-sized objects within a size range. Before the thread terminates, it passes its array of 10,000 objects to a new child thread to continue the process. The number of thread generations varies depending on the thread speed. It calculates memory operations per second as an indicator of the memory allocator's performance.

4.2 New Multi-Threaded Micro-Benchmarks

The following new benchmarks were created to assess multi-threaded programs at the KT and UT level. For generating random values, two generators are supported: uniform [30] and fisher [29].

4.2.1 Churn Benchmark

The churn benchmark measures the runtime speed of an allocator in a multi-threaded scenario, where each thread extensively allocates and frees dynamic memory. Only malloc and free are used to eliminate any extra cost, such as memcpy in calloc or realloc. Churn simulates a memory intensive program and can be tuned to create different scenarios.

```
Main Thread
    create worker threads
    note time T1
    ...
    note time T2
    churn_speed = (T2 – T1)
Worker Thread
    initialize variables
    ...
    for ( N )
        R = random spot in array
        free R
        allocate new object at R
```

Figure 4.1: Churn Benchmark

Figure 4.1 shows the pseudo code for the churn micro-benchmark. This benchmark creates a buffer with M spots and an allocation in each spot, and then starts K threads. Each thread picks a random spot in M, frees the object currently at that spot, and allocates a new object for that spot. Each thread repeats this cycle N times. The main thread measures the total time taken for the whole benchmark and that time is used to evaluate the memory allocator's performance.

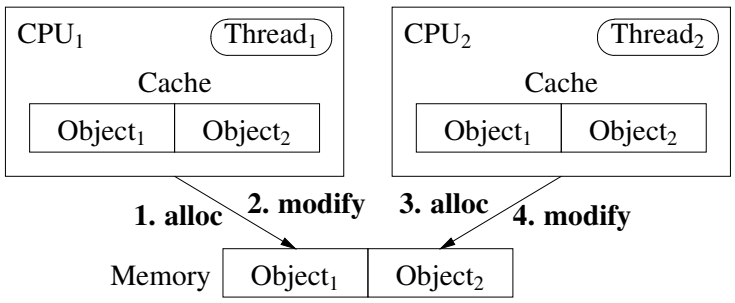
The adjustment knobs for churn are:

thread: number of threads (K).
spots: number of spots for churn (M).
obj: number of objects per thread (N).
max: maximum object size.
min: minimum object size.
step: object size increment.
distro: object size distribution

4.2.2 Cache Thrash

The cache-thrash micro-benchmark measures allocator-induced active false-sharing as illustrated in Section 2.3.2, p. 9. If memory is allocated for multiple threads on the same cache line, this can significantly slow down program performance. When threads share a cache line, frequent reads/writes to their cache-line object causes cache misses, which cause escalating delays as cache distance increases.

Cache thrash tries to create a scenario that leads to false sharing, if the underlying memory allocator is allocating dynamic memory to multiple threads on the same cache lines. Ideally, a memory allocator should distance the dynamic memory region of one thread from another. Having multiple threads allocating small objects simultaneously can cause a memory allocator to allocate objects on the same cache line, if its not distancing the memory among different threads.




```
Main Thread
    create worker threads
    ...
    signal workers to allocate
    ...
    signal workers to free
    ...
Worker Thread1
    warm up memory in chunks of 16 bytes
    ...
    For N
        malloc an object
        read/write the object M times
        free the object
    ...
Worker Thread2
    // same as Worker Thread1
```

Figure 4.2: Allocator-Induced Active False-Sharing Benchmark

Figure 4.2 shows the pseudo code for the cache-thrash micro-benchmark. First, it creates K worker threads. Each worker thread allocates an object and intensively reads/writes it for M times to possible invalidate cache lines that may interfere with other threads sharing the same cache line. Each thread repeats this for N times. The main thread measures the total time taken for all worker threads to complete. Worker threads sharing cache lines with each other are expected to take longer.

The adjustment knobs for cache access scenarios are:

thread: number of threads (K).
iterations: iterations of cache benchmark (N).
cacheRW: repetitions of reads/writes to object (M).
size: object size.

4.2.3 Cache Scratch

The cache-scratch micro-benchmark measures allocator-induced passive false-sharing as illustrated in Section 2.3.2, p. 9. As with cache thrash, if memory is allocated for multiple threads on the same cache line, this can significantly slow down program performance. In this scenario,

the false sharing is being caused by the memory allocator although it is started by the program sharing an object.

Cache scratch tries to create a scenario that leads to false sharing and should make the memory allocator preserve the program-induced false sharing, if it does not return a freed object to its owner thread and, instead, re-uses it instantly. An allocator using object ownership, as described in section Section 2.5.2, p. 14, is less susceptible to allocator-induced passive false-sharing. If the object is returned to the thread that owns it, then the new object that the thread gets is less likely to be on the same cache line.

Figure 4.3 shows the pseudo code for the cache-scratch micro-benchmark. First, it allocates K dynamic objects together, one for each of the K worker threads, possibly causing memory allocator to allocate these objects on the same cache line. Then it create K worker threads and passes an object from the K allocated objects to each of the K threads. Each worker thread frees the object passed by the main thread. Then, it allocates an object and reads/writes it repetitively for M times possibly causing frequent cache invalidations. Each worker repeats this N times.

Each thread allocating an object after freeing the original object passed by the main thread should cause the memory allocator to return the same object that was initially allocated by the main thread if the allocator did not return the initial object back to its owner (main thread). Then, intensive read/write on the shared cache line by multiple threads should slow down worker threads due to to high cache invalidations and misses. Main thread measures the total time taken for all the workers to complete.

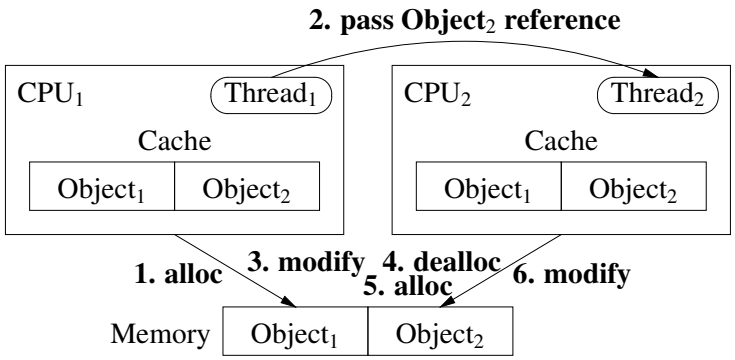
Similar to benchmark cache thrash in section Section 4.2.2, p. 48, different cache access scenarios can be created using the following command-line arguments.

threads: number of threads (K).
iterations: iterations of cache benchmark (N).
cacheRW: repetitions of reads/writes to object (M).
size: object size.

4.2.4 Speed Micro-Benchmark

The speed benchmark measures the runtime speed of individual and sequences of memory allocation routines:

1. malloc
2. realloc
3. free
4. calloc
5. malloc-free
6. realloc-free
7. calloc-free
8. malloc-realloc
9. calloc-realloc
10. malloc-realloc-free
11. calloc-realloc-free
12. malloc-realloc-free-calloc




```
Main Thread
    malloc N objects for each worker thread
    create worker threads and pass N objects to each worker
    ...
    signal workers to allocate
    ...
    signal workers to free
    ...
Worker Thread1
    warmup memory in chunks of 16 bytes
    ...
    free the object passed by the Main Thread
    For N
        malloc new object
        read/write the object M times
        free the object
    ...
Worker Thread2
    // same as Worker Thread1
```

Figure 4.3: Program-Induced Passive False-Sharing Benchmark

Figure 4.4 shows the pseudo code for the speed micro-benchmark. Each routine in the chain is called for N objects and then those allocated objects are used when calling the next routine in the allocation chain. This tests the latency of the memory allocator when multiple routines are chained together, *e.g.*, the call sequence malloc-realloc-free-calloc gives a complete picture of the major allocation routines when combined together. For each chain, the time is recorded to visualize performance of a memory allocator against each chain.

The adjustment knobs for memory usage are:

max: maximum object size.
min: minimum object size.
step: object size increment.
distro: object size distribution.
objects: number of objects per thread.
workers: number of worker threads.

```
Main Thread
    create worker threads
    foreach ( allocation chain )
        note time T1
        ...
        note time T2
        chain_speed = (T2 – T1) / number–of–worker–threads * N )
Worker Thread
    initialize variables
    ...
    foreach ( routine in allocation chain )
        call routine N times
```

Figure 4.4: Speed Benchmark

```
Main Thread
    print memory snapshot
    create producer threads
Producer Thread (K)
    set free start
    create consumer threads
    for ( N )
        allocate memory
        print memory snapshot
Consumer Thread (M)
    wait while ( allocations < free start )
    for ( N )
        free memory
        print memory snapshot
```

Figure 4.5: Memory Footprint Micro-Benchmark

4.2.5 Memory Micro-Benchmark

The memory micro-benchmark measures the memory overhead of an allocator. It allocates a number of dynamic objects and reads /proc/self/proc/maps to get the total memory requested by the allocator from the OS. It calculates the memory overhead by computing the difference between the memory the allocator requests from the OS and the memory that the program allocates. This micro-benchmark is like Larson and stresses the ability of an allocator to deal with object sharing.

Figure 4.5 shows the pseudo code for the memory micro-benchmark. It creates a producer-consumer scenario with K producer threads and each producer has M consumer threads. A producer has a separate buffer for each consumer and allocates N objects of random sizes following a configurable distribution for each consumer. A consumer frees these objects. After every memory operation, program memory usage is recorded throughout the runtime. This data is used to visualize the memory usage and consumption for the program.

The global adjustment knobs for this micro-benchmark are:

producer (K): sets the number of producer threads.
consumer (M): sets number of consumers threads for each producer.
round: sets production and consumption round size.

The adjustment knobs for object allocation are:

max: maximum object size.
min: minimum object size.
step: object size increment.
distro: object size distribution.
objects (N): number of objects per thread.

Chapter 5

Performance

This chapter uses the micro-benchmarks from Chapter 4, p. 46 to test a number of current memory allocators, including llheap. The goal is to see if llheap is competitive with the currently popular memory allocators.

5.1 Machine Specification

The performance experiments were run on two different multi-core architectures (x64 and ARM) to determine if there is consistency across platforms:

- **Algol** Huawei ARM TaiShan 2280 V2 Kunpeng 920, 24-core socket $\times$ 4, 2.6 GHz, GCC version 9.4.0
- **Nasus** AMD EPYC 7662, 64-core socket $\times$ 2, 2.0 GHz, GCC version 9.3.0

5.2 Existing Memory Allocators

With dynamic allocation being an important feature of C, there are many stand-alone memory allocators that have been designed for different purposes. For this thesis, 7 of the most popular and widely used memory allocators were selected for comparison, along with llheap.

llheap (llh) is the thread-safe allocator from Chapter 3, p. 24
Version: 1.0 **Configuration:** Compiled with dynamic linking, but without statistics or debugging.
Compilation command: `make`

glibc (glc) [34] is the default glibc thread-safe allocator.
Version: Ubuntu GLIBC 2.31-0ubuntu9.7 2.31
Configuration: Compiled by Ubuntu 20.04.
Compilation command: N/A

dlmalloc (dl) [28] is a thread-safe allocator that is single threaded and single heap. It maintains free-lists of different sizes to store freed dynamic memory.
Version: 2.8.6
Configuration: Compiled with preprocessor USE_LOCKS.
Compilation command: gcc -g3 -O3 -Wall -Wextra -fno-builtin-malloc -fno-builtin-calloc -fno-builtin-realloc -fno-builtin-free -fPIC -shared -DUSE_LOCKS -o libdlmalloc.so malloc-2.8.6.c

hoard (hrd) [1] is a thread-safe allocator that is multi-threaded and uses a heap layer framework. It has per-thread heaps that have thread-local free-lists, and a global shared heap.
Version: 3.13
Configuration: Compiled with hoard's default configurations and Makefile.
Compilation command: make all

jemalloc (je) [36] is a thread-safe allocator that uses multiple arenas. Each thread is assigned an arena. Each arena has chunks that contain contagious memory regions of same size. An arena has multiple chunks that contain regions of multiple sizes.
Version: 5.2.1
Configuration: Compiled with jemalloc's default configurations and Makefile.
Compilation command: autogen.sh; configure; make; make install

ptmalloc3 (pt3) [15] is a modification of dlmalloc. It is a thread-safe multi-threaded memory allocator that uses multiple heaps. ptmalloc3 heap has similar design to dlmalloc's heap.
Version: 1.8
Configuration: Compiled with ptmalloc3's Makefile using option "linux-shared".
Compilation command: make linux-shared

rpmalloc (rp) [24] is a thread-safe allocator that is multi-threaded and uses per-thread heap. Each heap has multiple size-classes and each size-class contains memory regions of the relevant size.
Version: 1.4.1
Configuration: Compiled with rpmalloc's default configurations and ninja build system.
Compilation command: python3 configure.py; ninja

tbb malloc (tbb) [35] is a thread-safe allocator that is multi-threaded and uses a private heap for each thread. Each private-heap has multiple bins of different sizes. Each bin contains free regions of the same size.
Version: intel tbb 2020 update 2, tbb_interface_version == 11102
Configuration: Compiled with tbbmalloc's default configurations and Makefile.
Compilation command: make

5.3 Experiments

Each micro-benchmark is configured and run with each of the allocators, The less time an allocator takes to complete a benchmark the better so lower in the graphs is better, except for the

Memory micro-benchmark graphs. All graphs use log scale on the Y-axis, except for the Memory micro-benchmark (see Section 5.3.5, p. 74).

5.3.1 Churn Micro-Benchmark

Churn tests allocators for speed under intensive dynamic memory usage (see Section 4.2.1, p. 47). This experiment was run with following configurations:

thread: 1, 2, 4, 8, 16, 32, 48
spots: 16
obj: 100,000
max: 500
min: 50
step: 50
distro: fisher

Figure 5.1 shows the results for algol and nasus. The X-axis shows the number of threads; the Y-axis shows the total experiment time. Each allocator's performance for each thread is shown in different colors.

Assessment All allocators did well in this micro-benchmark, except for dl on the ARM. dl's is the slowest, indicating some small bottleneck with respect to the other allocators. je is the fastest, with only a small benefit over the other allocators.

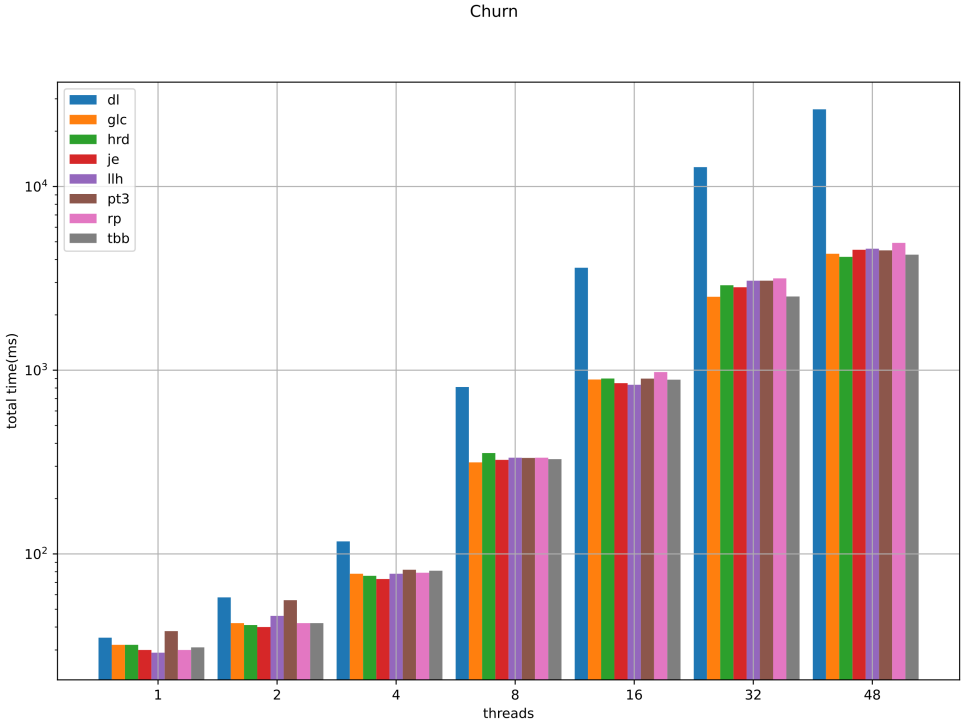
5.3.2 Cache Thrash

Thrash tests memory allocators for active false sharing (see Section 4.2.2, p. 48). This experiment was run with following configurations:

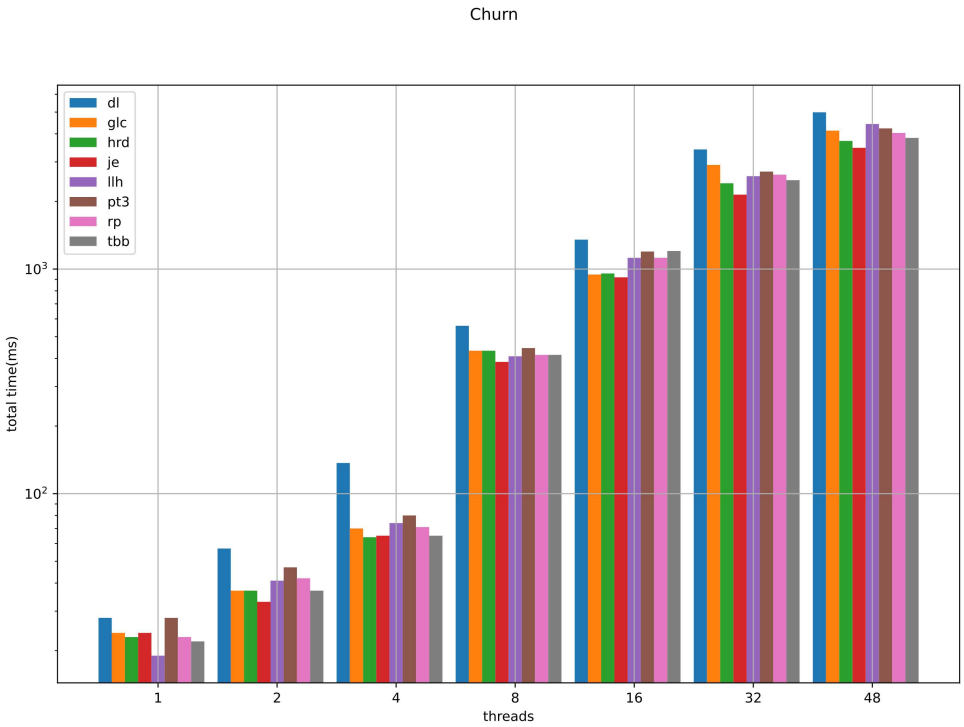
threads: 1, 2, 4, 8, 16, 32, 48
iterations: 1,000
cacheRW: 1,000,000
size: 1

Figure 5.2, p. 58 shows the results for algol and nasus. The X-axis shows the number of threads; the Y-axis shows the total experiment time. Each allocator's performance for each thread is shown in different colors.

Assessment All allocators did well in this micro-benchmark, except for dl and pt3. dl uses a single heap for all threads so it is understandable that it generates so much active false-sharing. Requests from different threads are dealt with sequentially by the single heap (using a single lock), which can allocate objects to different threads on the same cache line. pt3 uses the T:H model, so multiple threads can use one heap, but the active false-sharing is less than dl. The rest of the memory allocators generate little or no active false-sharing.

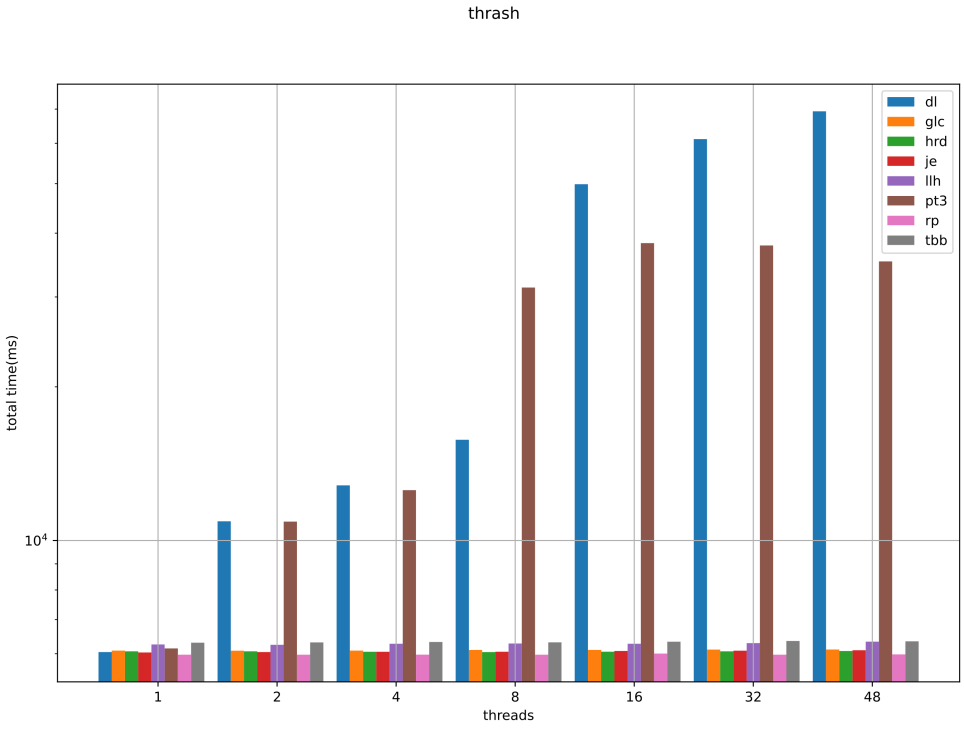


(a) Algol

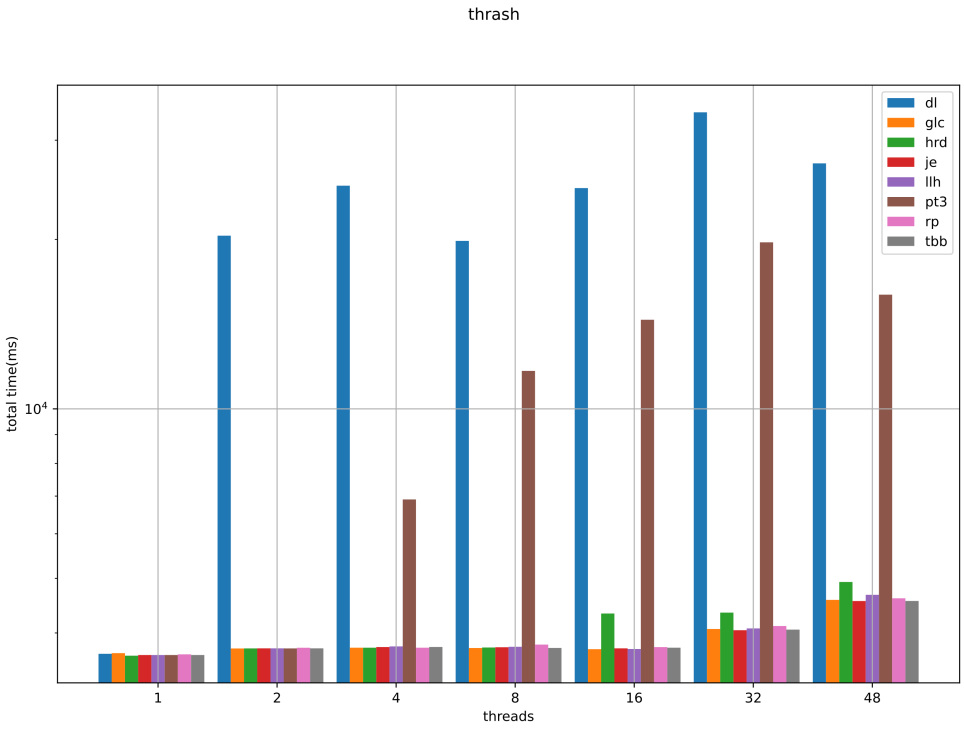


(b) Nasus

Figure 5.1: Churn



(a) Algol



(b) Nasus

Figure 5.2: Cache Thrash

5.3.3 Cache Scratch

Scratch tests memory allocators for program-induced allocator-preserved passive false-sharing (see Section 4.2.3, p. 49). This experiment was run with following configurations:

threads: 1, 2, 4, 8, 16, 32, 48
iterations: 1,000
cacheRW: 1,000,000
size: 1

Figure 5.3 shows the results for algol and nasus. The X-axis shows the number of threads; the Y-axis shows the total experiment time. Each allocator's performance for each thread is shown in different colors.

