Brug: an Adaptive Memory (Re-)Allocator

Weng, W.; Uta, A.; Rellermeijer, J.S.

2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 979-8-3503-9566-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/CCGrid59990.2024.00017

Brug: An Adaptive Memory (Re-)Allocator

Weikang Weng
LIACS, Leiden University
w.weng@liacs.leidenuniv.nl

Alexandru Uta
DFINITY, Zürich
alexandru.uta@gmail.com

Jan S. Rellermeyer
Leibniz University Hannover
rellermeyer@vss.uni-hannover.de

***Abstract*—Although memory allocation is well-studied, it is far from being a solved problem. There exist many allocators, each offering varied performance depending on the underlying workload. With workloads becoming ever more complex, practitioners need to take difficult decisions for the performance tuning of memory allocation: which allocators to choose and how to tweak their knobs are legitimate questions.**

In this article, we take a deep look at memory allocators and propose Brug, an adaptive memory allocator that builds upon the strengths of all existing allocators and discards their weaknesses. Brug can help programmers choose the suitable allocator for their applications or even for individual data structures and functions within applications, allowing for different allocators within the same program. Brug also offers an auto-tuner to minimize developer decision-making.

Brug comes in two flavors: (1) Rust-based library that can be added to modern Rust code bases, helping in allocation and re-allocation performance and diagnosis. (2) C-based library that can be dynamically linked at runtime for existing legacy programs to optimize their performance. Brug was deployed with industry standard-grade frameworks, such as Apache Arrow, Wasmtime WebAssembly virtual machine, and Redis. Our experiments show that Brug can improve performance in all types of applications and help developers toward taking otherwise difficult decisions. Brug consistently improves application execution time.

I. Introduction

Memory (mis-)management is a major (performance) limiting factor for demanding applications in growth areas like big data [1], [2], graph processing [3], or cloud-based workloads [4], [5]. Such applications use various dynamic heap objects [6] backed by memory allocators, software interfaces that manage memory usage. Different memory allocators follow different designs and leads to different performance characteristics. In Figure 1, we show the lifetime of an object being allocated and then re-allocated several times during the runtime of a program. To achieve optimal performance, four different allocators have to be used. This optimal result is impossible to achieve in practice without adaptive allocators and is difficult to predict.

Choosing the optimal memory allocator for the workload is often difficult for programmers and demands knowledge, experience and ultimately large amount of experimentation. Suboptimal allocator choice may lead to long-term performance loss. In this work, we present the design and implementation of Brug, an adaptive and efficient memory (re-)allocator. Brug builds upon the strengths of all existing allocation techniques but discards their weaknesses, helping developers to achieve good allocation and re-allocation performance.

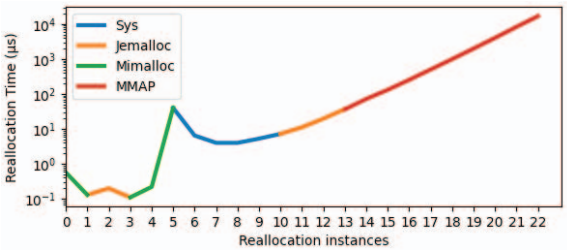

Fig. 1. Best re-allocation performance is achieved when a combination of allocators is used during runtime. Individual allocation performance is colored differently per allocator type to showcase this unexpected behavior.

Brug fills the existing gap in current memory allocation techniques—adapting to workload demand during runtime. Heap-based memory allocators are usually efficient but lack performance with multi-threaded allocation workloads. To solve this issue, previous research primarily focuses on allocation response time for new objects and reconciling multi-threading interaction with allocators. We identified 4 different approaches from literature: (1) fast multi-threaded techniques [7], (2) reducing locking overhead [8], (3) improving cache miss rate [9] and TLB efficiency [10], or (4) avoiding context switch overhead by using the memory pool model and moving the memory controller into user space.

Most of these allocators generally use a sequence called *malloc-copy-free* when it comes to subsequent re-allocations of objects. This sequence works as follows: (1) allocate a new memory object with the new size, (2) copy the old content into the new object, and (3) de-allocate the old object. In modern systems, these operations have a high potential for adding additional (performance) pressure on DRAM and CPU. Furthermore, with large-size memory objects [11]–[13] that are touched by concurrent threads in modern and emerging workloads, such as big data, databases, or modern data formats [14]–[16], performance degradation caused by copying (and page-faulting) can be significant.

In this paper, we take a systematic approach to assess different allocators and their impact on data structures and memory workloads. For example, in an experiment using the C++ Vector, we measured four different memory allocators: Jemalloc [17], Mimalloc [18], TCmalloc [19], Ptmalloc2 [20] and the MMAP-MREMAP [21] approach. We observed that different allocators could make up to a 10% difference in performance for the vector push operation, and even 20% in multi-threaded operation.

Tapping into this performance difference, we designed and

979-8-3503-9566-2/24/$31.00 ©2024 IEEE
DOI 10.1109/CCGrid59990.2024.00017

implemented the Brug allocator, which helps programmers by reducing the overhead of memory (re-)allocation and improving performance in modern code bases (e.g., Rust [1], as well as in legacy applications (which cannot be recompiled) with C. We chose Rust as a target for Brug as its developer community is growing rapidly and the language is gaining more and more attention in system programming [22], [23]. By using Brug in Rust programs, programmers can not only add different allocators for different data structures, but also mix and change them during program runtime. Moreover, Brug offers an auto-tuner which is able to closely follow the optimal allocator for every (re-)allocation decision. Our contributions are:

1. An in-depth analysis of existing memory allocation mechanisms. We provide an overview of different mechanisms used in different allocators. We further investigate the re-allocation performance of widespread memory allocators built on these primitives. The allocators we consider are Ptmalloc2 in Glibc-2.31, Jemalloc-5.0, TCmalloc in gperftools-2.9.1 and MImalloc 2.0.6.

2. The design and implementation of Brug: our open-source, library-based adaptive memory allocator for modern Rust code bases and (legacy) C/C++-based applications. Brug is an adaptive, lightweight, portable and flexible library-based allocator. The Brug Rust version is designed to explore more possibilities in interacting with different data structures. It allows programmers to set specific memory allocators for different function blocks and dynamically switch, monitor and tweak memory settings during runtime. In addition, Brug provides a convenient and flexible auto-tuner, which helps programmers optimize allocator performance. For legacy programs the Rust version is impossible to use. Therefore, we provide the Brug C version, an easy-to-use dynamic linking library for legacy applications. It overrides the default allocator in legacy binaries the Linux LD_PRELOAD primitive [24]. After the override, Brug helps fix performance degradation because of expensive re-allocation copies.

3. The many-fold evaluation of Brug. Our findings demonstrate the versatility of Brug in handling various Rust data structures while incurring minimal overhead during the auto-tuner's training process. Notably, we showcase the seamless integration of Brug with Rust-native code bases. Our experiments target industry-grade state-of-the-art frameworks, such as Apache Arrow [25], which is a foundational component for data-centric systems, Wasmtime [26], an industry-grade WebAssembly [27] VM, as well as the Redis key-value store running write-heavy workloads from YCSB [28], [29].

Our findings confirm that Brug optimizes re-allocation performance and enables allocator switching with minimal overhead, while the auto-tuner can take accurate decisions for re-allocation operations. For Redis, across various configurations, Brug achieves considerable throughput improvement.

II. Memory Allocation Mechanisms

In modern operating systems, applications operate on virtual addresses, which are software-controlled sets of memory addresses. These addresses provide each process with a unique view of a computer's physical memory. The translation between virtual and physical addresses includes a process called memory mapping. Memory mapping involves two hardware components: the Memory Management Unit (MMU) and the Translation Lookaside Buffer (TLB). The results of mapping are stored inside the Operating System-managed page tables.

Programs do not interact directly with these low-level constructs for security and isolation reasons. Instead, they interact with memory allocators, a type of software designed to handle memory-related operations. In UNIX-derivatives like Linux, allocating dynamic memory objects involves calling *malloc* or *calloc*. Resizing already allocated objects during runtime uses the *realloc* [30] API. When objects are no longer needed, *free* releases them.

We study four popular allocators: the GNU allocator (Ptmalloc2) from Glibc [20], Tcmalloc from Google [19], Jemalloc from Meta [17], and Mimalloc from Microsoft [18]. Additionally, we include the *Mmap-mremap* primitives [21].

Ptmalloc2 is the default memory allocator in Glibc and is used by Linux. It relies on a *heap-based* approach, where new memory is allocated and placed on the process heap. When objects needs to change size, the *brk/sbrk* system call is used to set the break value to addr and change the allocated space accordingly. For very large blocks, i.e., much larger than a page, these requests are allocated with *mmap*. This allows chunks to be returned to the system immediately when freed, avoiding large chunks being locked in between smaller ones, leading to fragmentation [31].

***Mmap-mremap* primitives** use the mapping mechanism between the Linux virtual memory pages and physical memory (i.e., the page table). However, this incurs significant overhead through context switches and TLB shootdowns [32]. Additionally, it only works at page granularity (i.e., 4 KiB).

Tcmalloc, Jemalloc, and Mimalloc can be categorized as user-space allocators. They share the same vision of using a pre-allocated memory pool in user space, setting up thread caches and size classes for higher parallel allocation performance. Tcmalloc involves a three-level caching system: ThreadCache, CentralCache (Central Freelist), and PageHeap. Jemalloc shares ThreadCache but uses multiple arenas instead of a central free list in the middle layer. Mimalloc adds a free list for each thread called a free list multi-sharding. These designs avoid the context switch between userspace and the kernel, and the parallel design ensures they serve the allocation request from multi-thread applications.

On the re-allocation side, Tcmalloc and Mimalloc take a straightforward approach to the *malloc-copy-free* sequence between different size classes. Jemalloc added additional merging for re-allocation. It checks the neighbors of the current size class and tries to merge them if possible. For very large objects, *mremap* was considered but not adopted. As Jemalloc operates under userspace, the remapped virtual memory will leave memory holes that cannot be released back to the OS. The solution for this is using huge size classes (4, 8, 12 MiB) [33] and keeping the merging mechanism if possible.

[1] https://github.com/WayneWeng95/brug/

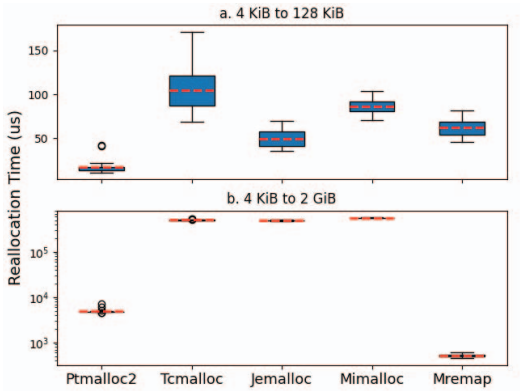

Fig. 2. Re-allocation time for different allocators with C dynamic array micro-benchmark experiment: **(a)**. The object grows from 4 KiB to 128 KiB. (**b**). The object grows from 4 KiB to 2 GiB. (Lower is better.)

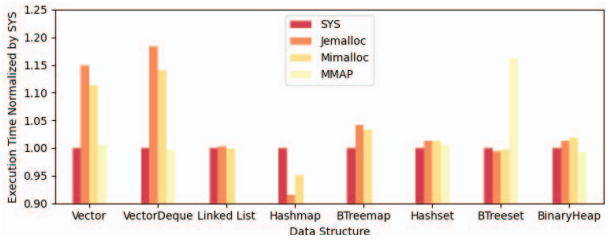

Fig. 3. The performance of Rust data structure allocation relative to the system allocator (SYS). Note the MMAP is not tested in linked list-based data structures (LinkedList, Hashmap, Btreemap) because of the internal fragmentation caused by page size. (The Y-axis indicates the execution time normalized by SYS. Results are mean value of 15 repetitions, lower is better.)

If the merge cannot be done or even for larger re-allocation, it falls back to the *malloc-copy-free* sequence.