Assessment This micro-benchmark divides the allocators into two groups. First is the high-performer group: llh, je, and rp. These memory allocators generate little or no passive false-sharing and their performance difference is negligible. Second is the low-performer group, which includes the rest of the memory allocators. These memory allocators have significant program-induced passive false-sharing, where hrd's is the worst performing allocator. All of the allocators in this group are sharing heaps among threads at some level.

Interestingly, allocators such as hrd and glc performed well in micro-benchmark cache thrash (see Section 5.3.2, p. 56), but, these allocators are among the low performers in the cache scratch. It suggests these allocators do not actively produce false-sharing, but preserve program-induced passive false sharing.

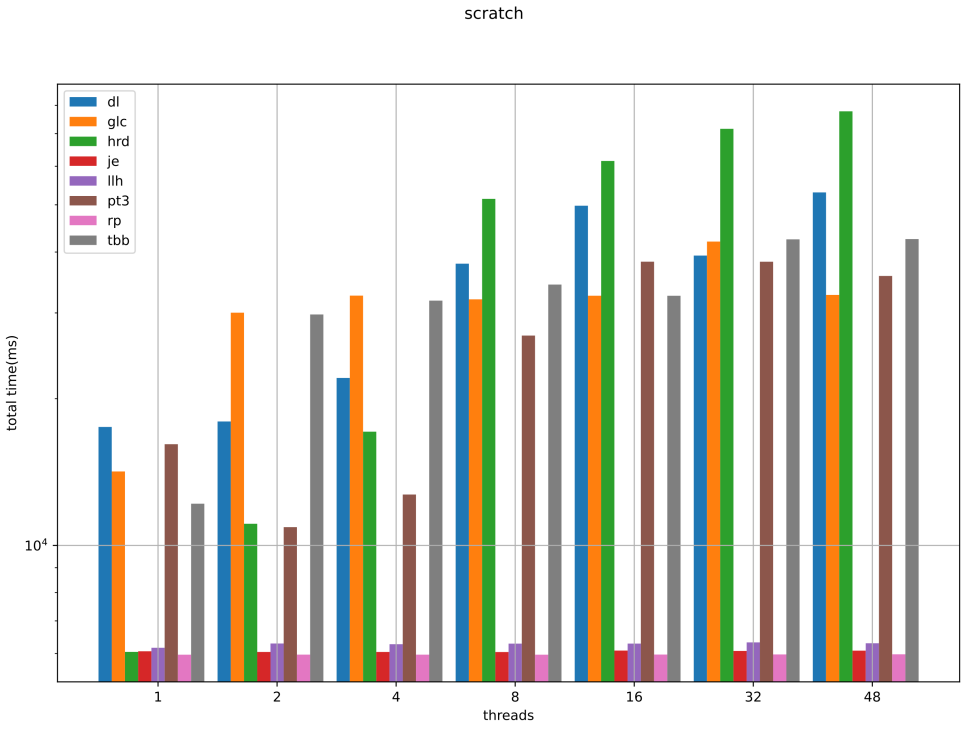
5.3.4 Speed Micro-Benchmark

Speed tests memory allocators for runtime latency (see Section 4.2.4, p. 50). This experiment was run with following configurations:

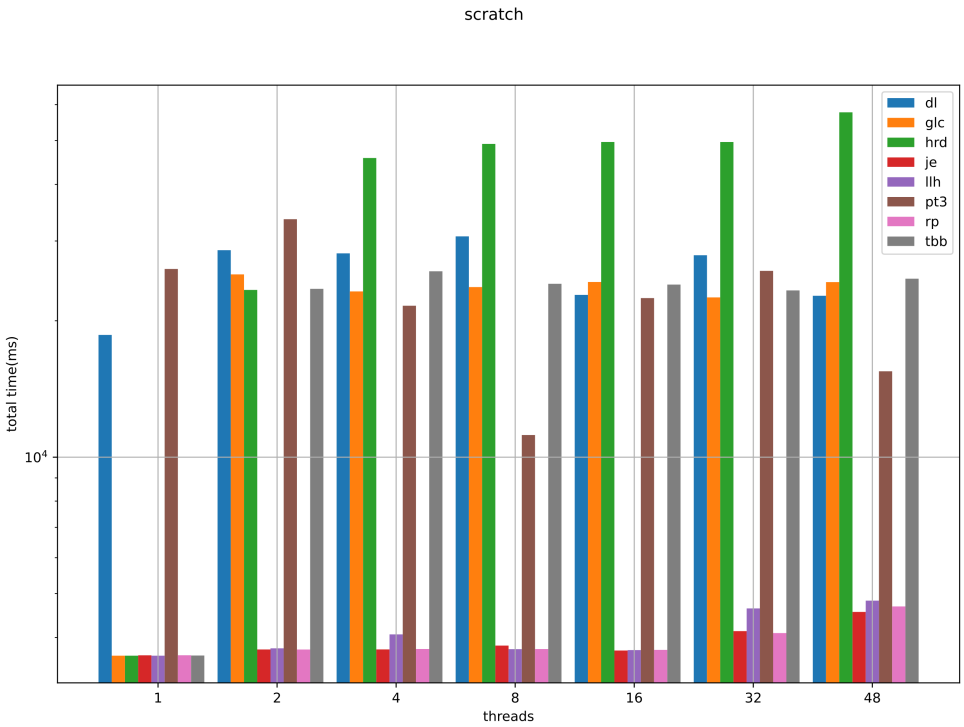
max: 500
min: 50
step: 50
distro: fisher
objects: 100,000
workers: 1, 2, 4, 8, 16, 32, 48

Figures 5.4 to 5.15, pp. 62–73 show 12 figures, one figure for each chain of the speed benchmark. The X-axis shows the number of threads; the Y-axis shows the total experiment time. Each allocator's performance for each thread is shown in different colors.

- Figure 5.4, p. 62 shows results for chain: malloc
- Figure 5.5, p. 63 shows results for chain: realloc
- Figure 5.6, p. 64 shows results for chain: free
- Figure 5.7, p. 65 shows results for chain: calloc



(a) Algol



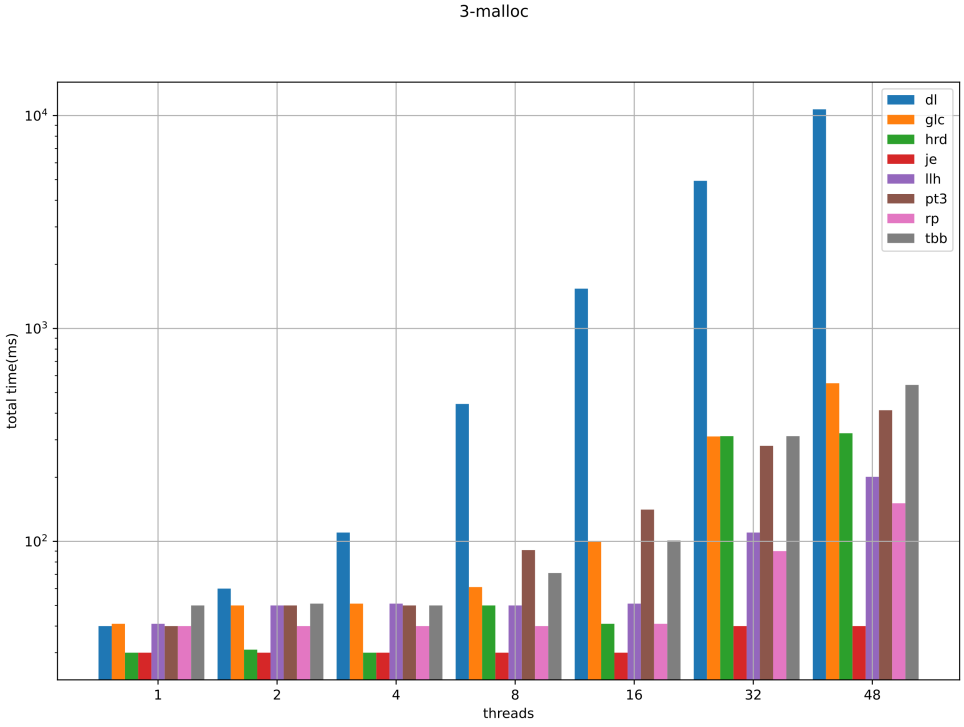
(b) Nasus

Figure 5.3: Cache Scratch

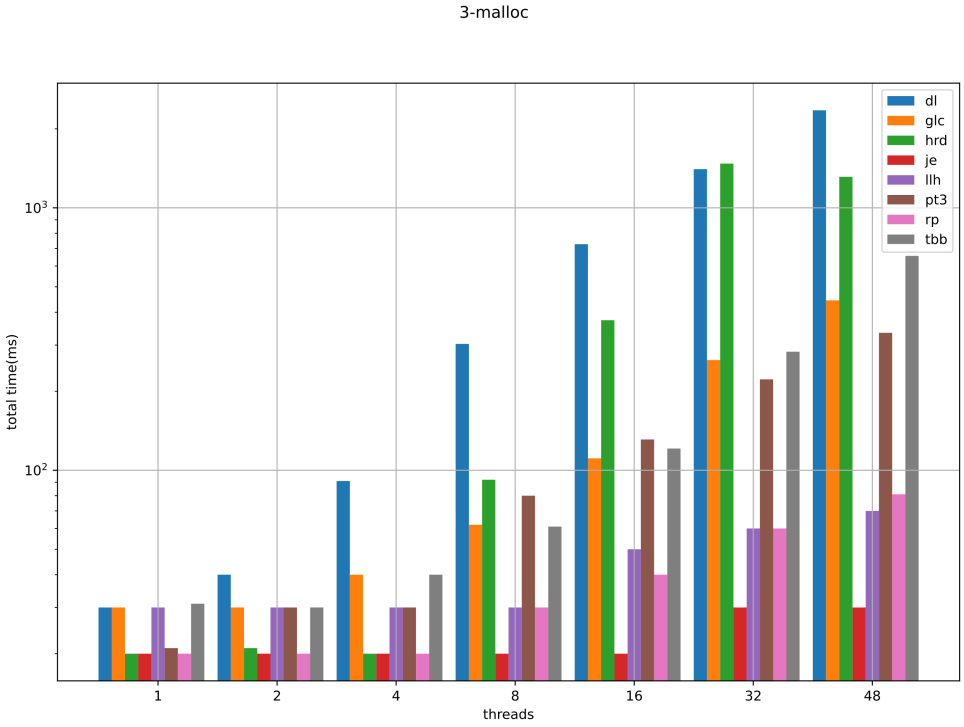
- Figure 5.8, p. 66 shows results for chain: malloc-free
- Figure 5.9, p. 67 shows results for chain: realloc-free
- Figure 5.10, p. 68 shows results for chain: calloc-free
- Figure 5.11, p. 69 shows results for chain: malloc-realloc
- Figure 5.12, p. 70 shows results for chain: calloc-realloc
- Figure 5.13, p. 71 shows results for chain: malloc-realloc-free
- Figure 5.14, p. 72 shows results for chain: calloc-realloc-free
- Figure 5.15, p. 73 shows results for chain: malloc-realloc-free-calloc

Assessment This micro-benchmark divides the allocators into two groups: with and without calloc. calloc uses memset to set the allocated memory to zero, which dominates the cost of the allocation chain (large performance increase) and levels performance across the allocators. But the difference among the allocators in a calloc chain still gives an idea of their relative performance.

All allocators did well in this micro-benchmark across all allocation chains, except for dl, pt3, and hrd. Again, the low-performing allocators are sharing heaps among threads, so the contention causes performance increases with increasing numbers of threads. Furthermore, chains with free can trigger coalescing, which slows the fast path. The high-performing allocators all illustrate low latency across the allocation chains, *i.e.*, there are no performance spikes as the chain lengths, that might be caused by contention and/or coalescing. Low latency is important for applications that are sensitive to unknown execution delays.