To quantify the re-allocation performance among different allocators, we set up a simple vector size increase benchmark with C. It works as follows: we allocate a 4 KiB buffer and double its size with *realloc* inside a loop. After each re-allocation, we fill the buffer and touch all buffer elements. We measure the total re-allocation time for this entire operation.

We further divide this experiment into two scenarios: small and large objects. The former grows from 4 KiB to 128 KiB, while the latter grows from 4 KiB to 2 GiB. In the small objects group (Figure 2(a)), we observe that Ptmalloc2 achieves the best performance, followed by Jemalloc, and *mremap* takes the third position. In the large objects group (Figure 2(b)), the re-allocation costs for TCmalloc, Jemalloc, and Mimalloc are one order of magnitude higher compared to Ptmalloc2 and *mremap*. Between the latter two, Ptmalloc2 provides solid performance, but *mremap* performs best.

As the size increases, we observed a significant difference between copying and page table operations. When dealing with larger objects, more data must be copied from the old block to the new one for user-space allocators. An inherent side effect of this copying process is a substantial increase in overall memory usage before the old memory block is freed. Apart from the burst in memory costs, this copying operation also consumes considerable CPU cycles, posing a substantial performance issue.

Taking our analysis one step further, we examined the performance of different allocators with Rust data structures from the standard collection. The results are represented in Figure 3. The collections can be grouped into four categories: (1) Sequences: Vector, VectorDequeue, and Linked List; (2) Maps: Hashmap and Btreemap; (3) Sets: Hashset and Btreeset; (4) Misc: BinaryHeap.

Among the three sequences, Vector and Vectordequeue are implementated using arrays and exhibit different characteristics compared to the linked list. With the two maps, Hashmap is backed by the LinkedList, while Btreemap is backed by an array. HashSet and Btreeset are set versions of Hashmap and Btreemap. The Binary heap is simply an array.

We observed that different allocators do exhibit a performance preference for different data structures, and the performance difference could be as high as 20%.

Observations: 1. Various allocators exhibit distinct performance characteristics in vector re-allocations, with significant differences observed. 2. User-space allocators experience noticeable performance degradation during re-allocations, primarily due to the copying cost associated with the *malloc-copy-free* process in large objects. 3. In multi-threaded scenarios, the copy overhead becomes even more pronounced when multiple objects undergo simultaneous *realloc* operations. 4. Different allocators showcase preferences for specific data structures, indicating varied suitability across different use cases.

Summarizing the allocators analyzed above, three mechanisms for implementing a *realloc* function are identified:

1) **brk/sbrk system call**: Moving the program break to expand or contract the memory area.
2) **malloc-copy-free sequence**: Allocating new chunks and copying existing contents.
3) **mmap-mremap primitive**: Remapping virtual address with page tables.

A. brk/sbrk *system call in Ptmalloc2*

The *brk/sbrk* system call in Glibc's ptmalloc2 represents the classic *heap-based* memory scheme [34] (Figure 4(a)). When the memory object grows, it moves the barrier, i.e., the program break. Specifically, the *brk* system call moves with the actual address, while *sbrk* moves with size increments.

In the ideal scenario, the *brk/sbrk* system call will consistently encounter soft page faults. The page fault handler quickly attaches new virtual pages after this memory object, notifying the MMU and TLB for further physical memory mappings. In this situation, the performance is relatively good, as shown in Figures 2 and 3.

However, the heap approach faces challenges in certain expansion scenarios. Ptmalloc2 stacks objects on the heap compactly, saving space but paying less attention to re-allocation. For objects frequently re-allocated (likely blocked by other objects) or exceeding arena limits (64 MiB in a 64-bit system), Ptmalloc2 is forced into a *malloc-copy-free* sequence, leading to performance degradation.

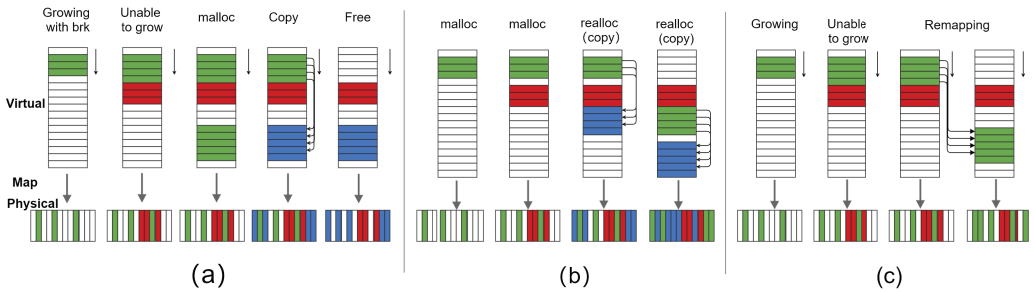

Fig. 4. Three different (re-)allocation mechanisms: **(a)** Heap using *brk/sbrk* to expand sizes. When the growing is blocked, one must perform a *malloc-copy-free* sequence to create space. **(b)** The process of a *malloc-copy-free*. Every re-allocation equals a copy of existing content to a larger memory space. **(c)** The process of *mmap-mremap*. Notice that the remapping only applies to the virtual memory space but does not lead to physical memory changes like in the previous approaches.

TABLE I
Brk/sbrk-TRIGGERED COPIES IN A MULTI-THREADED ENVIRONMENT. THE EXPERIMENT WAS REPEATED 30 TIMES.

Number of threads	1	2	4	8
Average copies per thread	5	13.5	49.75	61.75

Another issue with the heap implementation arises in multi-threaded environments. Threads sharing the same heap in the same process can cause different objects to potentially block each other during expansion. The number of memory copies increases, and the performance impact is detailed in Table I, indicating a significant increase in copies with the number of threads. Although the arena concept has been introduced to address this problem [31], complex workloads can lead to more frequent copies than anticipated.

Downside 1: *Brk/sbrk* is bundled with the heap design, but the heap cannot grow infinitely. Therefore there is always the risk of touching other objects and making expensive copies. This design exacerbates performance overhead in multi-threaded environments.

B. Malloc-copy-free *in fast user-space allocators*

Multi-threaded workloads pose a challenge as the heap becomes a bottleneck due to concurrent memory operations. To address this challenge, user-space memory management utilizes pre-allocated memory pools, allowing processes to maintain their own pools. Notable user-space allocators, such as Jemalloc, TCmalloc, and Mimalloc, implement different policies to optimize memory allocation.

However, a drawback of the user-space memory-pool design is that it makes *heap-growth realloc* unavailable, thereby forcing allocators to choose the *malloc-copy-free* sequence, illustrated in Figure 4(b). An additional source of overhead is that the approach requires an additional release process to return the memory back to the OS. Designing a proper pool release mechanism is still challenging.

While user-space memory-pool allocators work well with small objects and even demonstrate better performance due to fast allocations, the copy cost introduces substantial overhead in execution time and memory space when object sizes grow significantly, as illustrated in Figure 2(a) to Figure 2(b).

Downside 2: Multi-threaded user-space allocators use *malloc-copy-free* during *realloc*. This leads to performance penalties when object sizes grow. Combined with lazy release of memory, overall memory usage is higher.

C. Mremap

Unlike user-space memory re-allocation methods, *mremap* is rarely used due to its reputation as an expensive operation. During remapping, it triggers context switches and TLB shoot-down, leading to degraded performance and scalability [32].

The *mremap* system call [21] resizes memory mappings created by *mmap*. Figure 4(c) illustrates its mechanism. This modifies the mapping between virtual addresses and memory pages. *Realloc* can thus be implemented by changing the virtual address mapping in the page table without copying.

Mremap is considered expensive due to the significant work it entails. The process includes changing the MMU to re-link the existing physical page with the new virtual address and flushing the TLB to clean up the old links. There is also a limit to this approach; the minimum size of this operation is one memory page (i.e., 4 KiB on x86_64 Linux), causing internal fragmentation for small objects. Additionally, integrating it into user-space allocators, as we explained with Jemalloc, poses challenges. Despite these issues, the results in Figure 2 and 3 highlight its performance advantage compared to larger object *malloc-copy-free* in large object reallocation.

Downside 3: The *mremap* approach avoids the copy cost by using a remapping mechanism between virtual and physical memory. However, it incurs quite significant overhead (context switch and TLB shootdown) Moreover, it uses a minimum size of 4 KiB (i.e., standard page size).

Following from our analysis and experiments, we noticed that different allocators provide different re-allocation characteristics which emerge from different tradeoffs. For small-size memory objects, heap-based allocators can realloc faster and save space (single heap) under single-threaded environments. User-space allocators emphasize multi-threaded fast allocation performance but lead to larger space occupation. With larger size memory objects, the drawback from *malloc-copy-free* becomes more significant. To a certain extent, the copy cost

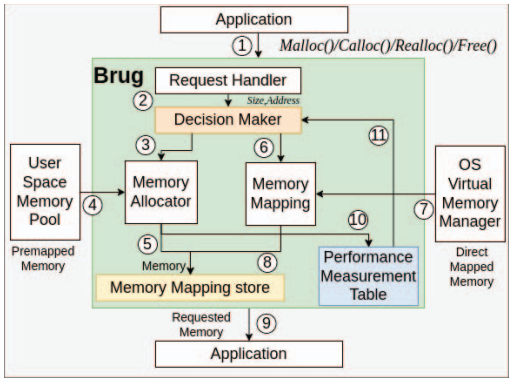

Fig. 5. Brug Allocator Design Overview: ① Application requests memory changes from the Brug allocator. The request handler receives the request. ② The request handler sends the request to the decision maker. ③ The Decision maker forwards the request to the allocator. ④ The allocator processes the request using its remapped memory pool. ⑤ The allocator returns the requested memory to the memory mapping store. ⑥ The Decision maker sends the request to the memory mapping. ⑦ The memory mapper manages the request with the OS Virtual Memory Manager. ⑧ The mapped memory is sent to the memory mapping store. ⑨ Brug delivers the memory to the user code based on the request. ⑩ The execution time of Memory allocators and Memory mapper is recorded in the performance measurement table.⑪ The Decision maker may update the policy based on the backlogs.

becomes a dominant factor in re-allocation overhead. This makes system-level page table modification attractive in large-size memory re-allocation with no copy cost and overcomes its problems like context switches and TLB shootdown. As a consequence, we argued and showed evidence that for the existing *realloc* mechanisms in allocators, there is significant opportunity for optimizations.

III. Brug Design and Implementation

We propose Brug, an adaptive memory allocator that optimizes for both efficient allocation and re-allocation. This is crucial for the performance of systems that tend to have large but dynamically growing working sets, as is the case for many data-centric systems.

Brug is designed as a library implemented in Rust and C. Brug takes advantage of the ability to allow different allocators during runtime and further includes a built-in auto-tuner mechanism to help programmers achieve good performance without resorting to large-scale experimentation to understand which allocator works best for given workloads. Brug Rust is available as a Rust crate, allowing for easy integration with existing code bases. The C version of Brug utilizes an adaptive mechanism that switches between Linux page table *mmap-mremap* primitives and existing user-space allocator implementations. It can override allocators in legacy C codes using the Linux LD_PRELOAD primitive [24].

We present the overview of Brug system design in Figure 5 and memory operations in Figure 6 When a request comes in, the handler sends it to the Brug decision maker. The decision maker will decide what action to take based on the metadata and request size and will pass the request along to the proper handling route.

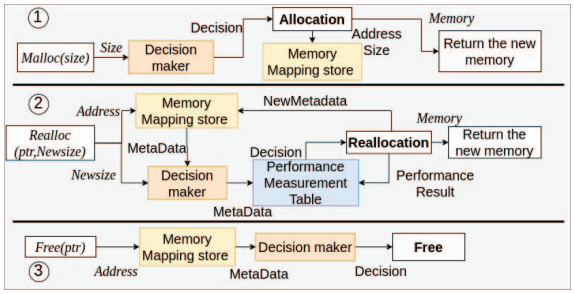

Fig. 6. Memory Operations.① Malloc: The malloc request with the size is passed to the decision maker. The decision maker decides the allocator to use. The allocator performs the allocation, stores the address and size in the memory mapping store, and returns the memory to the user. ② Realloc: The realloc request is passed into the mapping store to retrieve the metadata. Combined with the new size, the decision maker determines the re-allocation operation. (With the autotuner, we record the cost of each re-allocation in the measurement table, instructing further re-allocation decisions). ③ Free: The request is passed into the mapping store to choose the corresponding memory allocator to free it.