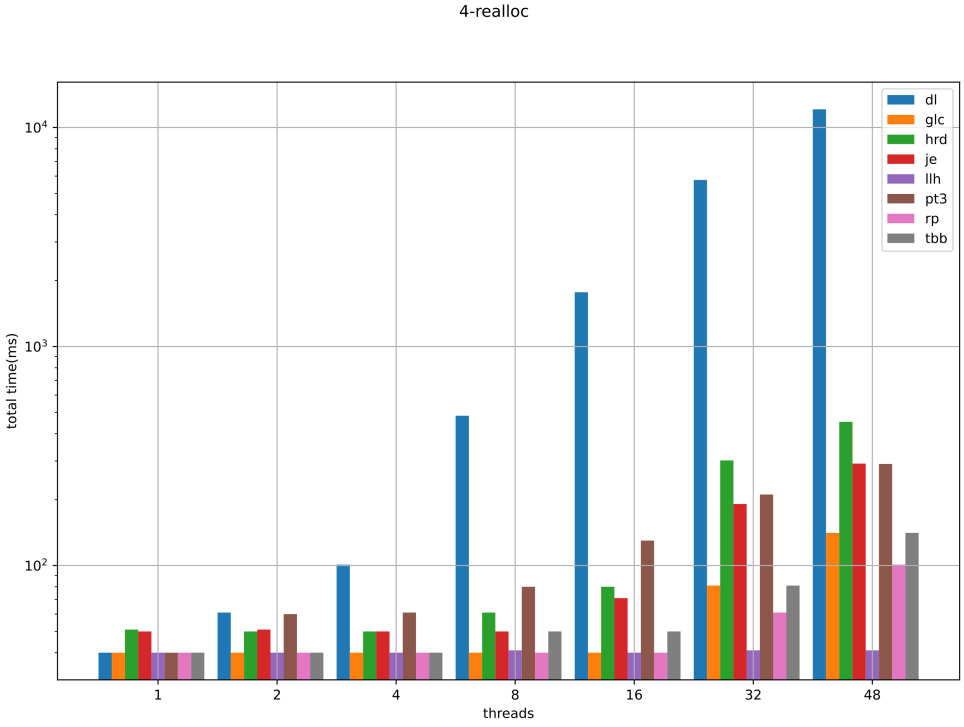


(a) Algol

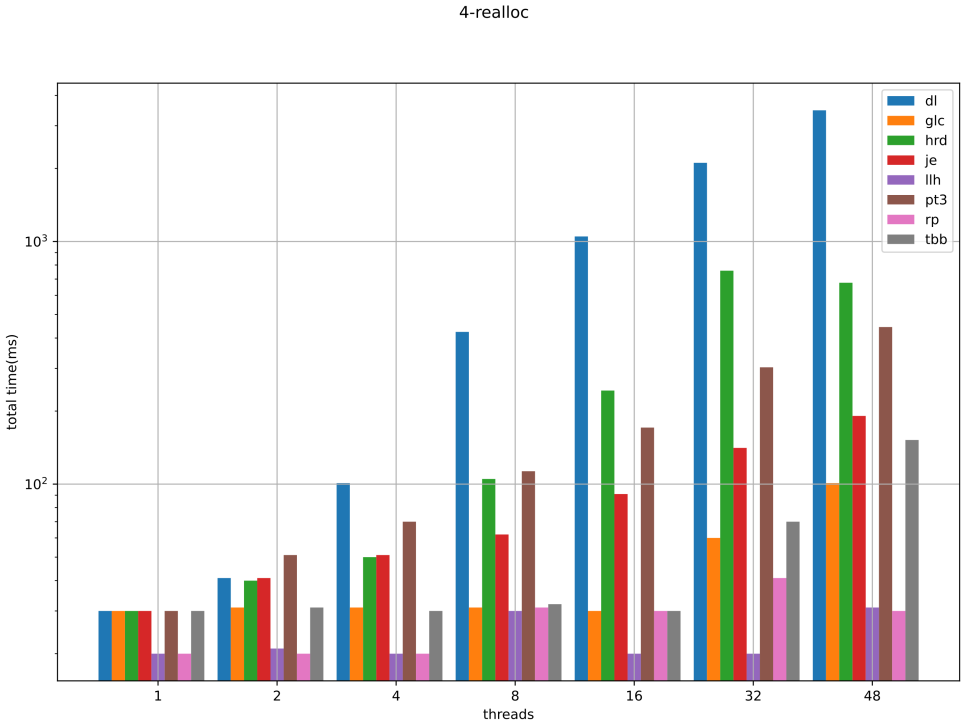


(b) Nasus

Figure 5.4: Speed benchmark chain: malloc

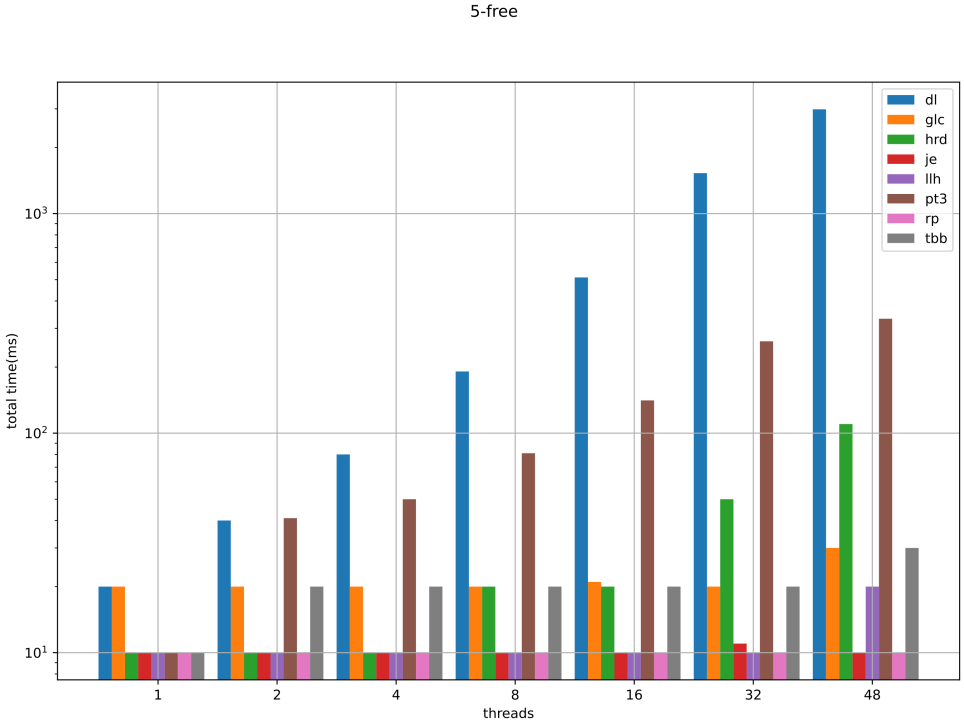


(a) Algol

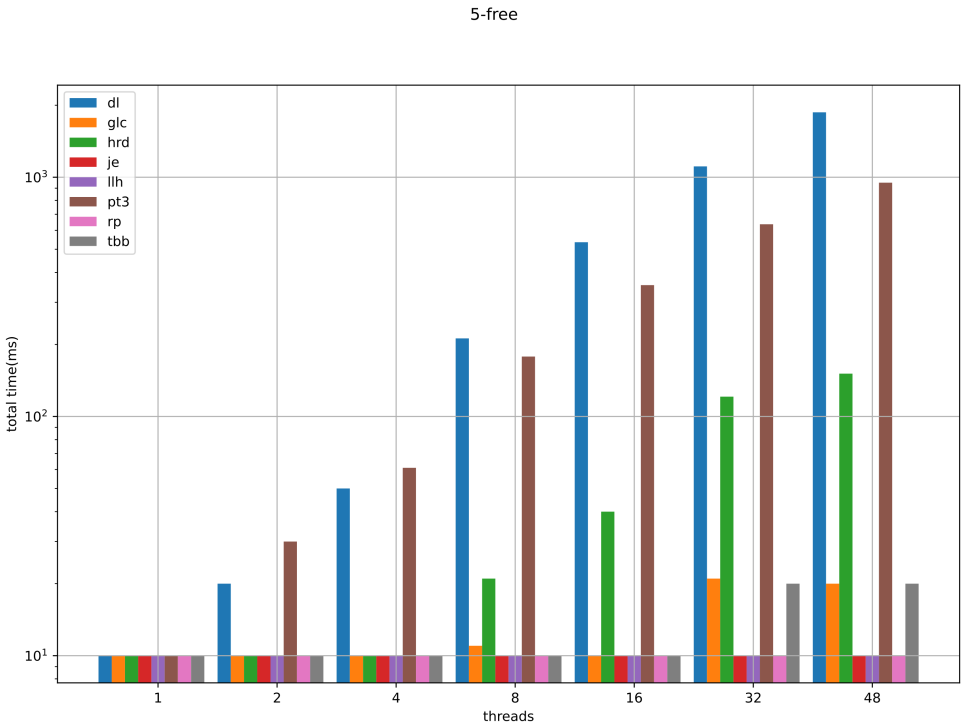


(b) Nasus

Figure 5.5: Speed benchmark chain: realloc

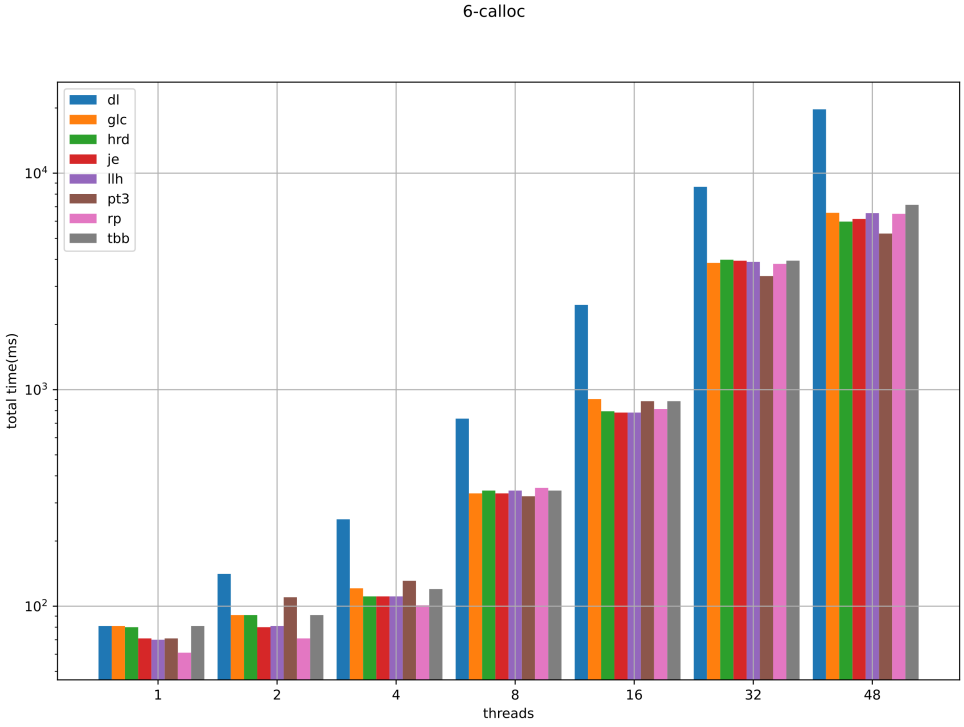


(a) Algol

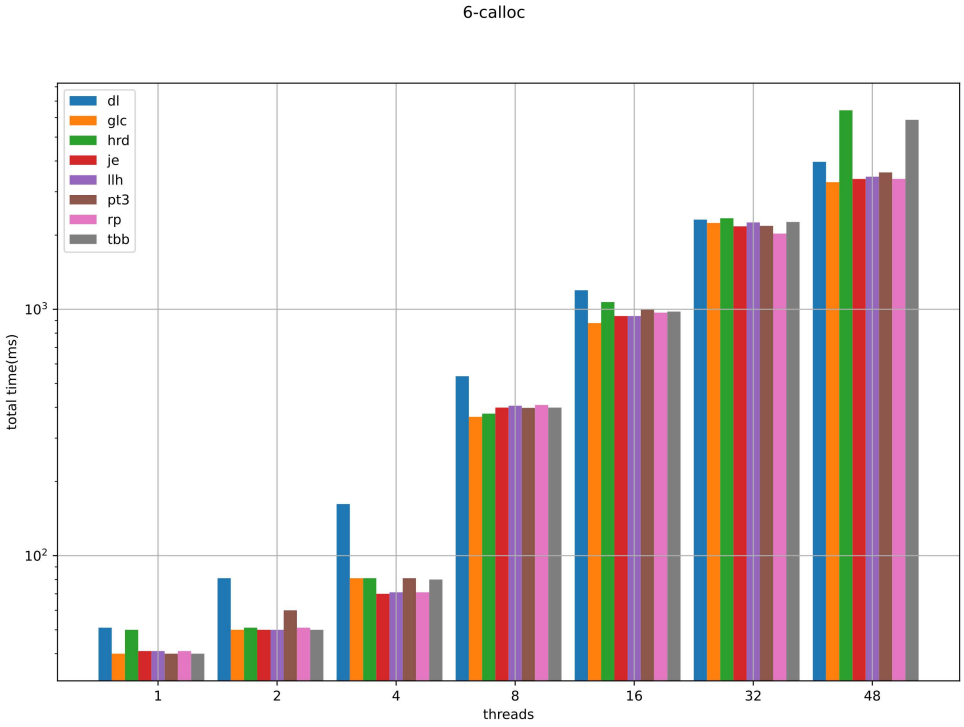


(b) Nasus

Figure 5.6: Speed benchmark chain: free

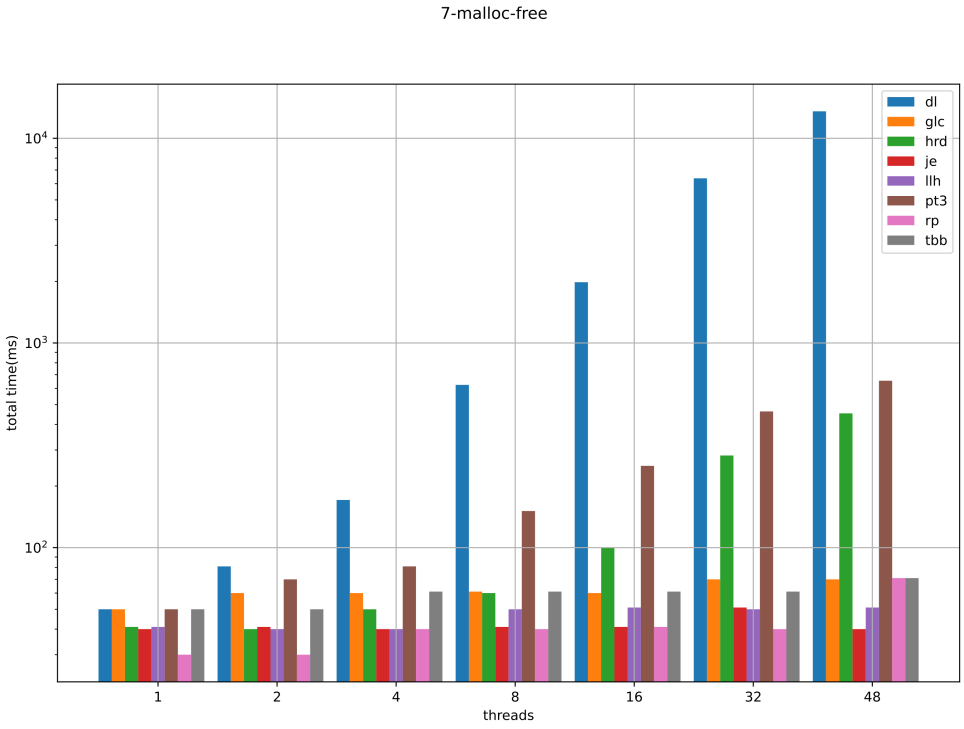


(a) Algol

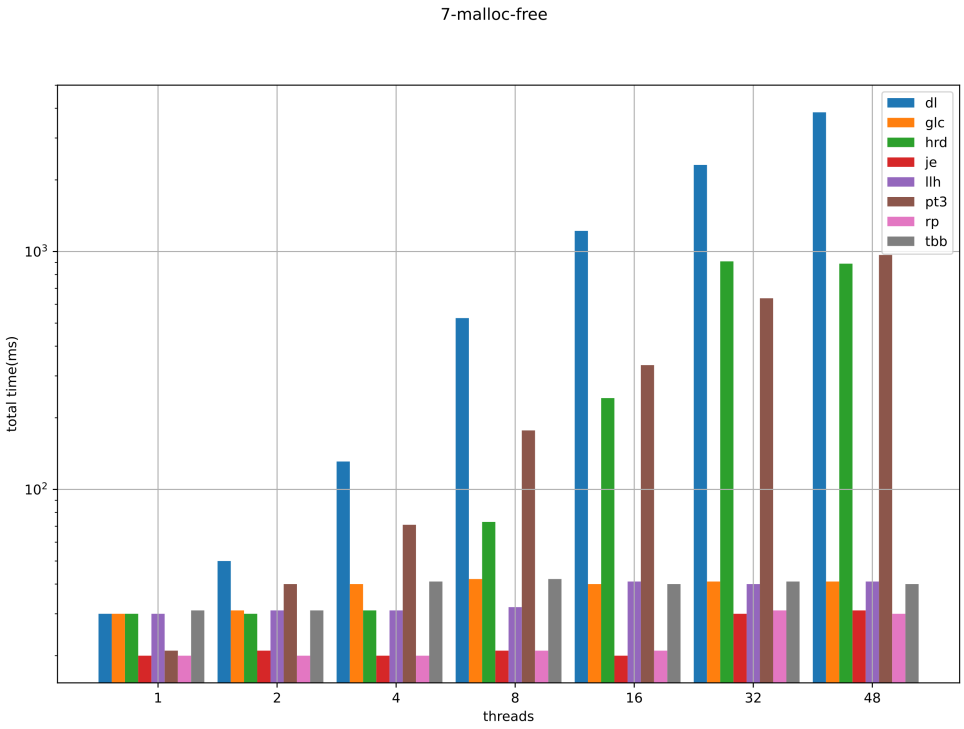


(b) Nasus

Figure 5.7: Speed benchmark chain: calloc

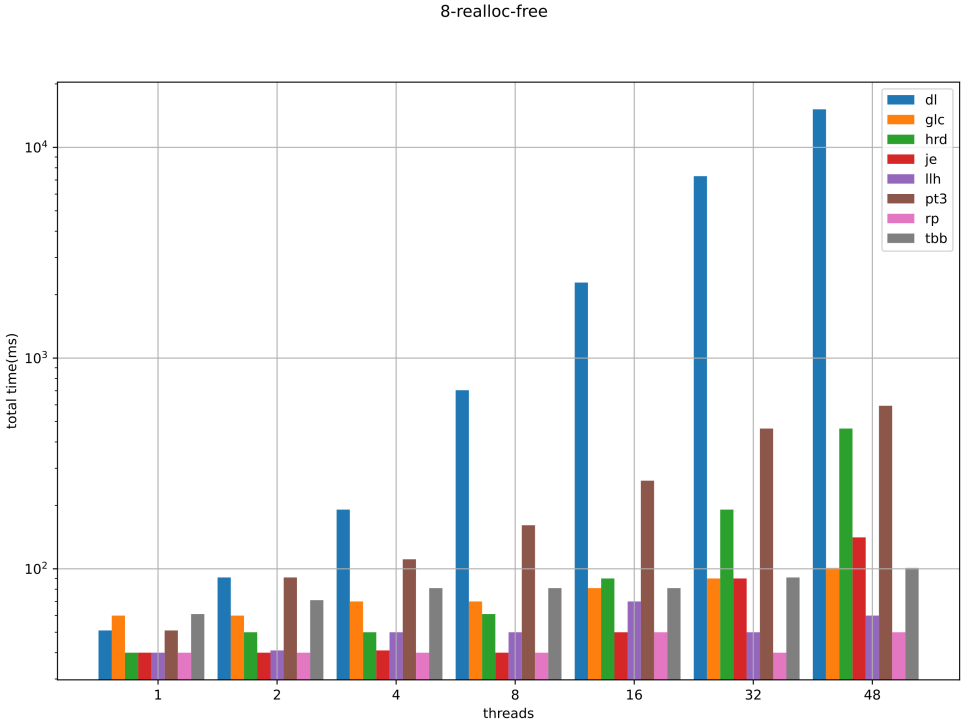


(a) Algol

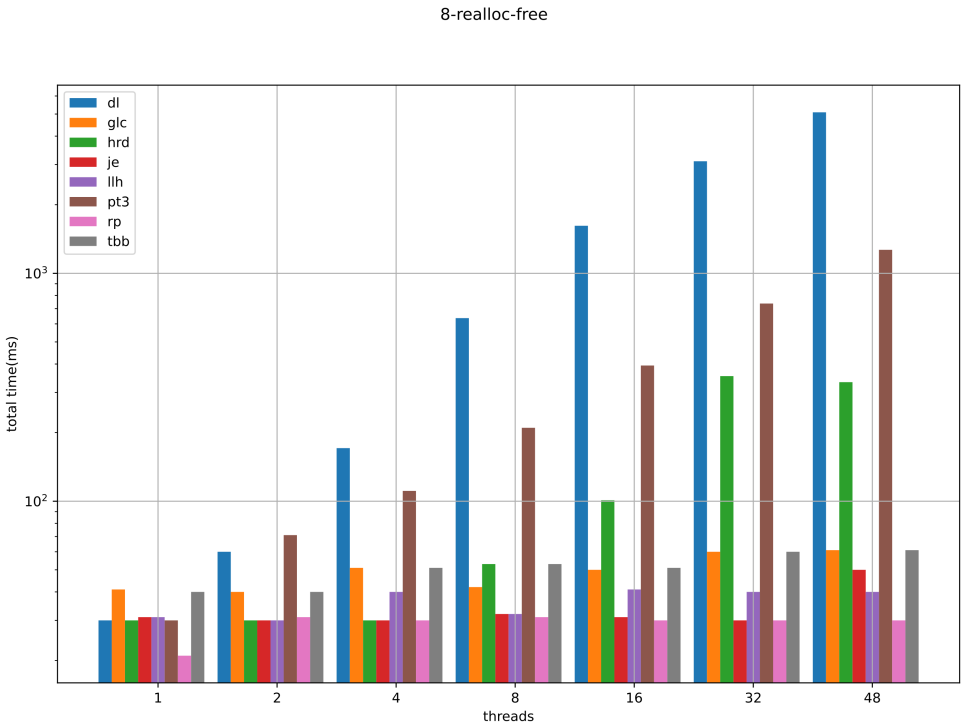


(b) Nasus

Figure 5.8: Speed benchmark chain: malloc-free

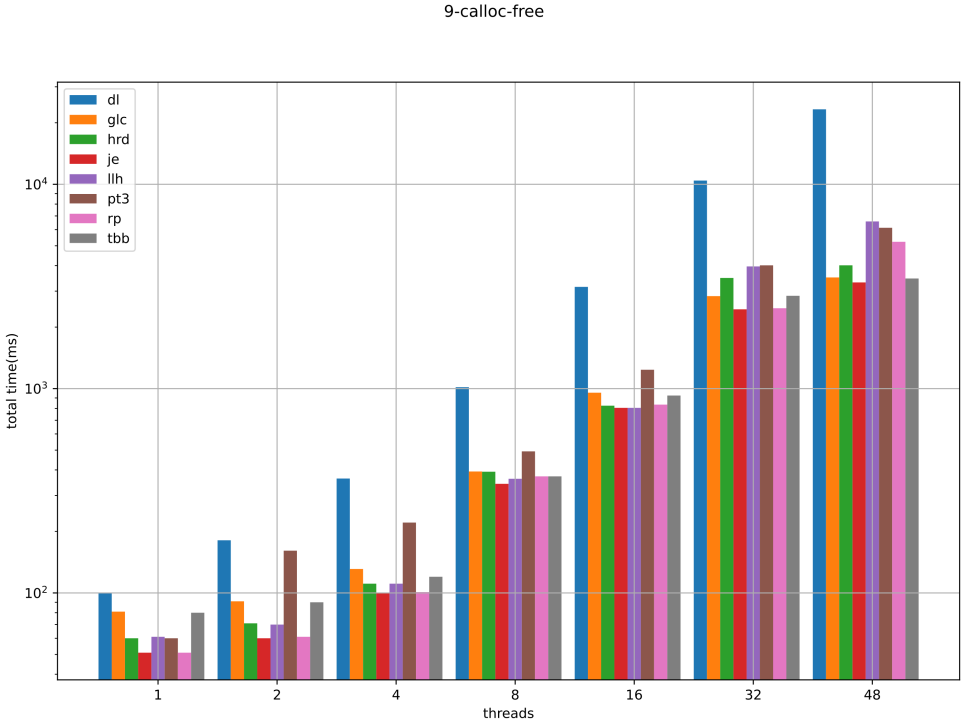


(a) Algol

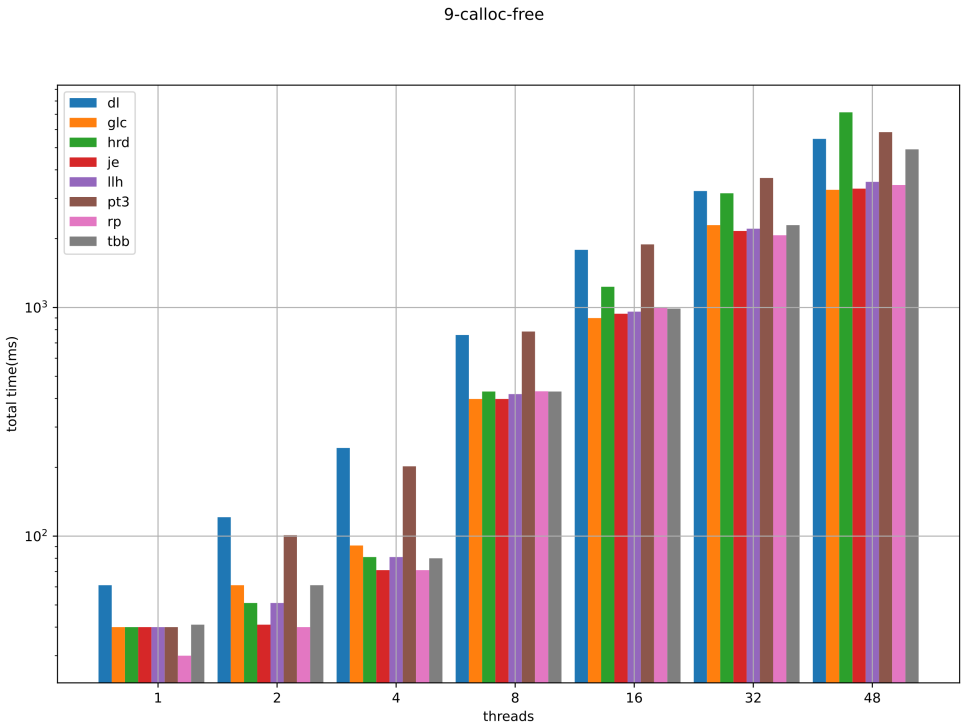


(b) Nasus

Figure 5.9: Speed benchmark chain: realloc-free

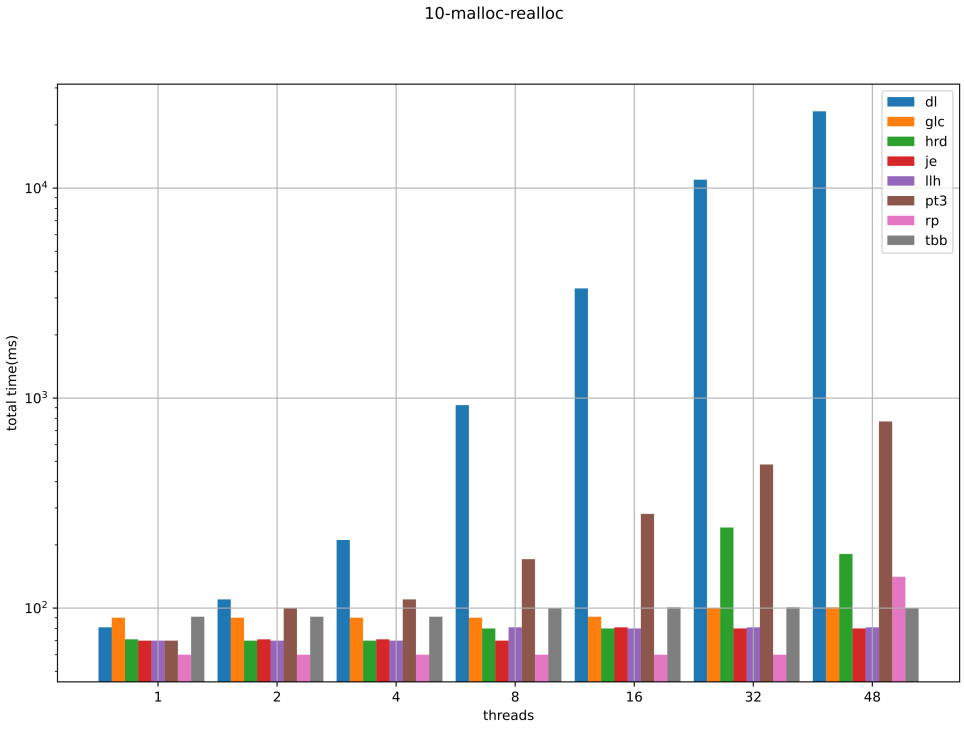


(a) Algol

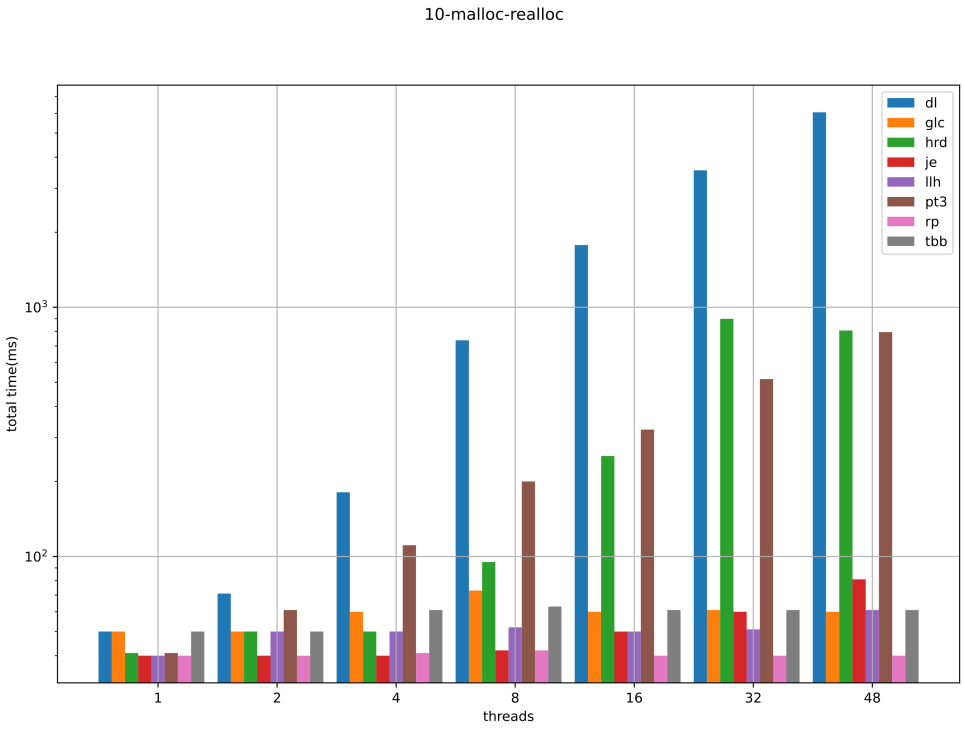


(b) Nasus

Figure 5.10: Speed benchmark chain: calloc-free

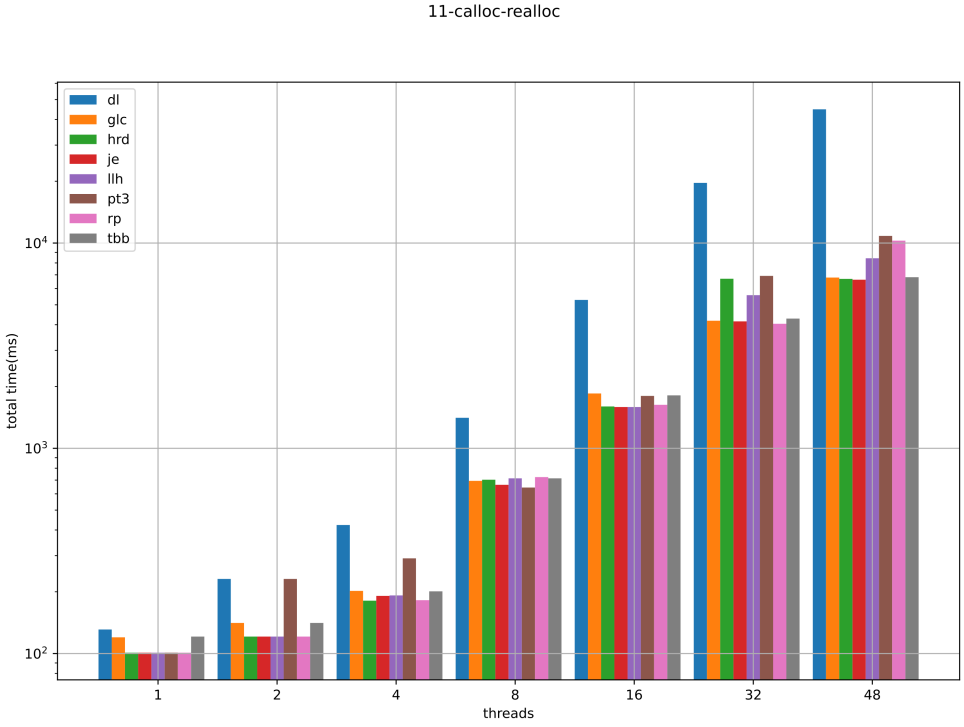


(a) Algol

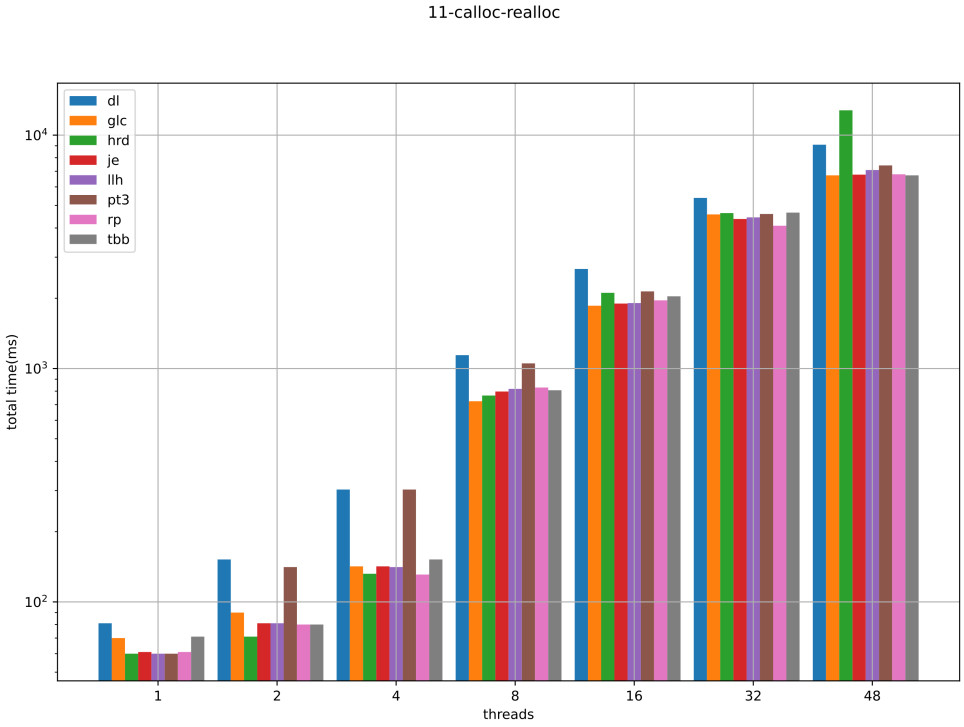


(b) Nasus

Figure 5.11: Speed benchmark chain: malloc-realloc

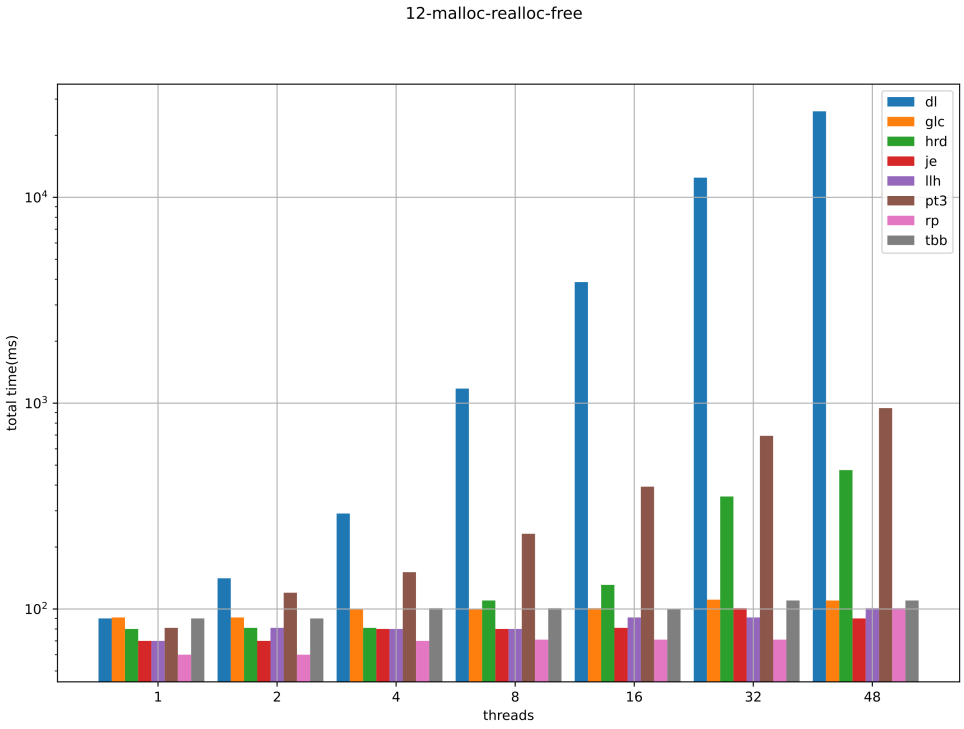


(a) Algol

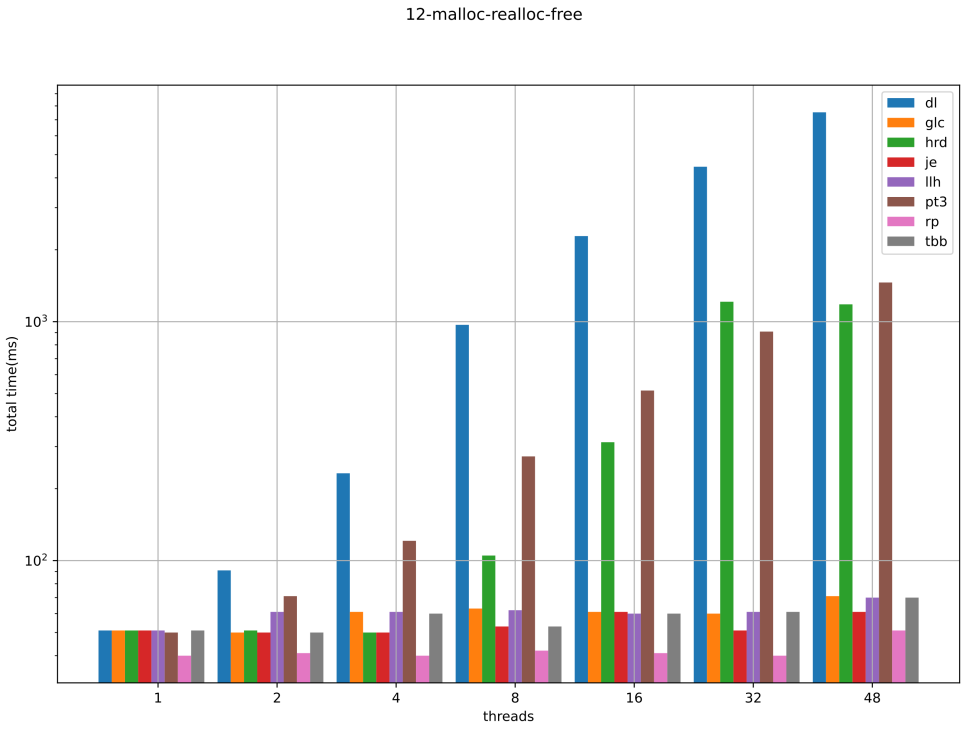


(b) Nasus

Figure 5.12: Speed benchmark chain: calloc-realloc

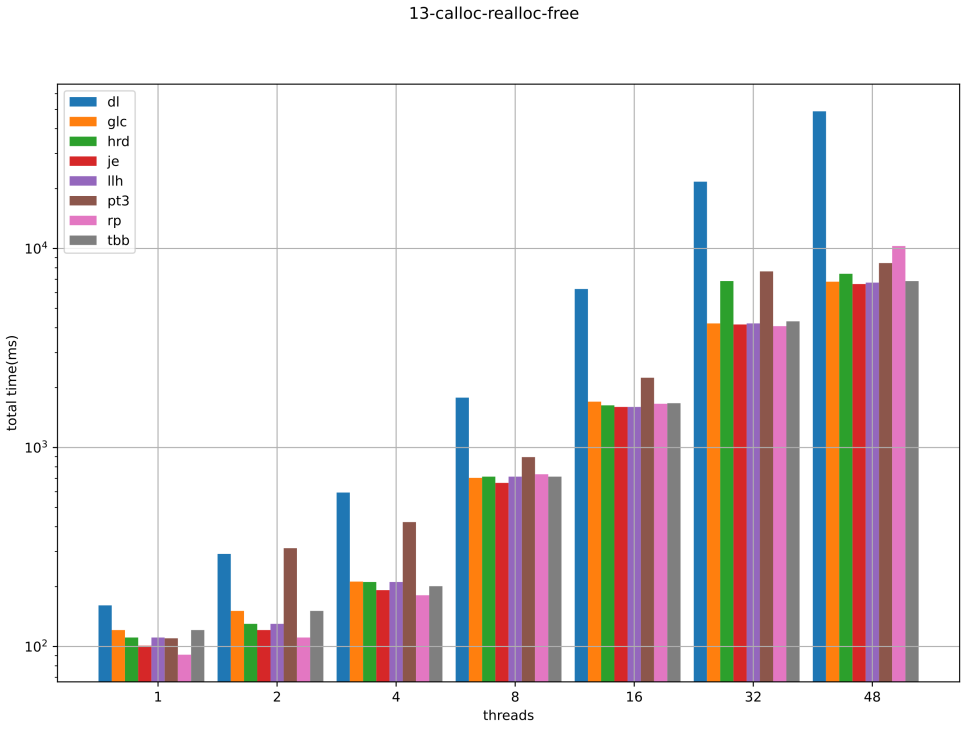


(a) Algol

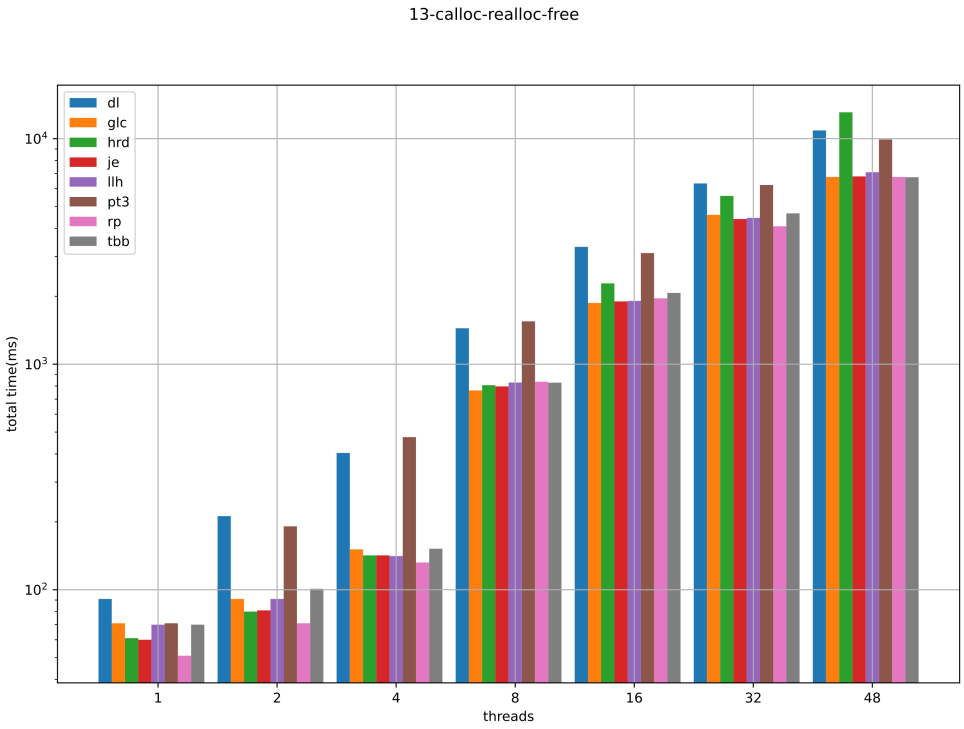


(b) Nasus

Figure 5.13: Speed benchmark chain: malloc-realloc-free

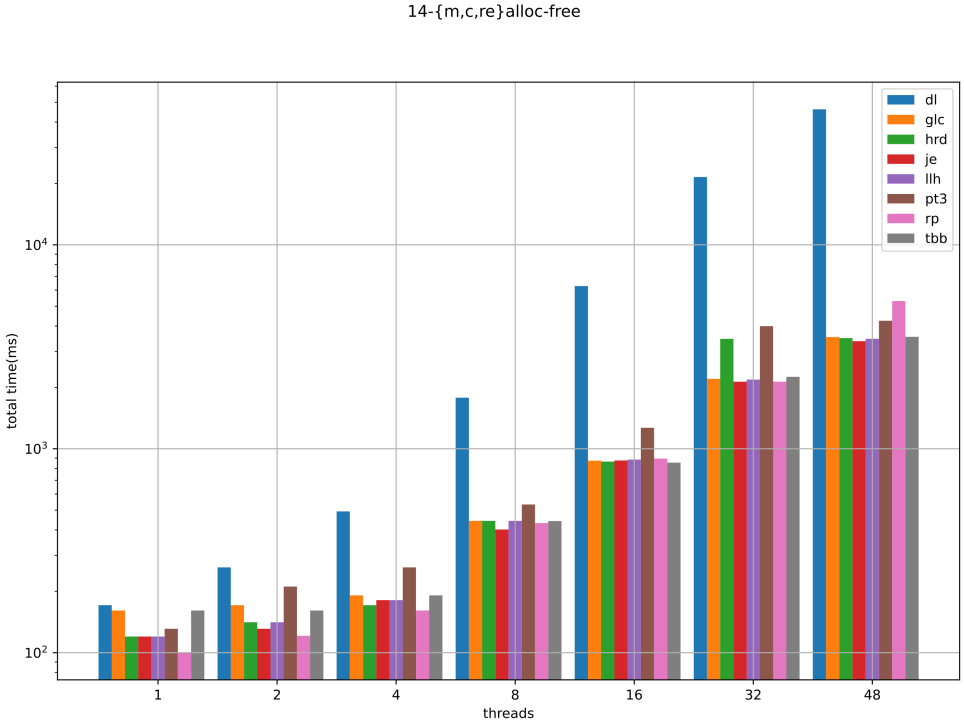


(a) Algol

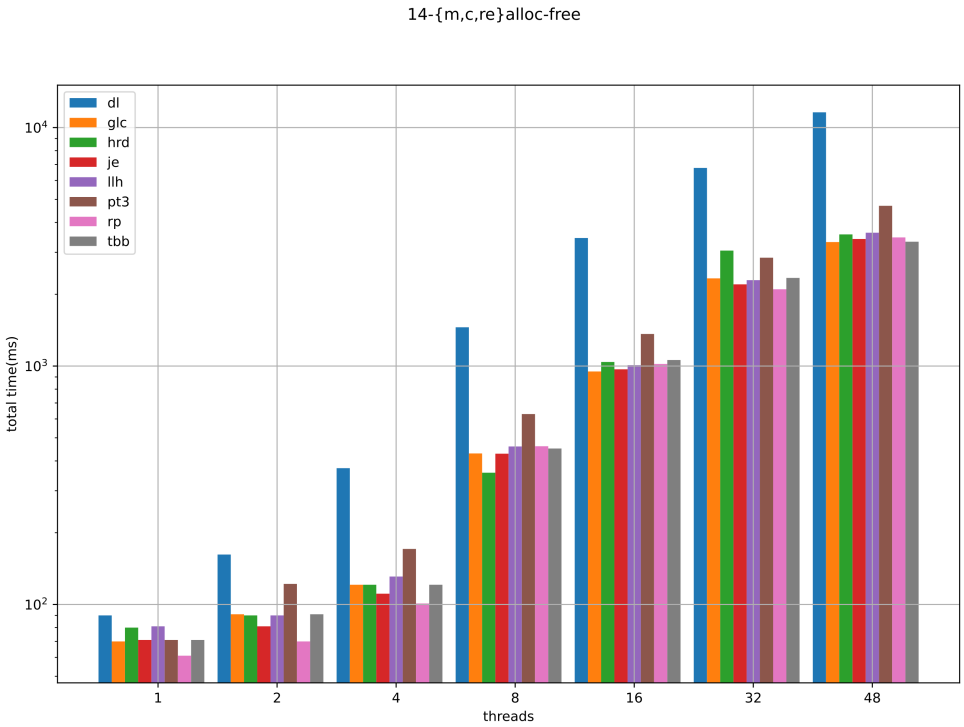


(b) Nasus

Figure 5.14: Speed benchmark chain: calloc-realloc-free



(a) Algol



(b) Nasus

Figure 5.15: Speed benchmark chain: malloc-calloc-realloc-free

5.3.5 Memory Micro-Benchmark

This experiment is run with the following two configurations for each allocator. The difference between the two configurations is the number of producers and consumers. Configuration 1 has one producer and one consumer, and configuration 2 has 4 producers, where each producer has 4 consumers.