Memory Allocators: The memory allocators are more suitable for dealing with small and short-lived objects. Different allocators promise different benefits according to their characteristics. Ptmalloc2 in Glibc [20] is the default base allocator providing re-allocation performance, less memory waste, and high availability.

Memory Mapping: On the other hand, large and highly-active objects use *mmap-mremap* to take advantage of Linux OS memory management characteristics. When a memory request meets the policy (such as crossing a sweet spot), we proceed with a *mmap-copy-free* routine to switch the object from the memory allocator to the OS-managed mmap area.

In this way, we preserve all the mechanisms and provide the optimal solution for dealing with all spectrum of memory requests (small and fast allocation memory objects as well as long-time, large memory objects).

A. System Design

1) Rust Library for Modern Code Bases: Brug takes advantage of several Rust characteristics and incorporates a powerful decision-maker equipped with an auto-tuner. As Rust has emerged as a viable alternative for system software and modern middleware, we view it as pivotal in advancing the efficiency of memory-intensive applications. We leverage the Rust runtime interface to implement memory-related functions (`malloc`, `realloc`, `calloc`, and `free`).

Brug comprises three main components: the Decision Maker, Memory Mapping Store, and Performance Measurement Table. When a request is received, the handler forwards it to the Brug decision maker. After determining the action based on metadata from the Memory Mapping Store and Performance Measurement Table, it directs the request to the appropriate handling route. The design of Brug is outlined as follows:

Decision Maker: Decision Maker includes both the manual mode and the automatic mode. In manual mode, the user can supply their own size-allocator mapping to allow different

sizes of the objects to use the desired memory allocator. Alternatively, the user can choose the convenient auto-tuner mode, which can decide the desired memory allocator based on collected data.

Memory Mapping Store: The Memory Mapping Store is implemented using a `BtreeMap` to record the address of the memory object with the allocators it used. When further memory operations touch this object (realloc or free), its allocator information is retrieved. This ensures the proper memory allocator handles this object. During program runtime, an object can also switch to a different allocator if necessary. This switch happens during a re-allocation, following a `malloc-copy-free` sequence (the specified allocator allocates a new memory block, and copies old content there).

Performance Measurement Table: Each re-allocation time will be recorded into the Performance Measurement Table (a 2-D array representing different allocators and size classes (multiples of 4 KiB)). The recording process calculates the arithmetic mean between old and new record values.

Brug Modes: Six different modes are available in Brug and can be switched via a flag (`BRUG_ALLOC`). Three modes utilize existing Rust memory allocators: Ptmalloc2 (SYS), Jemalloc, and Mimalloc. The fourth mode is a Rust implementation using *Mmap-Mremap* primitives. The final two modes use the new features of the decision maker: size template (`BrugTemplate`) and auto-tuner (`BrugAutoOpt`).

BrugTemplate: The size template uses a simple map to record the re-allocation size with the allocator to use. It is configurable to the user using the mapping entry before the program execution. Users can define which allocator they want to use in specific memory size groups (e.g., Ptmalloc2 for objects between 0-4 KiB, MiMalloc for objects between 4 KiB-10 KiB, and Jemalloc for 10 KiB-10 MiB). With previous knowledge from our large-scale experimentation, we have preset the `BrugTemplate` as follows: Jemalloc (0 to 4 KiB), Ptmalloc2 (4 KiB to 64 KiB), *mremap* (64 KiB and larger).

BrugAutoOpt: `BrugAutoOpt` includes an auto-tuner to automatically choose the optimal memory allocator. With the data recorded in the Performance Measurement Table, the following re-allocation operation will choose the best-performing allocator strategy within a specific size class. However, it is important to note that this training process incurs additional overhead before all the data is learned, and the auto-tuner reaches its full potential.

Object Monitor: Expanding the functionality with the Memory Mapping Store and Performance Measurement Table, we build this monitoring function. In this mode, we record additional data, including the number of re-allocations, the times an object gets remapped, and the total size and re-allocation duration during the life of a memory object. This information could help programmers understand deeper details about the lifespan of an object and provide information for further optimization.

With Brug design, what we achieve is flexibility and ease of use tools for the programmer to interact with memory allocators. In a typical program, programmers are limited to only one specific allocator across the whole program. Brug brings the capability to set up specific allocators at the program level, data structure level, and function level.

Brug is packaged into a crate that can be accessed through the Rust package system. Loading it into new projects is also convenient using the `#[global_allocator]` attribute. Mode changes are easily accomplished via this macro, as shown in Listing 1.

```
set_allocator_mode!(mode,{func});
```

Listing 1. Macro switch mode

2) C Library for Legacy Applications: In the C implementation, Brug includes a simpler decision-maker designed to mitigate API differences and implement the switch policy. It only takes the re-allocation size as the deciding factor and checks the object when it gets re-allocated. For requests with a size larger than a certain value (switch point, 128 KiB by default), it applies the *mmap-mremap* scheme. Otherwise, it uses the basic base allocators (Ptmalloc2/Jemalloc/TCmalloc).

In addition to the size switch point policy, we have introduced the concept of identifying frequently re-allocated objects. When we categorize an object as frequently resized, it will be switched from the heap or memory pool to the *mmap* area. This movement ensures Brug eliminates copies for future re-allocations while providing fast allocation speed and avoiding internal fragmentation for small objects.