Configuration 1:

producer (K): 1
consumer (M): 1
round: 100,000
max: 500
min: 50
step: 50
distro: fisher
objects (N): 100,000

Configuration 2:

producer (K): 4
consumer (M): 4
round: 100,000
max: 500
min: 50
step: 50
distro: fisher
objects (N): 100,000

Figures 5.16 to 5.31, pp. 76–91 show 16 figures, two figures for each of the 8 allocators, one for each configuration. Each figure has 2 graphs, one for each experiment environment. Each graph has following 5 subgraphs that show memory usage and statistics throughout the micro-benchmark's lifetime.

- ***current_req_mem(B)*** shows the amount of dynamic memory requested and currently in-use of the benchmark.
- ***heap**** shows the memory requested by the program (allocator) from the system that lies in the heap (sbrk) area.
- ***mmap_so**** shows the memory requested by the program (allocator) from the system that lies in the mmap area.
- ***mmap**** shows the memory requested by the program (allocator or shared libraries) from the system that lies in the mmap area.
- ***total_dynamic*** shows the total usage of dynamic memory by the benchmark program, which is a sum of *heap*, *mmap*, and *mmap_so*.

* These statistics are gathered by monitoring a process's /proc/self/maps file.

The X-axis shows the time when the memory information is polled. The Y-axis shows the memory usage in bytes.

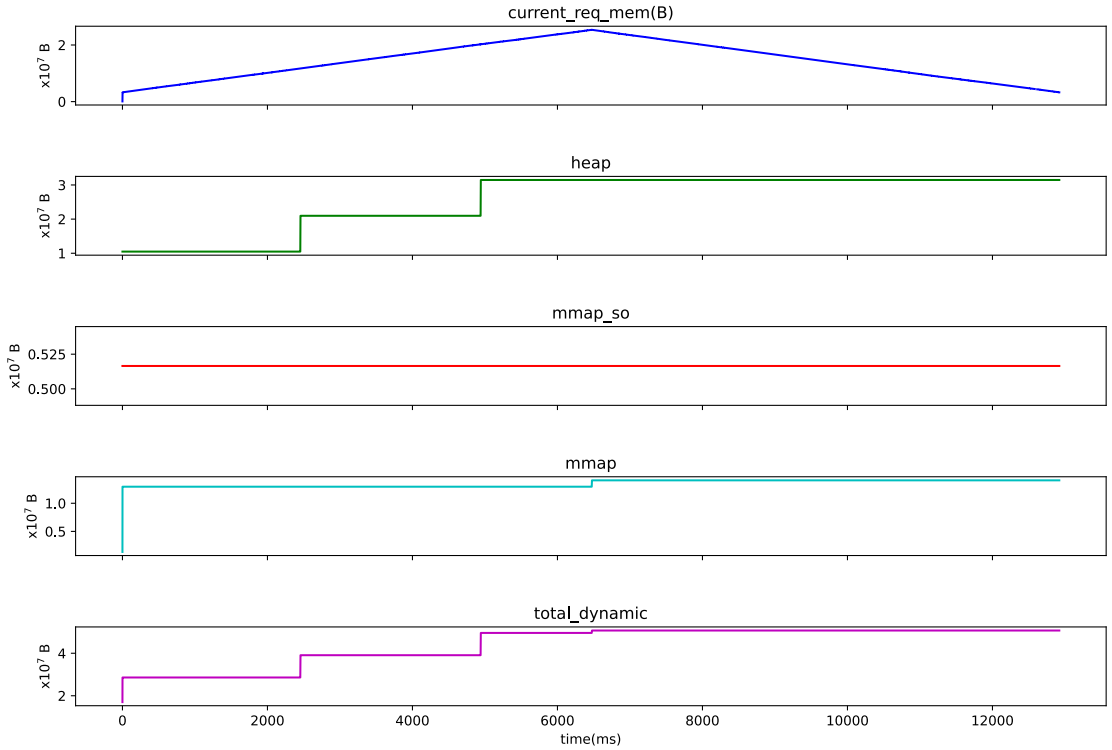
For this experiment, the difference between the memory requested by the benchmark (*current_req_mem(B)*) and the memory that the process has received from system (*heap*, *mmap*) should be minimum. This difference is the memory overhead caused by the allocator and shows the level of fragmentation in the allocator.

Assessment First, the differences in the shape of the curves between architectures (top ARM, bottom x64) is small, where the differences are in the amount of memory used. Hence, it is possible to focus on either the top or bottom graph.

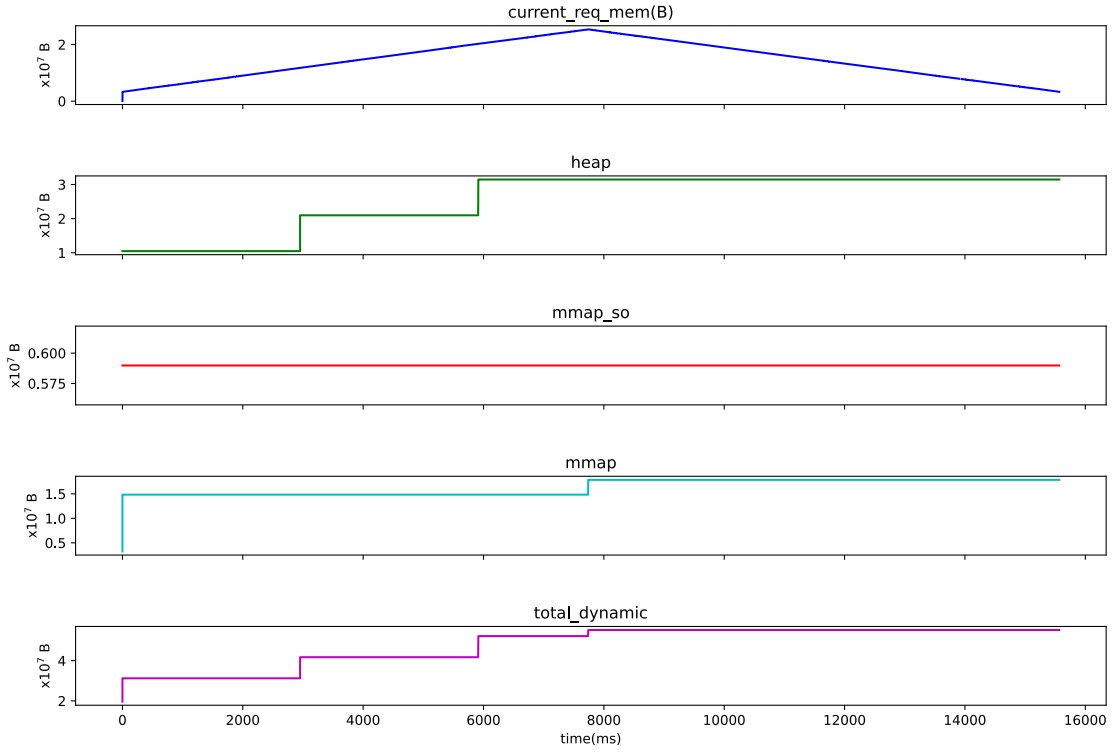
Second, the heap curve is 0 for four memory allocators: hrd, je, pt3, and rp, indicating these memory allocators only use mmap to get memory from the system and ignore the sbrk area.

The total dynamic memory is higher for hrd and tbb than the other allocators. The main reason is the use of superblocks (see Section 2.6, p. 16) containing objects of the same size. These superblocks are maintained throughout the life of the program.

pt3 is the only memory allocator where the total dynamic memory goes down in the second half of the program lifetime when the memory is freed by the benchmark program. It makes pt3 the only memory allocator that gives memory back to the operating system as it is freed by the program.

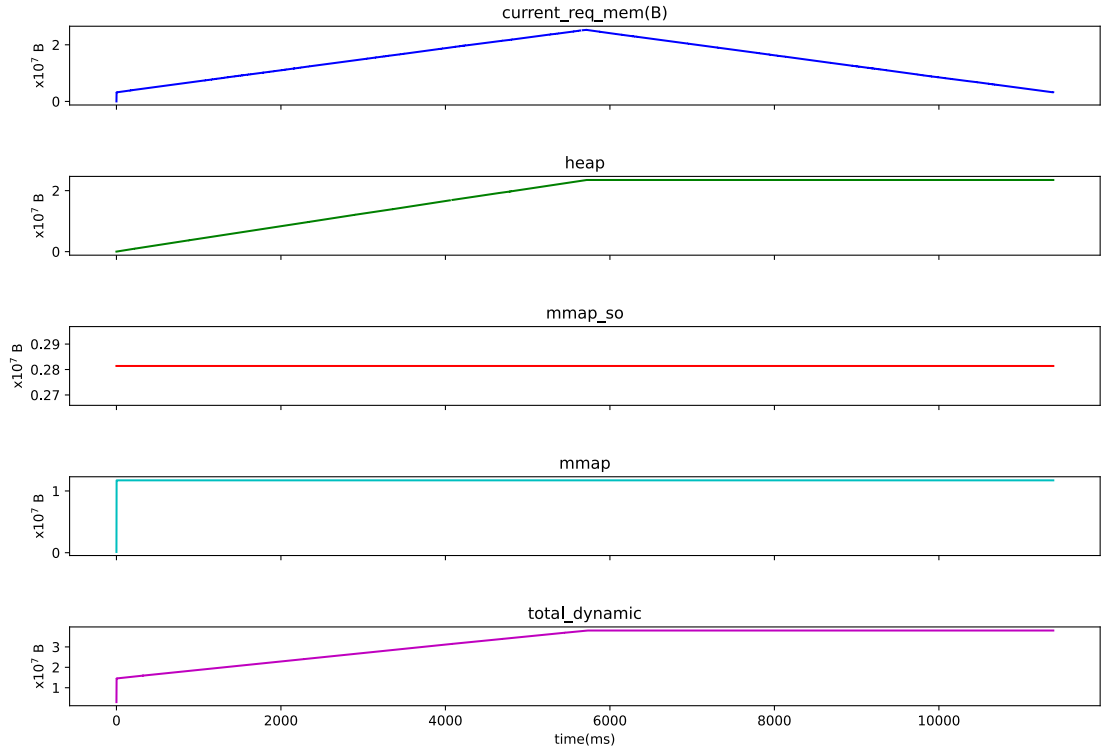


(a) Algol

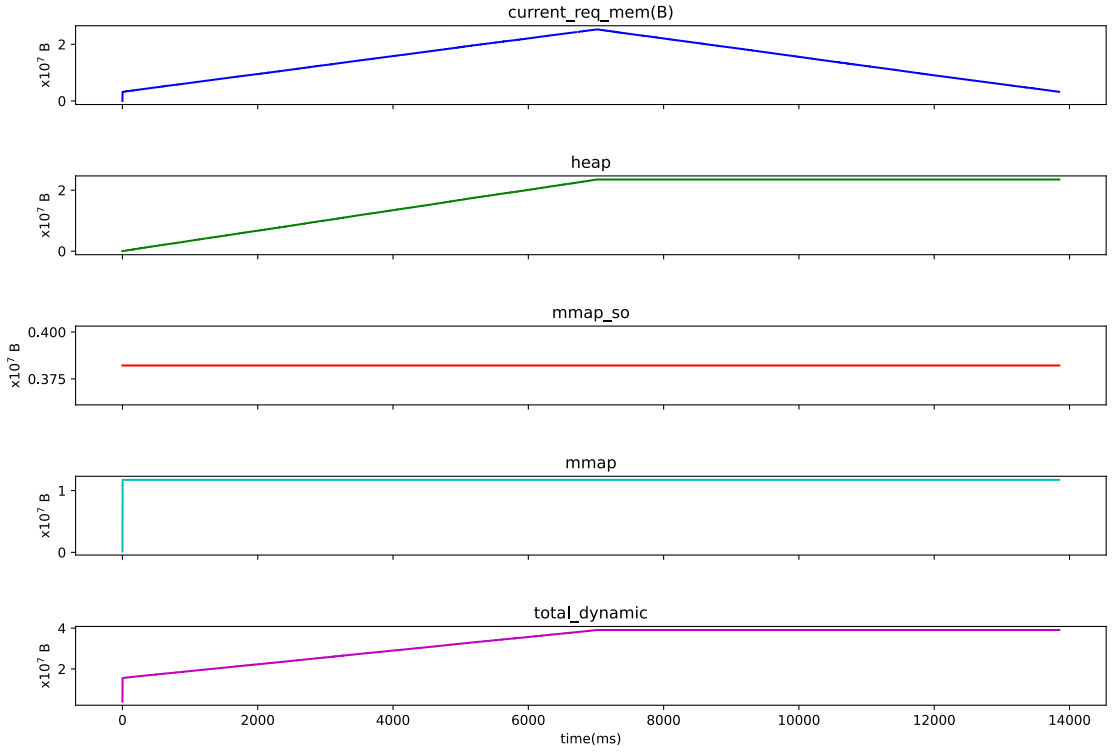


(b) Nasus

Figure 5.16: Memory benchmark results with Configuration-1 for llh memory allocator

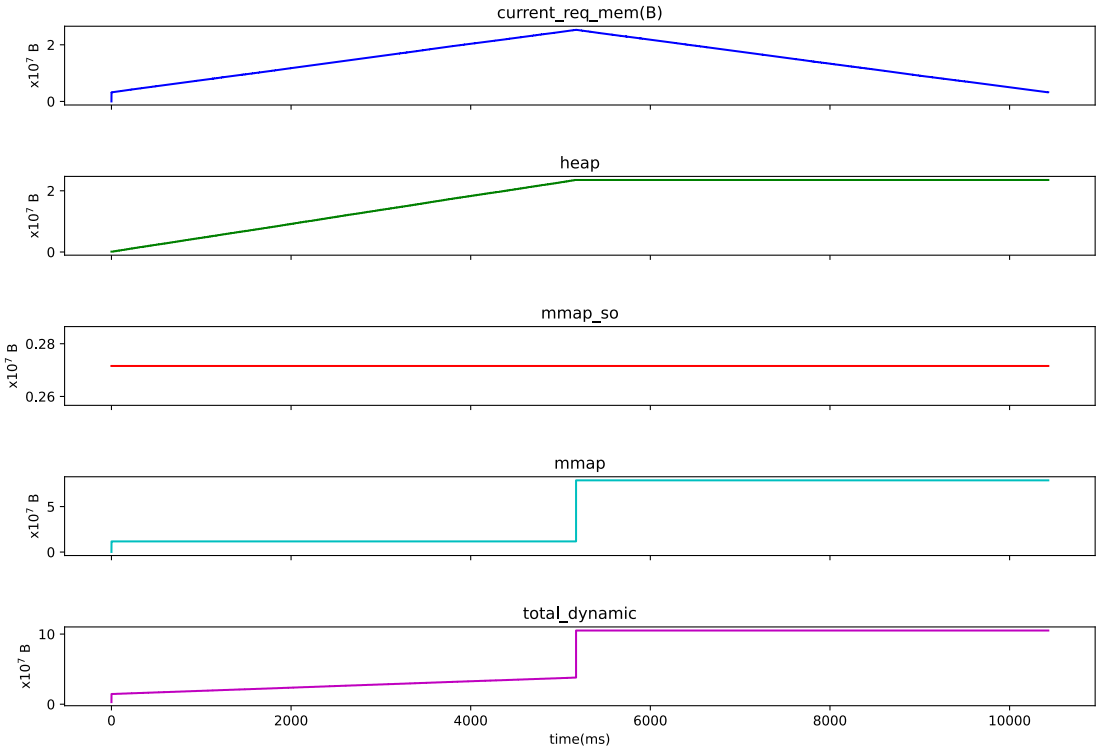


(a) Algol

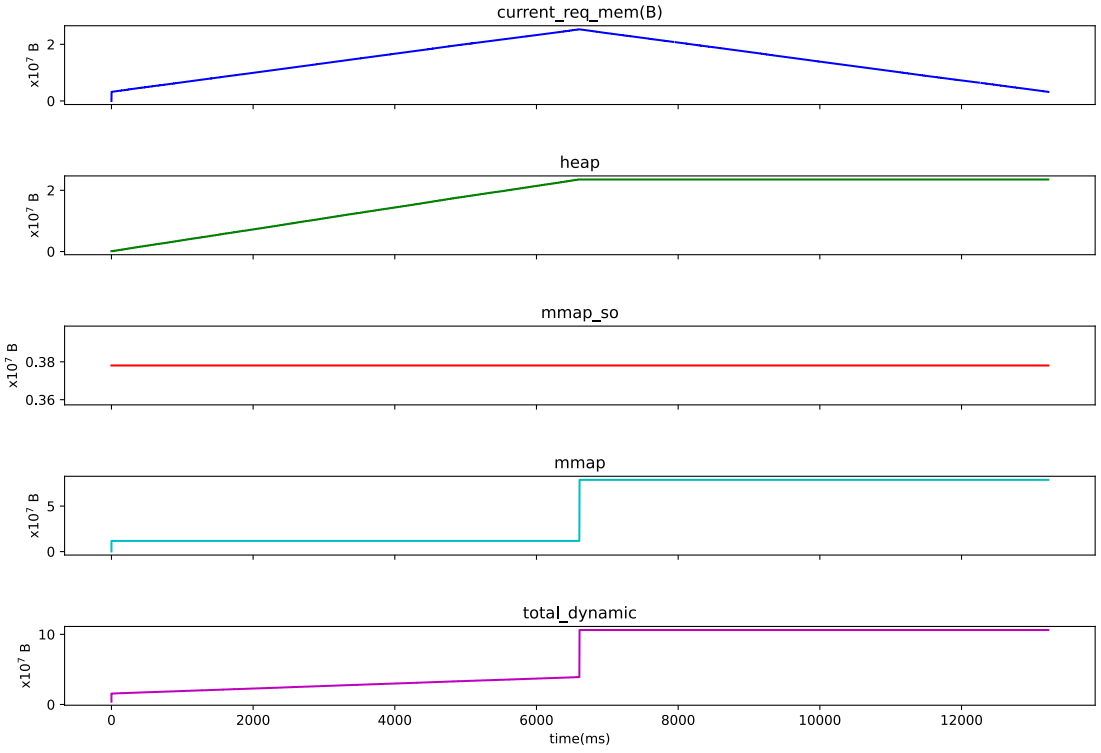


(b) Nasus

Figure 5.17: Memory benchmark results with Configuration-1 for dl memory allocator

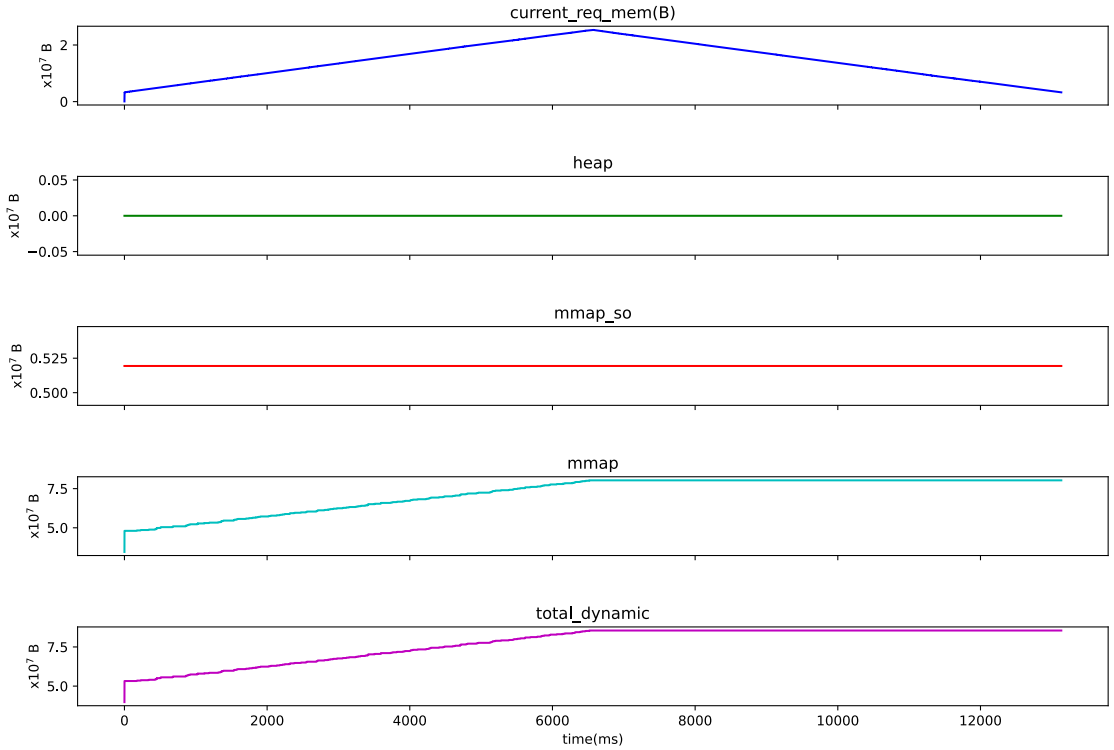


(a) Algol

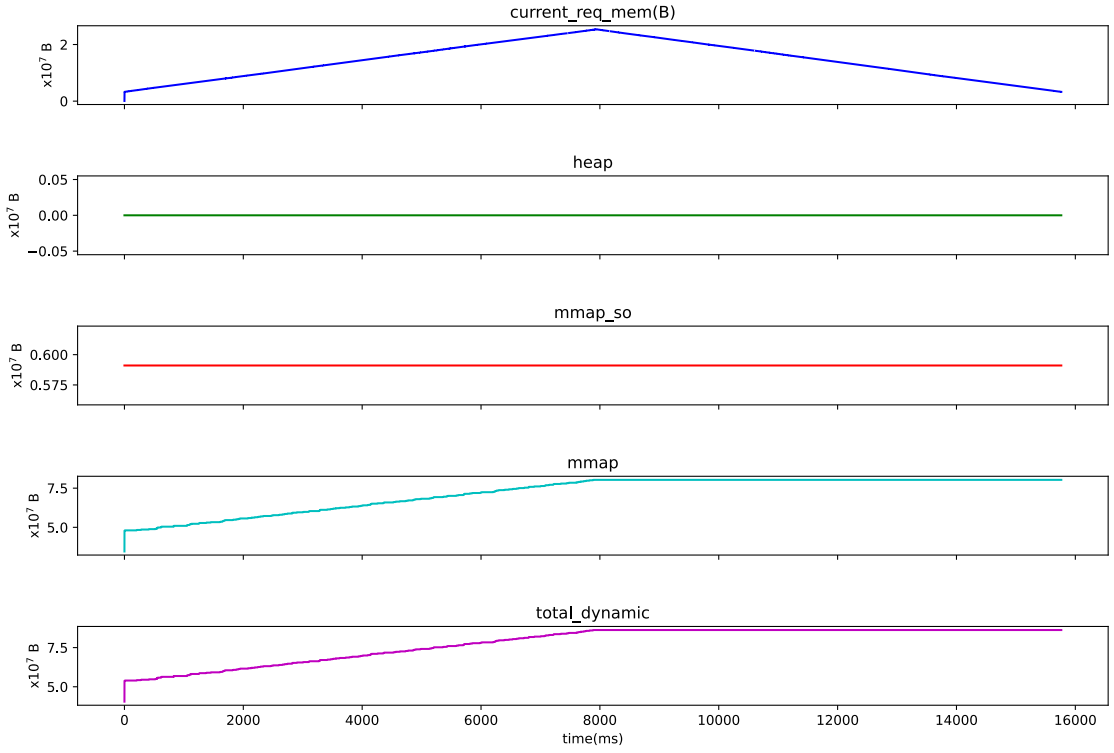


(b) Nasus

Figure 5.18: Memory benchmark results with Configuration-1 for glibc memory allocator

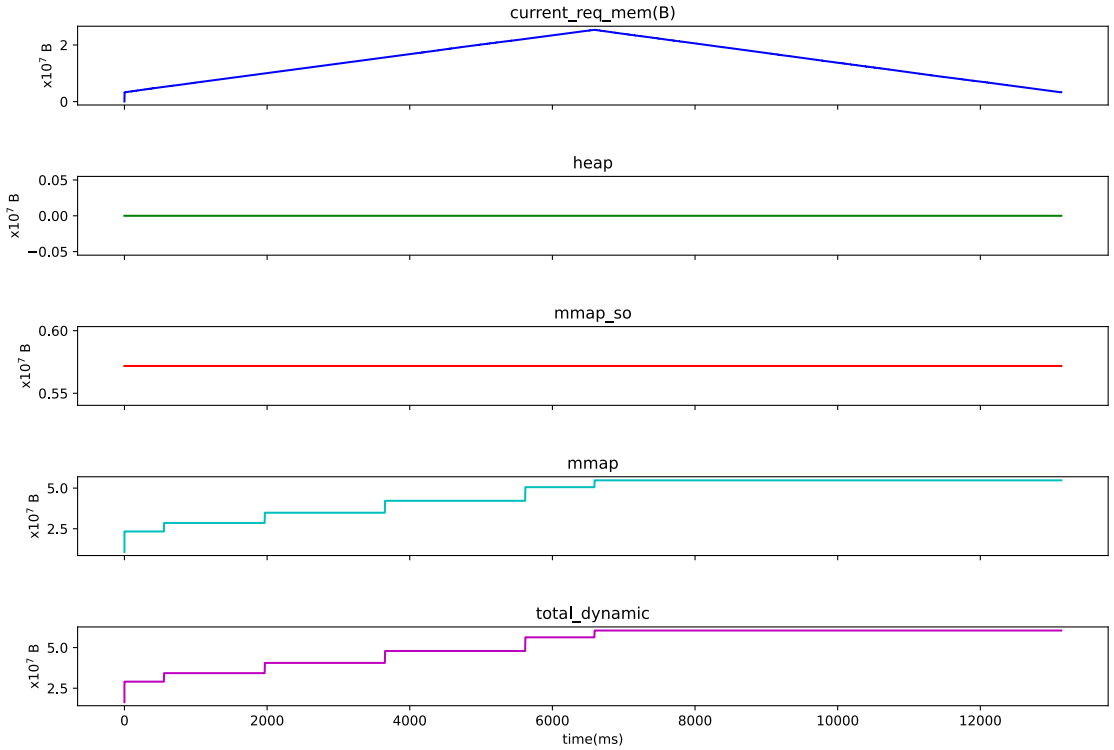


(a) Algol

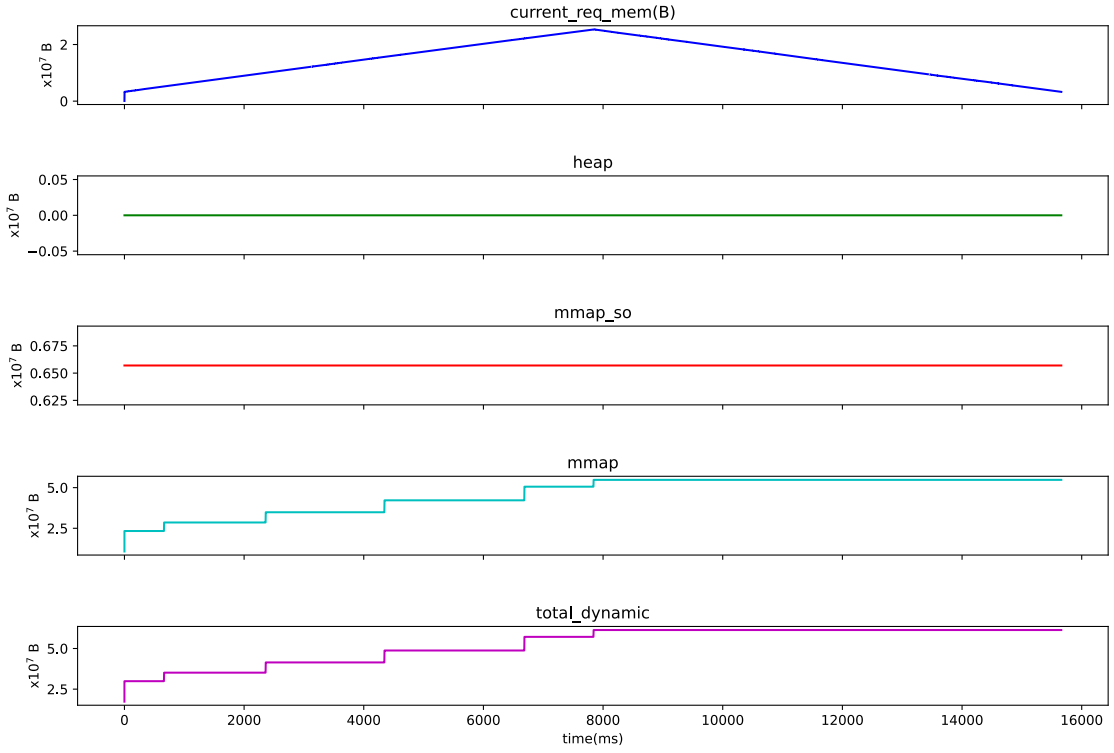


(b) Nasus

Figure 5.19: Memory benchmark results with Configuration-1 for hoard memory allocator