We use an integer called allocation counter for this functionality. It works as follows: each expansion grows the counter by one, and each contraction will decrease until zero. We expect objects with a high counter in two scenarios. The first is a small memory object with high activeness (i.e., gets re-allocated often). In this case, this object could significantly benefit from overcommitting the virtual memory. After overcommitting, this object gets a fast response and has less chance of fragmentation. The second scenario is large data structures or data chunks with a long lifespan. They may have significant copy overhead which can be eliminated by the *mremap*.

To record additional metadata of Brug, we append a header before the object. Considering the space needed by the header, we will only store the size variable for objects less than a page size (4096 KiB).

IV. Brug Performance Evaluation

We evaluate Brug using benchmarks and real-world applications. For the Rust version, we conducted experiments with typical data structures as well as industry-grade frameworks and workloads, such as Apache Arrow [25] and Wasmtime WebAsembly engine [26]. For the C version, we assessed Brug with Redis [15] running the YCSB [28], [29] benchmark. Our experiments ran on multiple platforms listed in Table II.

A. Brug Microbenchmarks

We start by discussing three Brug Rust modes and compare them with the Rust #[global_allocator] attribute. The first set of experiments showcases all six scenarios with a Rust standard

TABLE II
HARDWARE PLATFORMS USED FOR BRUG EXPERIMENTATION.

Evaluation platform	CPU	RAM	L1 Cache	L2 Cache	L3 Cache	Software Version	Implementation
Commodity Local machine	AMD Ryzen 9 5900X	64 GiB	768 KiB	6 MiB	64 MiB	Ubuntu 22.04, Linux-6.2.6, Rust-1.67.1	Rust
AWS C6a.4xlarge	AMD EPYC 7R13	32 GiB	256 KiB	4 MiB	32 MiB	Ubuntu 20.04, Linux-5.15, GCC-9.4.0	C

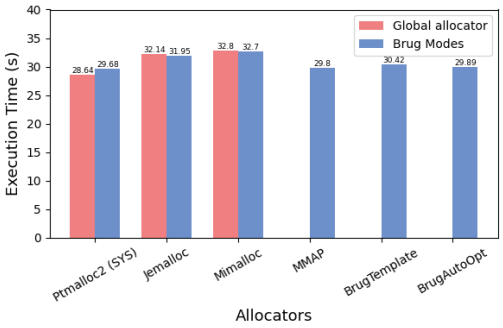

Fig. 7. Total execution time from inserting 100 million integers into a Rust vector, all allocators performance. The results are the sum of 15 repetitions. (Lower is better.)

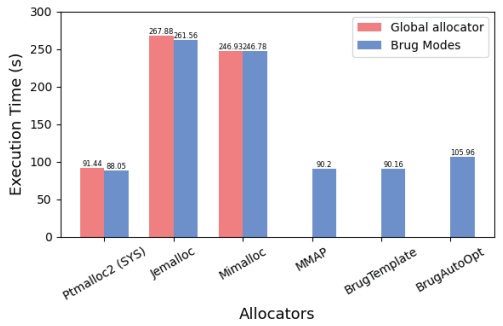

Fig. 8. Total execution time from writing 15 GiB data into a Rust vector with page granularity (4 KiB), all allocators performance. Data is the sum of 15 repetitions. (Lower is better.)

vector, set up as follows: we insert a hundred million integers into a standard library Rust vector.

In the experiments shown in Figure 7, we observe a 3.5% overhead between the default setting using the Brug Ptmalloc2 (SYS) mode. However, when using Jemalloc and Mimalloc, which employ the same attributes, the results are very close. This difference suggests that there is a certain overhead when using the #[global_allocator] attribute.

Further interpreting the results presented in Figure 7, we observe the auto-tuner (BrugAutoOpt) performing closely in terms of time. The size template (BrugTemplate) mode lags behind, with a 6% slower execution time compared to the baseline. Meanwhile, all these modes exhibit slightly faster performance compared to the two user-space allocators, Jemalloc and Mimalloc. Overall, the performance of Rust standard vector integer insertions across all nine groups is fairly close.

In a subsequent step, we increased the data size in each insert operation, conducting an experiment where we wrote 15 GiB of data into a Rust standard vector, with each write consisting of a single page size (4 KiB). The significant overhead incurred by the two user-space allocators, Jemalloc and Mimalloc, is evident, reporting a 1.95X increase in execution time compared to the baseline. Among all nine groups, Brug SYS mode gives the best performance, surpassing the baseline performance. Brug MMAP mode and BrugTemplate also show similar results to the baseline. Meanwhile, BrugAutoOpt does not perform as well; we believe that result is due to the additional cost during the training process.

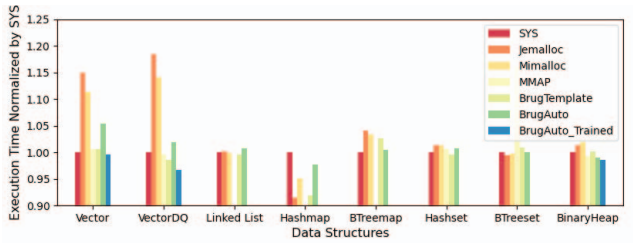

Fig. 9. Results of inserting data into different Rust std::collection data structures, showcasing the performance of all allocators. Experiments trigger both the re-allocation and allocation phases. BrugAuto_Trained is not applied to Linked List, Hashmap, BTreemap, Hashset, and BTreeset, as they primarily involve small allocations. Importantly, BrugAuto does not negatively impact allocation performance or re-allocation performance (The Y-axis indicates the execution time normalized by SYS. The results are the mean value of 15 repetitions; lower is better).

This experiment demonstrates that the overhead observed in write-based workloads with user-space allocators also applies to Rust. Moreover, we have shown that with the proper setup, Brug can maintain close performance to the baseline considering the additional cost imposed by the #[global_allocator] attribute. These results show that the default allocator still has room for better performance.

We evaluated various Rust standard collection data structures, and the results are presented in Figure 9. In the VectorDequeue experiment, similar results were observed to Vector: user-space allocators (Jemalloc and Mimalloc) incurred overhead compared to the system allocator. Various Brug modes demonstrated comparable performance to the baseline. Notably, the trained Brug auto tuner mode outperforms all other modes in Vector and VectorDequeue.