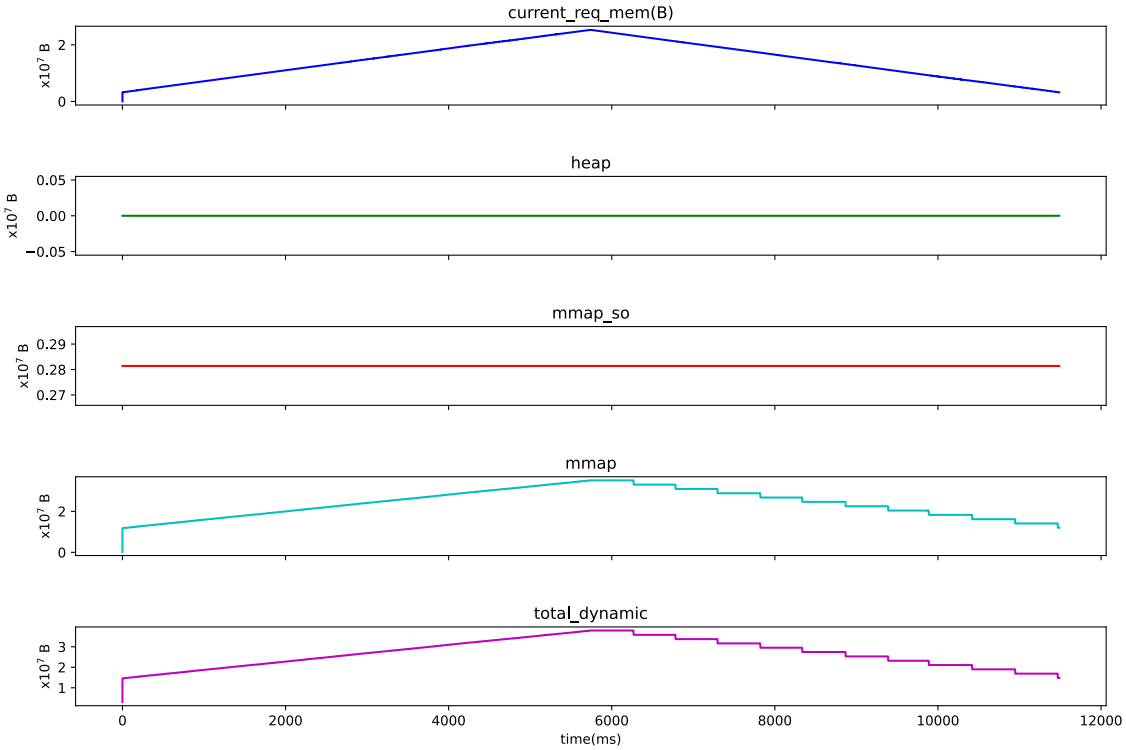


(a) Algol

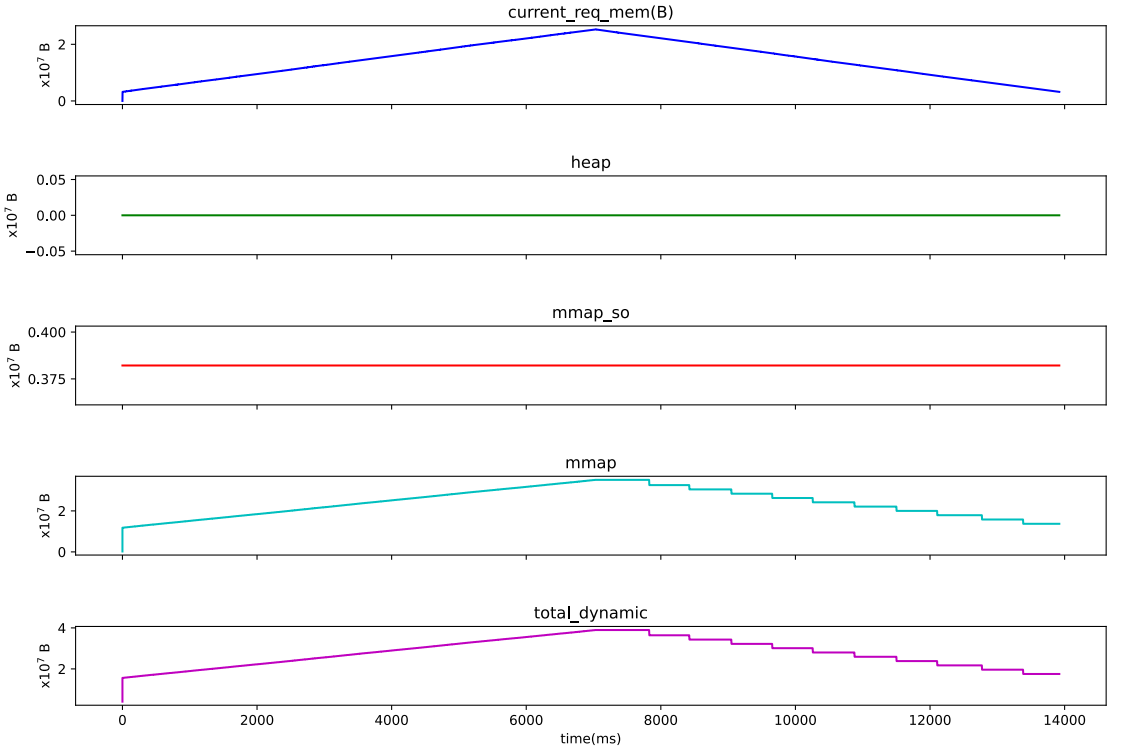


(b) Nasus

Figure 5.20: Memory benchmark results with Configuration-1 for je memory allocator

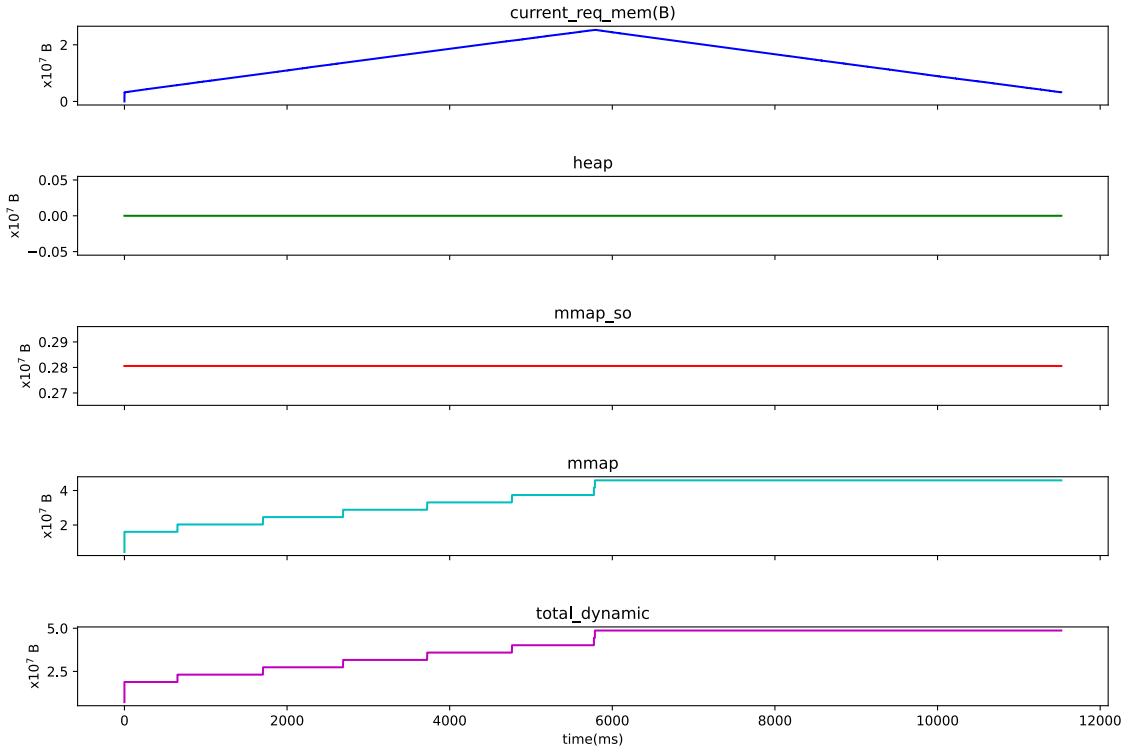


(a) Algol

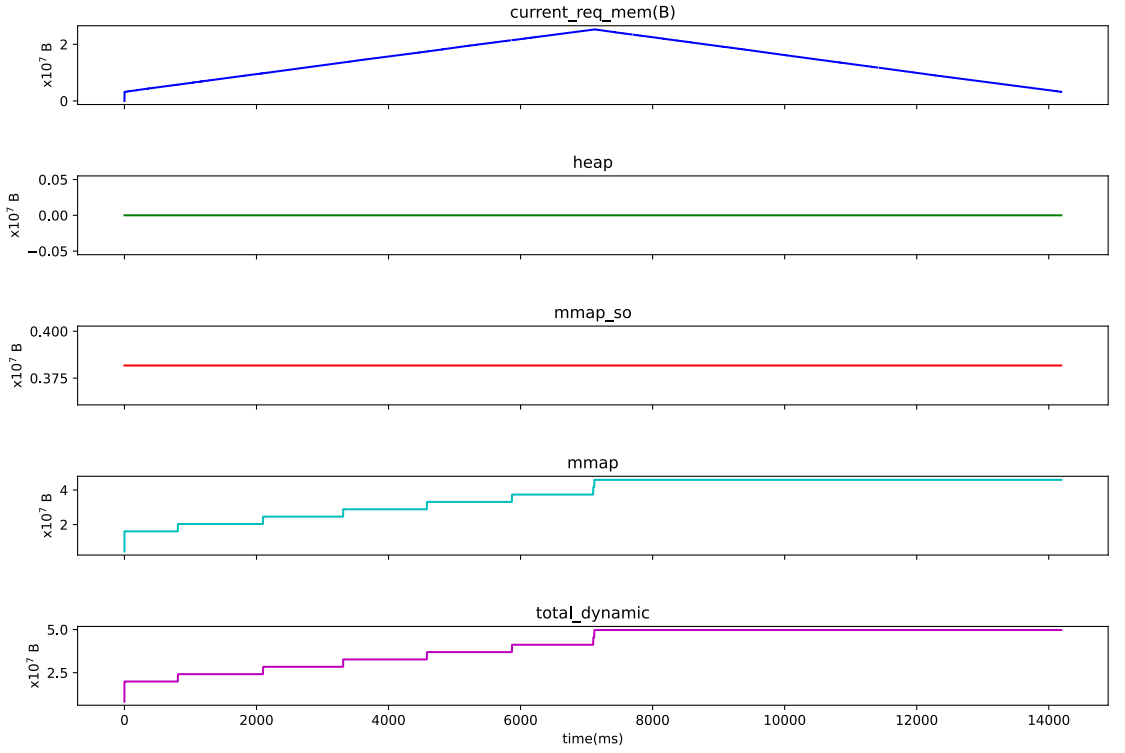


(b) Nasus

Figure 5.21: Memory benchmark results with Configuration-1 for pt3 memory allocator

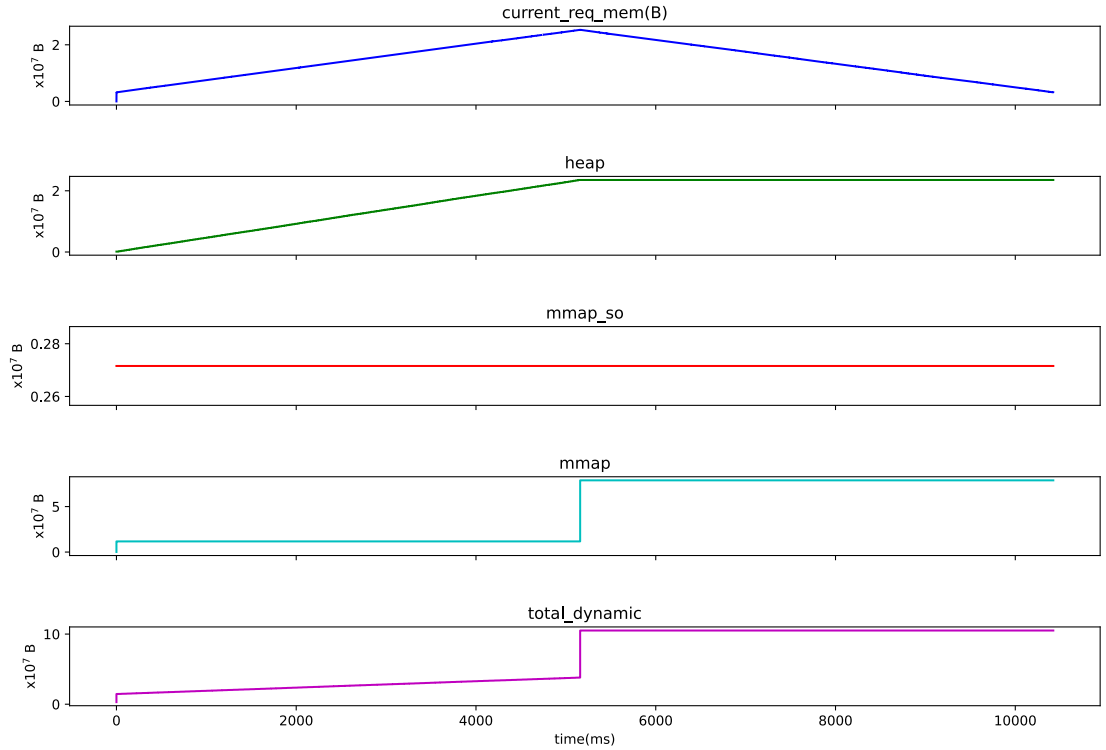


(a) Algol

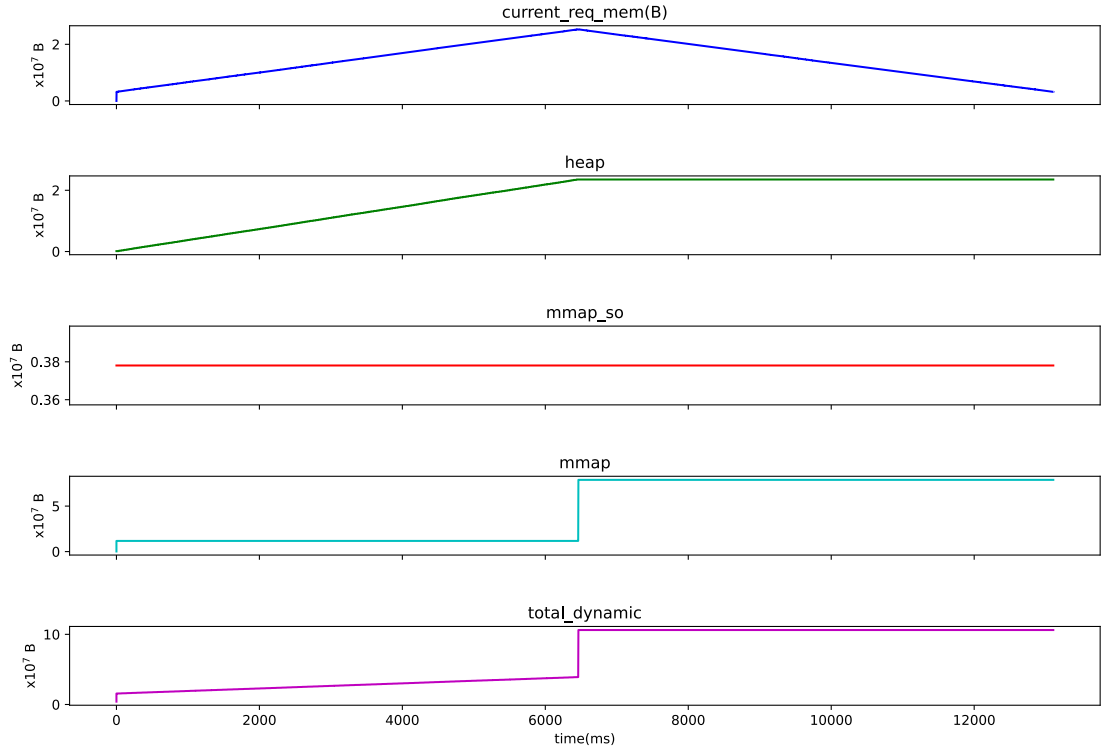


(b) Nasus

Figure 5.22: Memory benchmark results with Configuration-1 for rp memory allocator

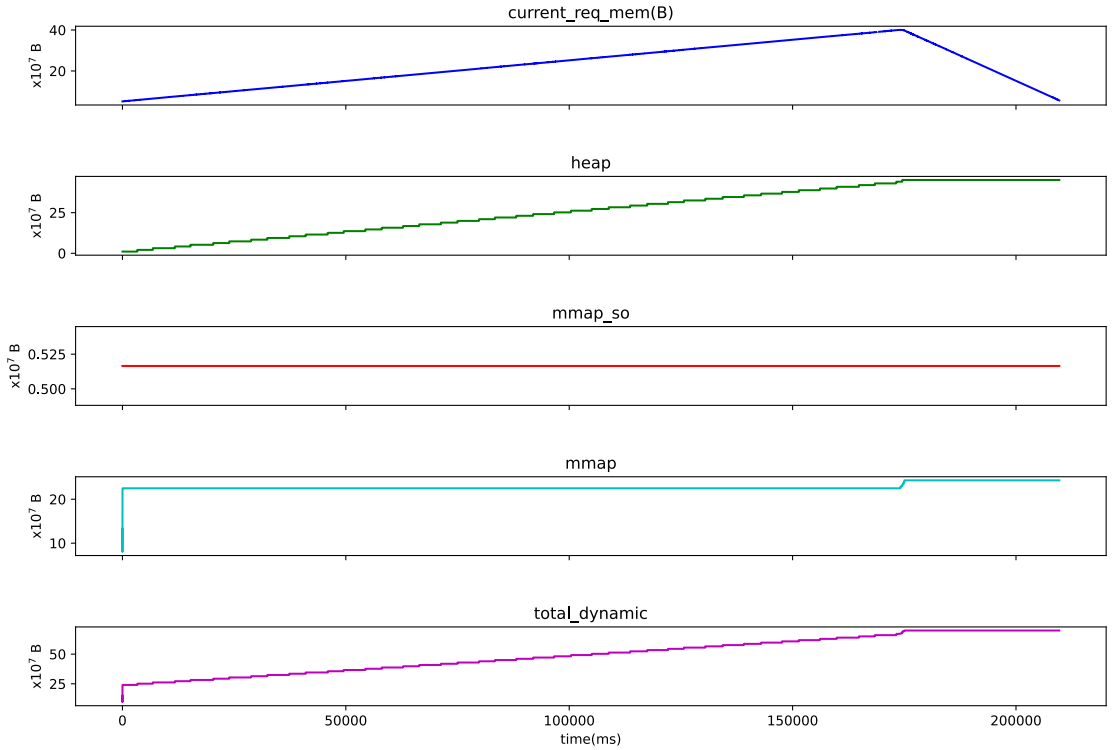


(a) Algol

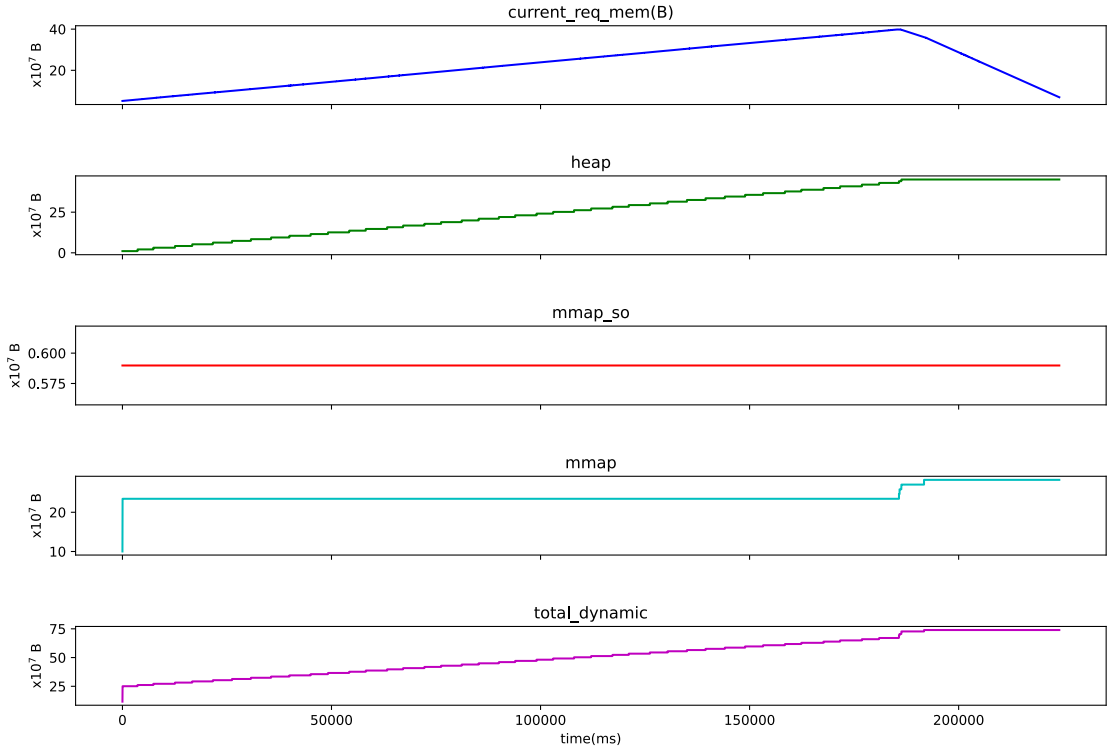


(b) Nasus

Figure 5.23: Memory benchmark results with Configuration-1 for tbb memory allocator

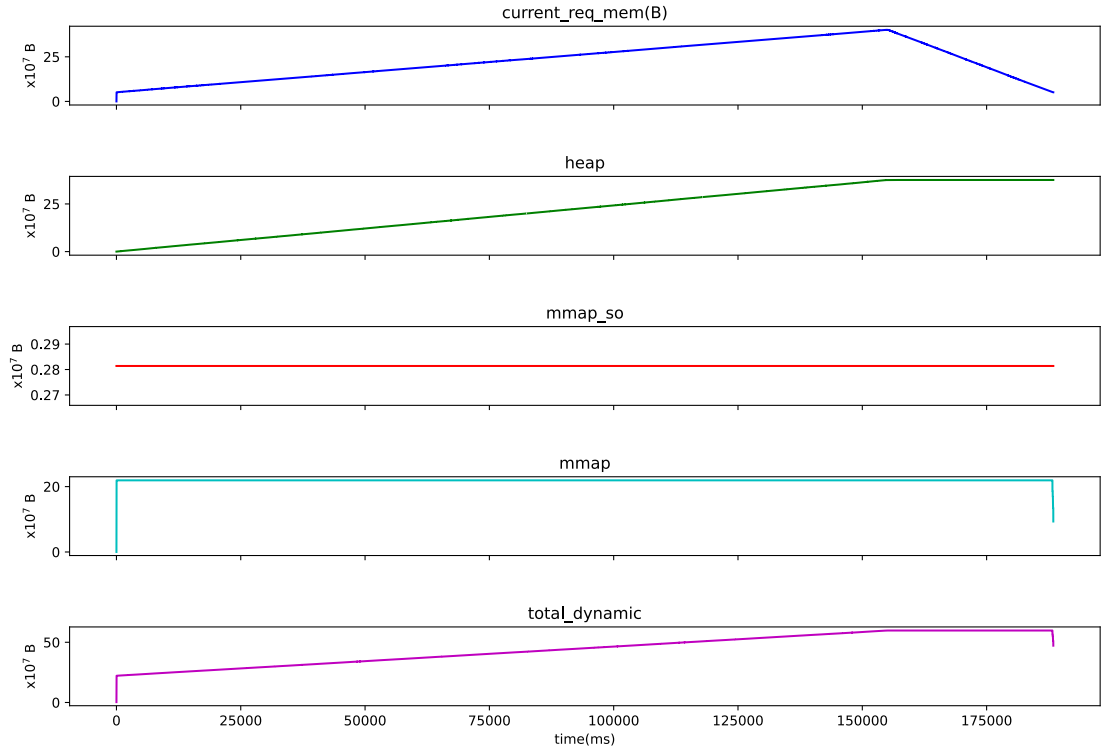


(a) Algol

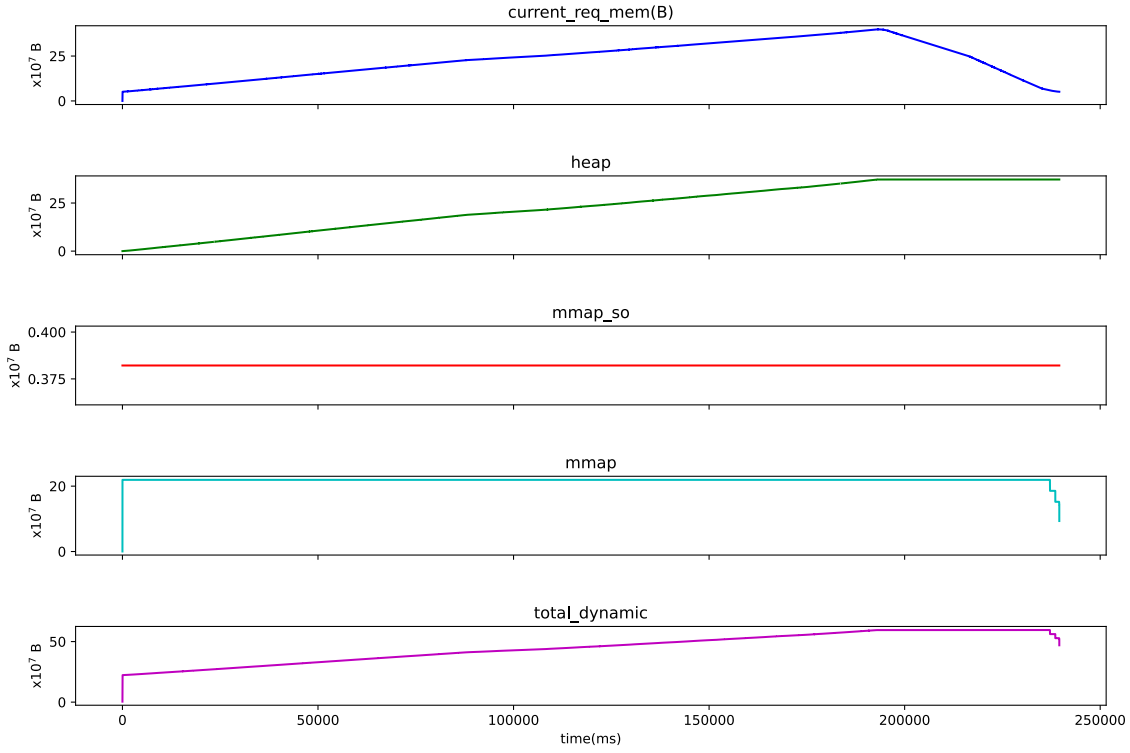


(b) Nasus

Figure 5.24: Memory benchmark results with Configuration-2 for llh memory allocator

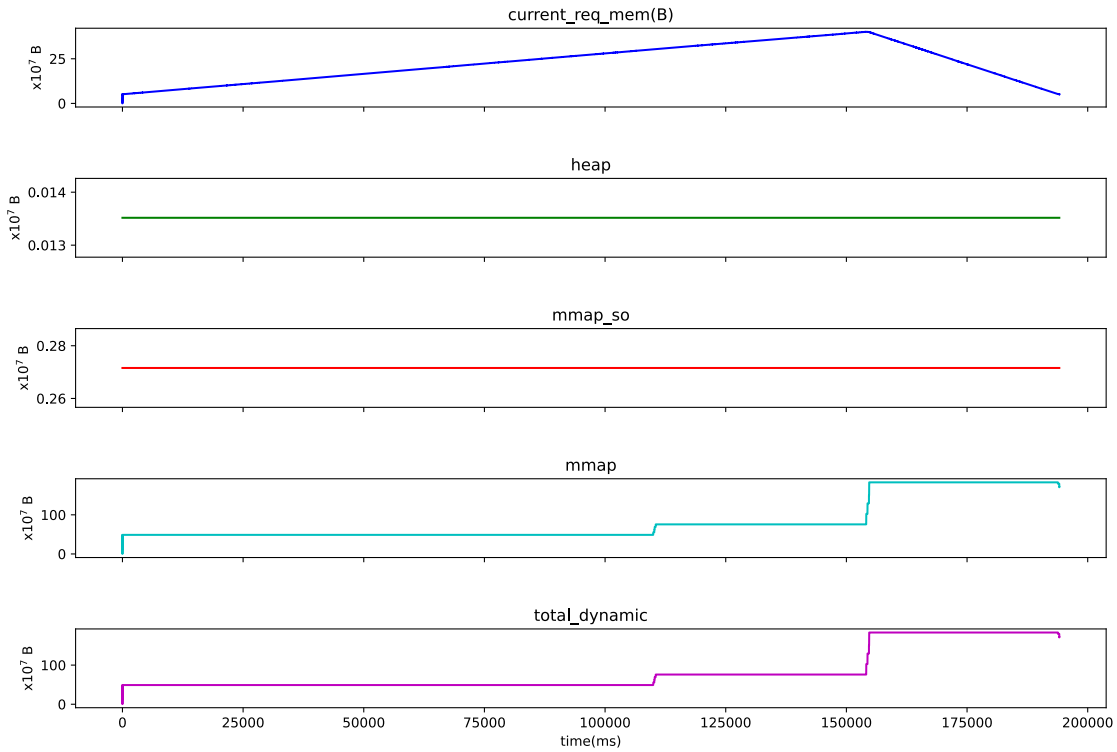


(a) Algol

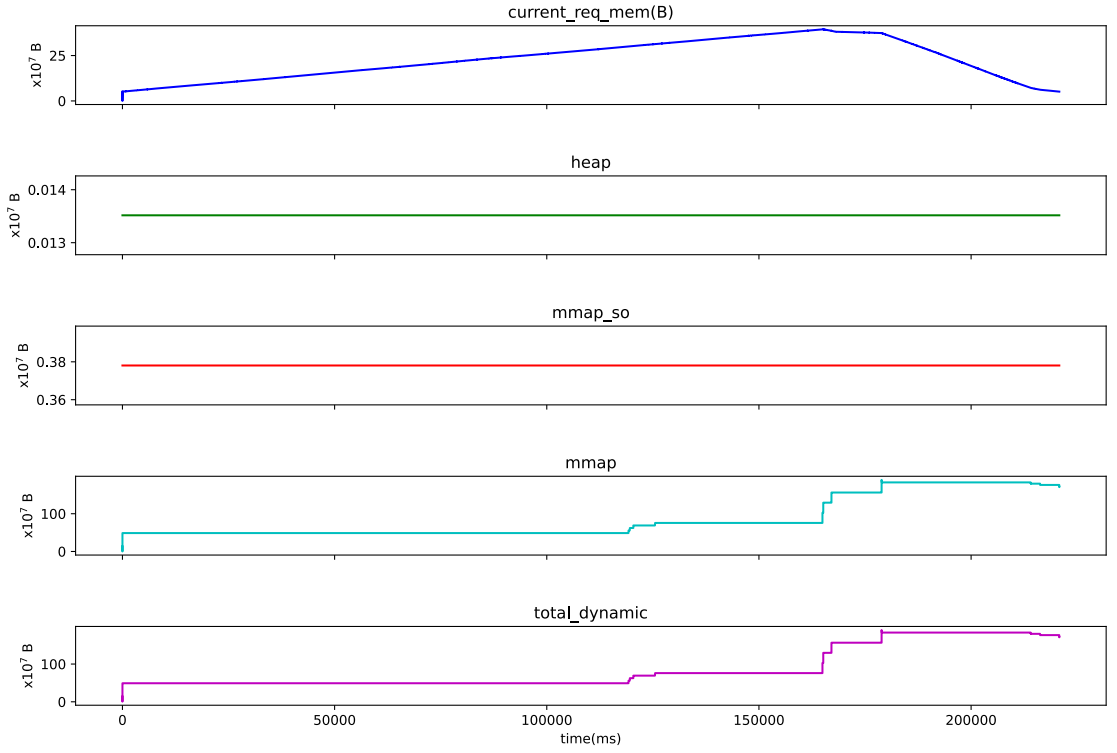


(b) Nasus

Figure 5.25: Memory benchmark results with Configuration-2 for dl memory allocator

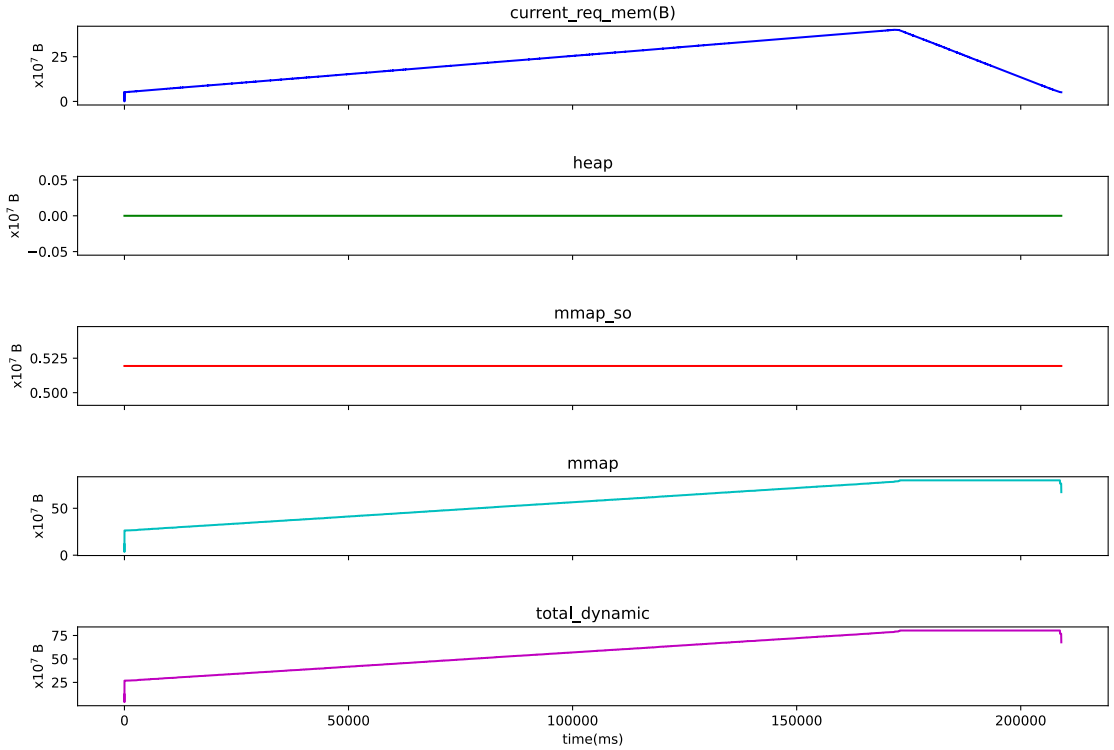


(a) Algol

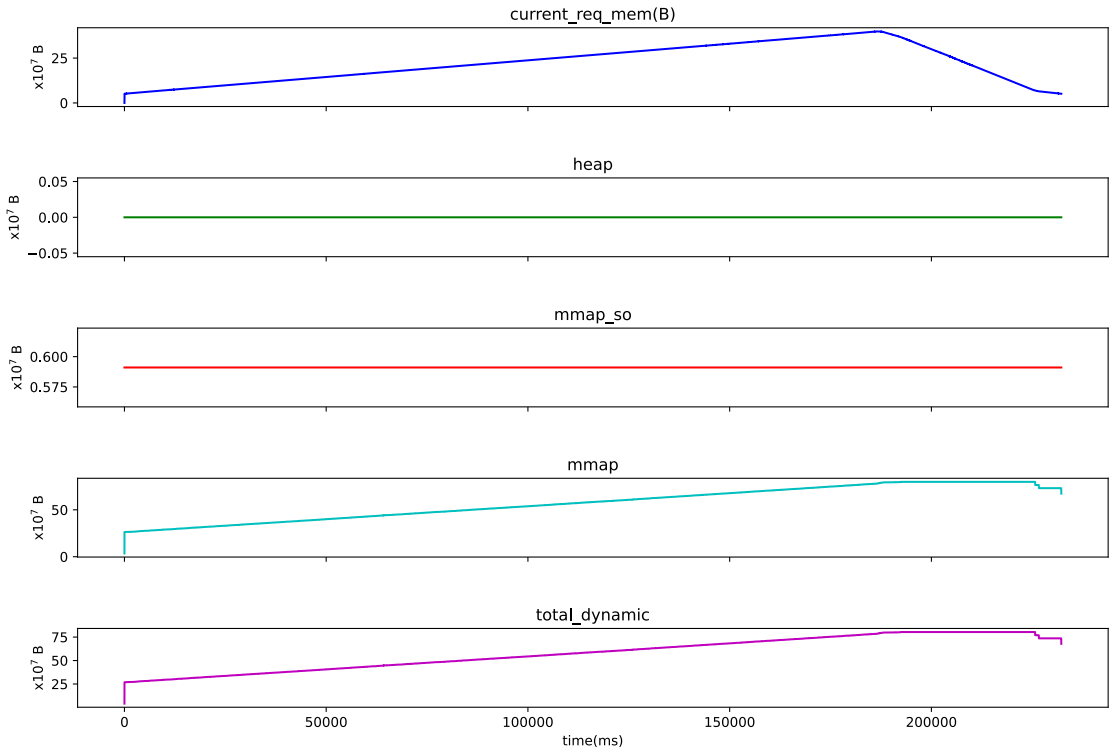


(b) Nasus

Figure 5.26: Memory benchmark results with Configuration-2 for glibc memory allocator

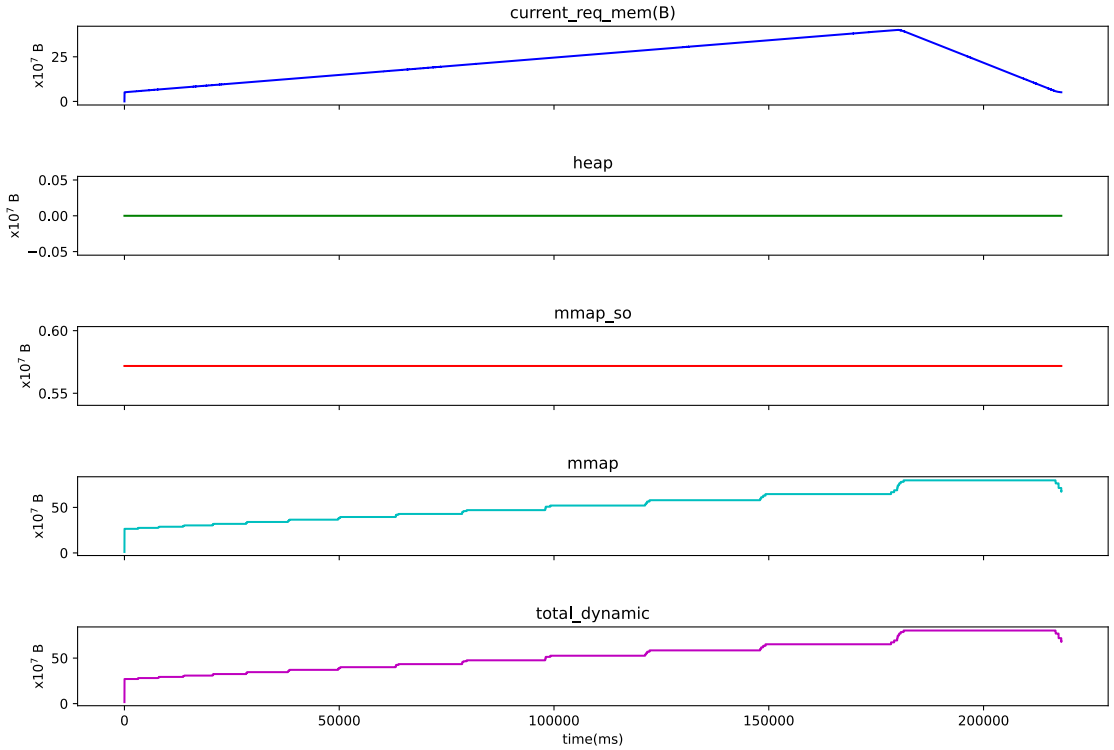


(a) Algol

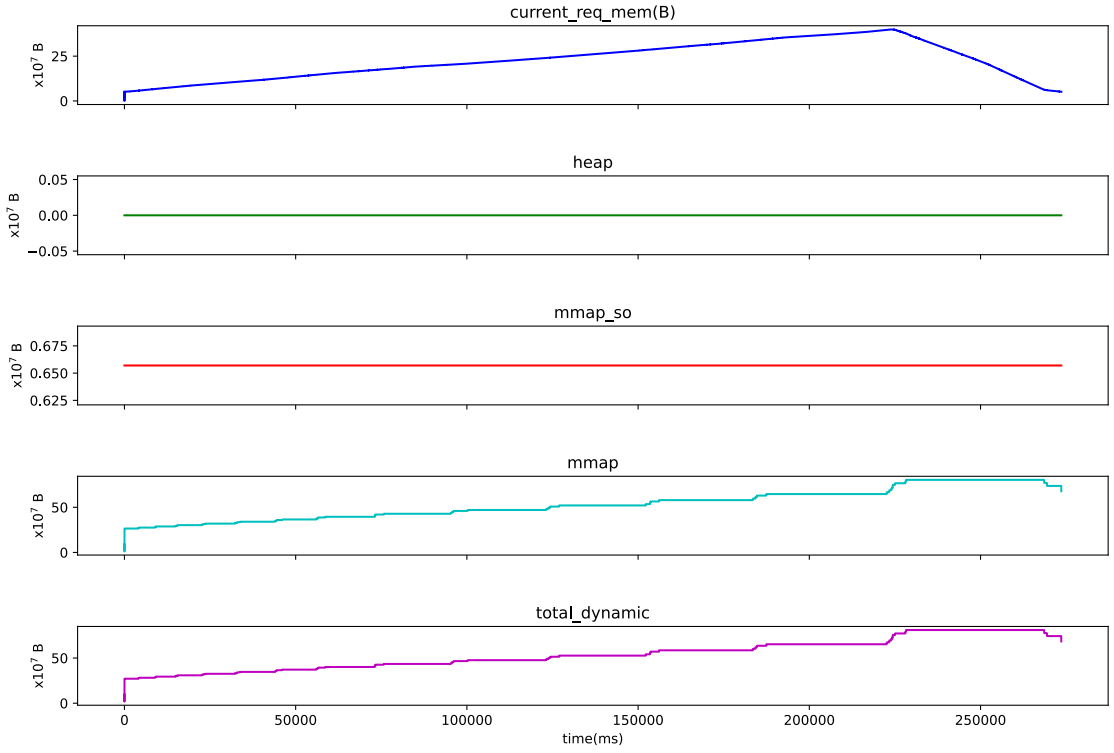


(b) Nasus

Figure 5.27: Memory benchmark results with Configuration-2 for hoard memory allocator