In the Maps experiment, Hashmap showed improved performance with user-space allocators (Jemalloc and Mimalloc) outperforming the baseline. Jemalloc demonstrated the most significant gain of 8%, and Mimalloc around 4%, due to the design of Hashmap based on the allocation of small objects. BrugAuto still managed to show better performance than the baseline, gaining a 1-2% advantage. However, in BTreemap, which is backed by an array-based data structure, the baseline performed the best again, and Jemalloc exhibited the worst performance with a 4% decrease compared to the baseline.

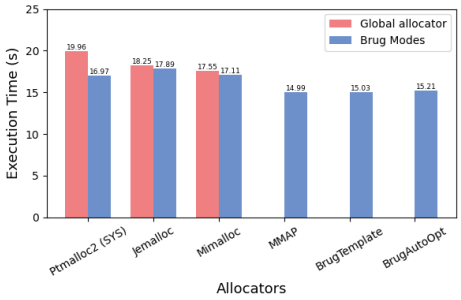

Fig. 10. Total execution time from inserting 100 million integers into an Arrow mutable buffer, all allocators performance. Data is the sum of 15 repetitions. (Lower is better.)

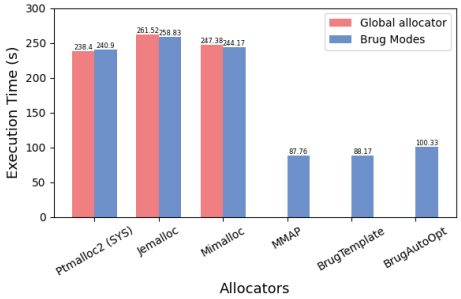

Fig. 11. Total execution time from writing 15 GiB data into an Arrow mutable buffer with page granularity, all allocators performance. Data is the sum of 15 repetitions. (Lower is better.)

BrugAuto managed to handle this situation as well, showing a $< 1\%$ performance difference and gaining the second position.

In the Sets experiment, most allocators yielded similar results, except for the *mmap-mremap* mode, which incurred a performance loss of nearly 20% in BTreeset. In the rest of the data structures (LinkedList and BinaryHeap), different allocators exhibited a narrow margin of 1-2% performance difference. Ptmalloc2 managed to slightly outperform the other allocators in HashSet and BinaryHeap, while user-space allocators won in Btreeset. In all three cases, BrugAuto managed to keep the performance losses lower than the worst case and even become the best performing in BinaryHeap, showing promising results.

B. Brug Integration with Industry-grade Frameworks

Brug seamlessly integrates with various applications, offering a straightforward integration process. This integration extends to applications like Apache Arrow, enabling developers to leverage the performance benefits provided by Brug. Furthermore, we demonstrate its applicability in real-world scenarios through experiments conducted with Wasmtime, a web assembly engine. This diverse integration showcases the versatility of Brug and its ability to deliver performance gains across different applications.

Apache Arrow: In our experiments with the Arrow Mutable buffer, presented in Figure 10, we write into this buffer at either integer (4 bytes) or page-size (4 KiB) granularities.

Interestingly, we observed a turnaround when comparing the results with the standard vector experiments. The baseline

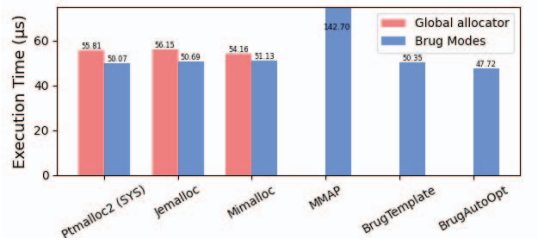

Fig. 12. Wasmtime linking workload, all allocators performance. Data is the average of 5 repetitions. (Lower is better.)

Ptmalloc2 allocator performed the worst in all groups, trailing behind the best-performing Brug *mmap-mremap* mode by 33% in execution time. Simultaneously, BrugTemplate and BrugAutoOpt showed very close results to the optimal. Additionally, we found that the SYS mode in Brug outperformed the baseline Ptmalloc2 by 17%. These findings suggest that the baseline allocator may encounter issues when handling the Arrow mutable buffer.

Expanding the experiment from integer inserts to writing large file data, as depicted in Figure 11, notable differences emerged in the results when utilizing the *MMAP* allocator compared to the other three allocators. Both *MMAP* and BrugTemplate demonstrated similar performance, while BrugAutoOpt exhibited a slight performance gap due to its training overhead.

Wasmtime WebAssembly Engine: WebAssembly [27] is an increasingly important technology that offers practitioners a hardware-independent program representation that is used in applications ranging from browser workloads to blockchain technology and serverless applications [35]. Wasmtime [26] is the industry de-facto standard for running WebAssembly code under a runtime engine.

In Figure 12, we explore the performance of different allocators when handling the Wasmtime linking example in Rust [36]. Linking is very important in this case as it lies in the cold start path of WebAssembly-enabled serverless workloads [35]. Saving time here means running client code more efficiently.

The Linker data structure in this example serves as a comprehensive case, encompassing both Hashmap, Vector, and PhantomData (representative of various data structures). The example illustrates how to set a linker in the Wasmtime runtime. The results reveal that Brug outperforms any individual allocator, highlighting the potential benefits of judiciously employing different allocators in tandem and how they fit with different data structures.

Overall, Brug demonstrates competitive performance across

TABLE III
THE BRUG AUTO-TUNER PERFORMANCE COMPARED WITH BEST- AND WORST-CASE PERFORMANCE SCENARIOS.