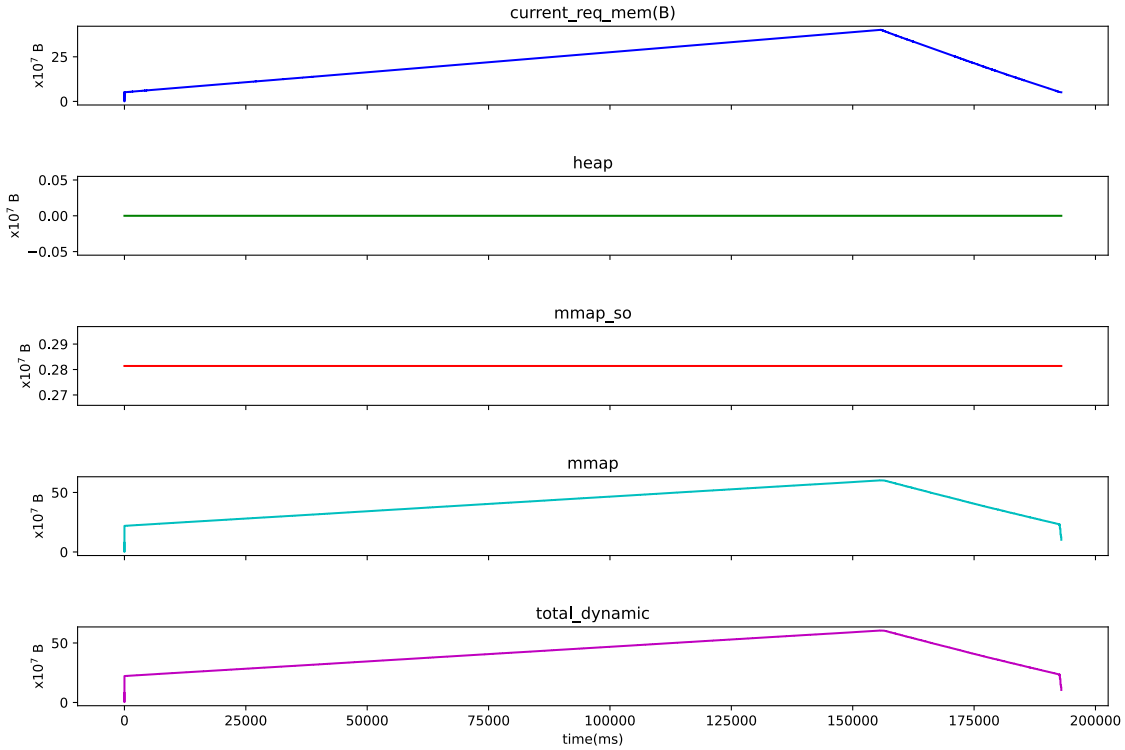


(a) Algol

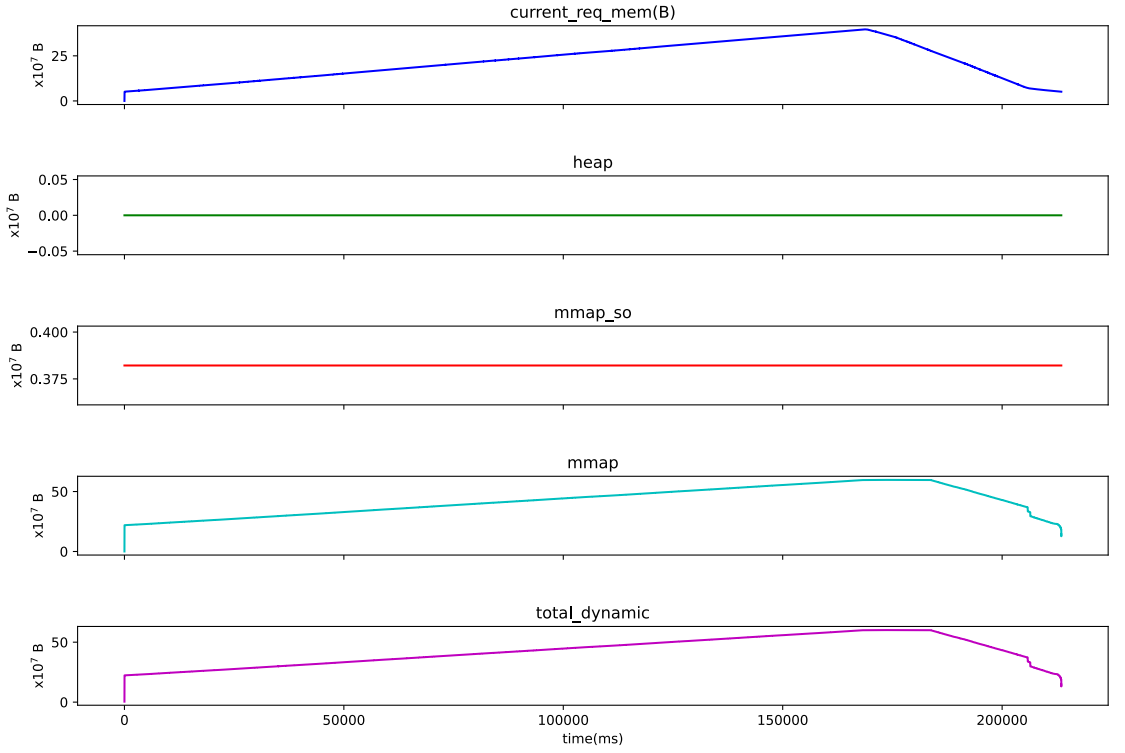


(b) Nasus

Figure 5.28: Memory benchmark results with Configuration-2 for je memory allocator

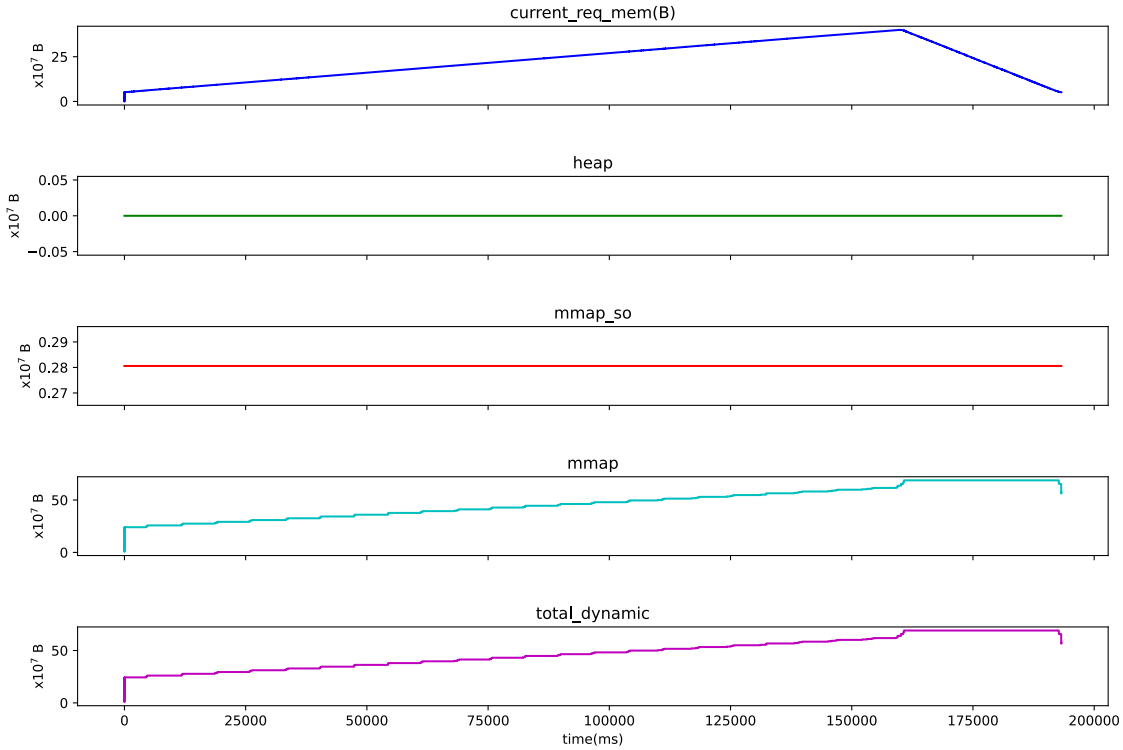


(a) Algol

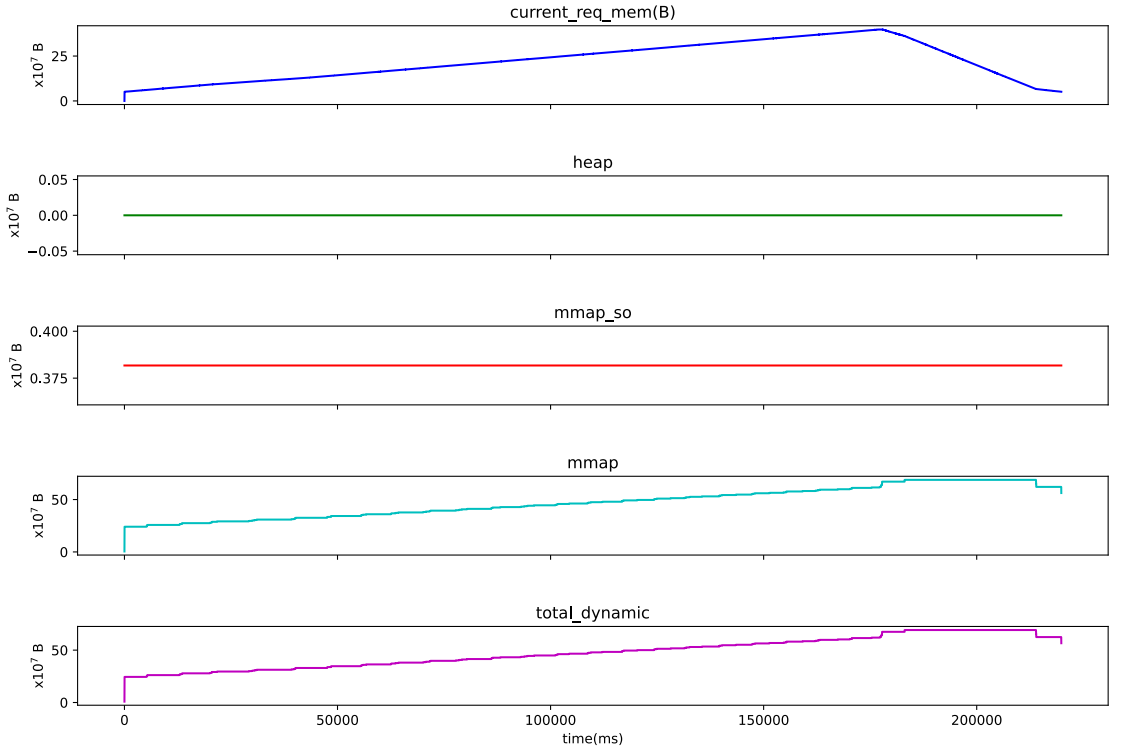


(b) Nasus

Figure 5.29: Memory benchmark results with Configuration-2 for pt3 memory allocator

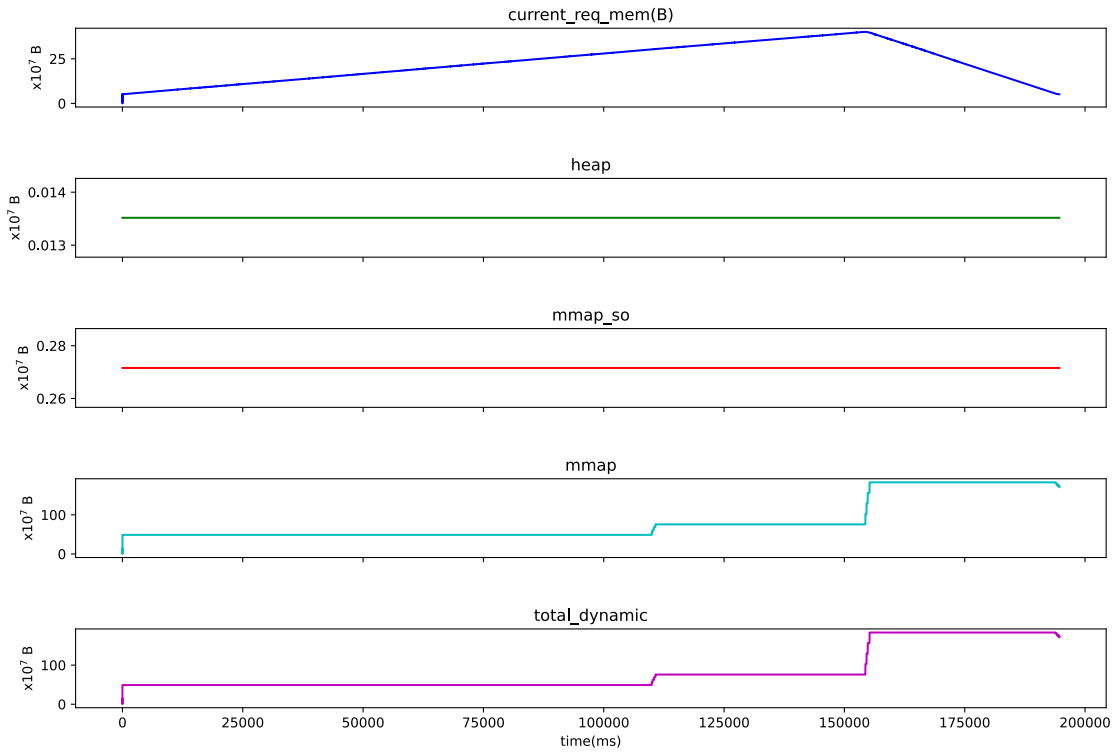


(a) Algol

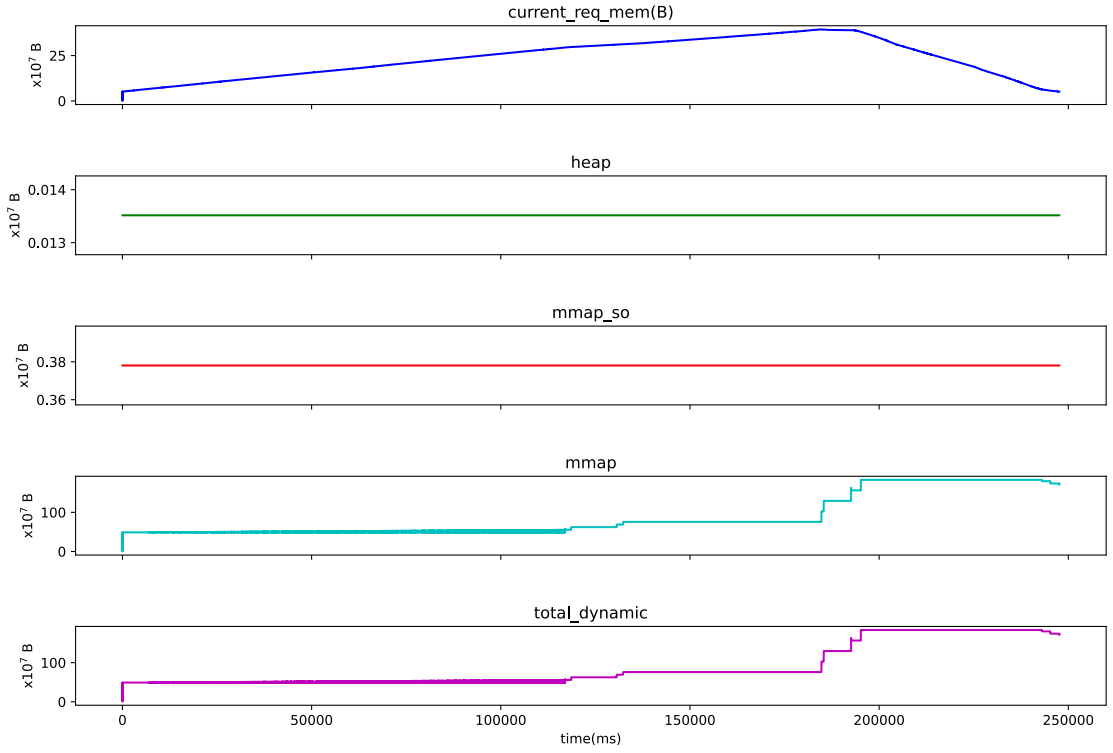


(b) Nasus

Figure 5.30: Memory benchmark results with Configuration-2 for rp memory allocator



(a) Algol



(b) Nasus

Figure 5.31: Memory benchmark results with Configuration-2 for tbb memory allocator

Chapter 6

Conclusion

The goal of this thesis was to build a low-latency (or high bandwidth) memory allocator for both KT and UT multi-threading systems that is competitive with the best current memory allocators while extending the feature set of existing and new allocator routines. The new llheap memory-allocator achieves all of these goals, while maintaining and managing sticky allocation information without a performance loss. Hence, it becomes possible to use realloc frequently as a safe operation, rather than just occasionally. Furthermore, the ability to query sticky properties and information allows programmers to write safer programs, as it is possible to dynamically match allocation styles from unknown library routines that return allocations.

Extending the C allocation API with resize, advanced realloc, aalloc, amemalign, and cmemalign means programmers do not have to do these useful allocation operations themselves. The ability to use C∀'s advanced type-system (and possibly C++'s too) to have one allocation routine with completely orthogonal sticky properties shows how far the allocation API can be pushed, which increases safety and greatly simplifies programmer's use of dynamic allocation.

Providing comprehensive statistics for all allocation operations is invaluable in understanding and debugging a program's dynamic behaviour. No other memory allocator provides such comprehensive statistics gathering. This capability was used extensively during the development of llheap to verify its behaviour. As well, providing a debugging mode where allocations are checked, along with internal pre/post conditions and invariants, is extremely useful, especially for students. While not as powerful as the valgrind interpreter, a large number of allocation mistakes are detected. Finally, contention-free statistics gathering and debugging have a low enough cost to be used in production code.

The ability to compile llheap with static/dynamic linking and optional statistics/debugging provides programers with multiple mechanisms to balance performance and safety. These allocator versions are easy to use because they can be linked to an application without recompilation.

Starting a micro-benchmark test-suite for comparing allocators, rather than relying on a suite of arbitrary programs, has been an interesting challenge. The current micro-benchmarks allow some understanding of allocator implementation properties without actually looking at the implementation. For example, the memory micro-benchmark quickly identified how several of the allocators work at the global level. It was not possible to show how the micro-benchmarks adjustment knobs were used to tune to an interesting test point. Many graphs were created and discarded until a few were selected for the thesis.

6.1 Future Work

A careful walk-though of the allocator fastpath should yield additional optimizations for a slight performance gain. In particular, analysing the implementation of rpmalloc, which is often the fastest allocator,

The micro-benchmark project requires more testing and analysis. Additional allocation patterns are needed to extract meaningful information about allocators, and within allocation patterns, what are the most useful tuning knobs. Also, identifying ways to visualize the results of the micro-benchmarks is a work in progress.

After llheap is made available on GitHub, interacting with its users to locate problems and improvements will make llbench a more robust memory allocator. As well, feedback from the μC++ and C$\forall$ projects, which have adopted llheap for their memory allocator, will provide additional information.

References

[1] Emery D. Berger. hoard version 3.13, April 2022. https://github.com/emeryberger/Hoard. 55

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). 2, 6, 16, 46, 47

[3] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 114–124, 2001. 46

[4] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, New York, NY, USA, November 2002. ACM. 46

[5] Steve Blackburn, Robin Garner, and Daniel Frampton. *MMTk: The Memory Management Toolkit*, September 2006. http://cs.anu.edu.au/∼Robin.Garner/mmtk-guide.pdf. 2

[6] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988. 2

[7] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005. 7

[8] Mathieu Desnoyers. Library for restartable sequences, March 2022. https://github.com/-compudj/librseq. 27

[9] David L. Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado, 130 Lytton Avenue, Palo Alto, CA 94301 and Campus Box 430, Boulder, CO 80309, 1993. 46

[10] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd international symposium on Memory management (ISMM'02)*, pages 163–174, New York, NY, USA, June 2002. ACM. 14

[11] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Boston, 1st edition, 1990. 2

[12] Yi Feng and Emery D. Berger. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 Workshop on Memory System Performance*, pages 68–77, New York, NY, USA, June 2005. ACM. 8, 16

[13] Sanjay Ghemawat and Paul Menage. tcmalloc version 1.5, January 2010. http://google-perftools.googlecode.com/files/google-perftools-1.5.tar.gz. 2

[14] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating memory in a lock-free manner. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Algorithms – ESA 2005*, volume 3669 of *Lecture Notes in Computer Science*, pages 329–342. Springer, New York, 2005. 23

[15] Wolfram Gloger. ptmalloc version 3, May 2006. http://www.malloc.de/malloc/-ptmalloc3-current.tar.gz. 55

[16] GNU. Summary of malloc-related functions, 2020. https://www.gnu.org/software/libc/-manual/html_node/Summary-of-Malloc.html. 24

[17] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, 2nd edition, 2000. 2

[18] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, 2009. http://golang.org/ref/spec. 2

[19] Dirk Grunwald, Benjamin G. Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–186, 1993. 8

[20] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *Transactions on Programming Languages and Systems*, 15(5):745–770, 1993. 23

[21] Xianglong Huang, Brian T Lewis, and Kathryn S McKinley. Dynamic code management: Improving whole program code locality in managed runtimes. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 133–143, New York, NY, USA, 2006. ACM Press. 1

[22] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *SIGPLAN Not.*, 27(5):T1–53, May 1992. 2

[23] IBM. Serially reusable programs, March 2021. https://www.ibm.com/docs/en/ztpf/-1.1.0.15?topic=structures-serially-reusable-programs. 26

[24] Mattias Jansson. rpmalloc version 1.4.1, April 2022. https://github.com/mjansson/-rpmalloc. 55

[25] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? *SIGPLAN Not.*, 34(3):26–36, 1999. 7

[26] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, 2nd edition, 1988. 2

[27] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. *SIGPLAN Not.*, 34(3):176–185, 1999. 46

[28] Doug Lea. dlmalloc version 2.8.4, May 2009. ftp://g.oswego.edu/pub/misc/malloc.c. 55

[29] C++ Standard Library. fister_f_distribution, April 2022. https://www.cplusplus.com/-reference/random/fisher_f_distribution. 47

[30] C++ Standard Library. uniform_int_distribution, April 2022. https://en.cppreference.com/w/-cpp/numeric/random/uniform_int_distribution. 47

[31] Tian F. Lim, Przemyslaw Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 118–129, New York, NY, USA, 1998. ACM Press. 6

[32] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, volume 39, pages 35–46, New York, NY, USA, June 2004. ACM. 23

[33] mtmalloc, 2009. http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/-libmtmalloc/common/mtmalloc.c. 2

[34] multiple contributors. glibc version 2.31, February 2020. https://www.gnu.org/software/-libc. 54

[35] multiple contributors. Threading building blocks, March 2020. https://github.com/-oneapi-src/oneTBB/releases/tag/v2020.2. 55

[36] multiple contributors. jemalloc version 5.2.1, April 2022. https://github.com/jemalloc/-jemalloc. 55

[37] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99)*, pages 182–204, Sendai, Japan, 1999. World Scientific. 9

[38] *Rust Programming Language*, 2015. https://doc.rust-lang.org/reference.html. 2

[39] Peter Jay Salzman. Memory layout and the stack. http://dirac.org/linux/gdb/-02a-Memory_Layout_And_The_Stack.php. 1

[40] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *International Symposium on Memory Management (ISSM'06)*, pages 84–94, New York, NY, USA, June 2006. ACM. 2, 46

[41] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 9–17, New York, NY, USA, 2000. ACM Press. 6

[42] Guy Steele. *COMMON LISP: The Language*. Digital Press, New York, 1984. 2

[43] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *J. Parallel Distrib. Comput.*, 68(7):1008–1020, 2008. 23

[44] John D. Valois. Implementing lock-free queues. In *Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, Nevada, U.S.A., 1994. 23

[45] Ayelet Wasik. Features of a multi-threaded memory allocator. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, January 2008. http://-uwspace.uwaterloo.ca/bitstream/10012/3501/1/Thesis.pdf. 4

[46] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer, September 1992. 2

[47] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland, UK, 1995. 6