	ArrowBuffer		**Wasmtime**
	Page granularity	Integer granularity	Linker
Best solution	14.99 (s)	87.76 (s)	47.72 (us)
Auto tuner	15.21 (s) (1.45%)	100.33 (s) (14.32%)	47.72 (us)
Worst case	19.96 (s) (31.69%)	261.52 (s) (183.67%)	142.70 (us)

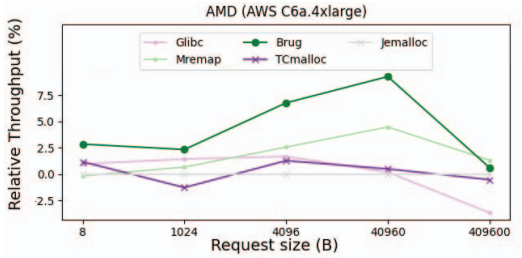

Fig. 13. Redis YCSB time series workload on AWS AMD C6a.4xlarge instance listed in Table II. Throughput normalized to Jemalloc. (Higher is better.)

various data structures and applications (Table III). Additionally, it provides programmers with a powerful auto-tuner mechanism, enabling them to achieve optimal performance without the overhead of extensive experimentation in identifying the right allocator for a specific workload.

C. Brug with Legacy Applications

Whereas modern Rust code-bases are growing and the Rust language is constantly increasing in popularity, legacy applications still exist and make up a large portion of modern workloads. We therefore believe it is important to optimize their memory (re-)allocation behavior. We thus deploy the (more limited) Brug C-library version to memory-intensive legacy applications.

Using the LD_PRELOAD mechanism we integrated the Brug C-version library with the widely used, industry-standard Redis key-value store. The Yahoo! Cloud Serving Benchmark (YCSB) [28] is an open-source and widely-used realistic benchmark for key-value stores. In this context, we employ the write-based time series workload, comprising 90% inserts and 10% reads, to simulate a write-heavy workload.

We utilized an Amazon AWS AMD-based platform, as outlined in Table II. The results are presented in Figure 13.

Brug shows a clear advantage over the rest, achieving 2.5% better performance in the small size groups and more than 5% better performance in the 4 KiB and 40 KiB groups. The *mmap-mremap* approach provides 1-2% better performance than baseline Jemalloc. The rest three allocators share a similar performance. We conclude that Brug provides higher throughput compared to the baseline Jemalloc in write-heavy workloads in key-value store workloads.

V. Discussion

Our investigation underscores the impact of (re-)allocation performance on various allocator designs and their consequential influence on application performance. This perspective offers a fresh lens for understanding memory allocators, proving critical for enhancing the memory performance of systems and middleware dealing with large and dynamic working sets. Below, we delve into key insights derived from our work.

Adaptation: The intricacies of optimizing performance from both memory allocator and data structure perspectives pose a challenge for most programmers. Despite not always providing the optimal solution, the Brug auto-tuner consistently yields favorable results in various scenarios without necessitating additional modifications. Across all experiment results in Table III, Brug incurs at most a 20% overhead due to the training process, while managing to achieve performance gains ranging from 12% to 170% compared to worst-case scenarios. We stress that this is very important going forward because, as our experiments attest, there is no one-size-fits all allocator. Therefore, an adaptive mechanism that can leverage strengths of all kinds of allocation techniques, while discarding their weaknesses is key for achieving acceptable performance.

Object Lifespan Opaqueness: The effect of long-running resizable objects, particularly those accessed by multiple threads, represents a seldom-explored area in current research. To address this gap, we integrated a monitoring mechanism into our design, enabling programmers to gain clearer insights into the runtime behavior of such objects.

Limitations: While our allocator introduces additional storage and runtime overhead for small objects, its primary advantages become apparent in write-heavy workloads. Future optimizations could aim to mitigate overheads related to small objects.

VI. Related Work

Memory allocation research encompasses two key domains: OS memory management and user-space allocators.

Linux Memory Management: Noteworthy contributions in Linux memory management have been made by Huang et al., who conducted an extensive study on its evolution, highlighting both achievements and persisting challenges [37]. Additional works, such as DAOS [38], have aimed at advancing our understanding of data and memory. Proposals for new page table designs, including hashed page tables [39], clustered page tables [40], and the integration of page walk caches [41], showcase ongoing innovation in this space.

Allocators: The optimization trajectory for TCmalloc includes significant strides, particularly with Huge Page support evident in TCMalloc's page heap implementation [42] and strategies for releasing partial huge pages to the operating system [43]. Recent research has delved into the development of smarter allocators tailored for embedded systems [44] and the dynamic landscape of mobile edge computing [45].

Diverging from the outlined research, our work introduces a fresh perspective on memory allocators. Our focus lies on modern workloads characterized by dynamic changes in object sizes during runtime. The distinctive aspect of our approach lies in making judicious decisions between employing user-space allocators and OS-based schemes. By leveraging the strengths of both paradigms, Brug aims to optimize performance in the context of contemporary workloads.

VII. Conclusion

We have presented, designed, and implemented Brug, an adaptive memory allocator. Brug provides seamless integration and competitive performance across different industry-grade applications. In addition, it offers practitioners an auto-tuner mechanism that allows for good performance without the need for large-scale experimentation or much application-related knowledge. For legacy code-bases, a drop-in library version is available to help practitioners with re-allocation performance.

REFERENCES

[1] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch, "In-memory performance for big data," 2014.

[2] D. Durner, V. Leis, and T. Neumann, "On the impact of memory allocation on high-performance query processing," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019, pp. 1–3.

[3] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.

[4] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon, "Overdriver: Handling memory overload in an oversubscribed cloud," *ACM SIGPLAN Notices*, vol. 46, no. 7, pp. 205–216, 2011.

[5] N. K. Sehgal and P. C. P. Bhatt, *Cloud Workload Characterization*. Cham: Springer International Publishing, 2018, pp. 61–83. [Online]. Available: https://doi.org/10.1007/978-3-319-77839-6_5

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, no. 4, p. 37–48, mar 2012. [Online]. Available: https://doi.org/10.1145/2248487.2150982

[7] B. C. Kuszmaul, "Supermalloc: A super fast multithreaded malloc for 64-bit machines," in *Proceedings of the 2015 International Symposium on Memory Management*, ser. ISMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 41–55. [Online]. Available: https://doi.org/10.1145/2754169.2754178

[8] D. Dice and A. Garthwaite, "Mostly lock-free malloc," *SIGPLAN Not.*, vol. 38, no. 2 supplement, p. 163–174, jun 2002. [Online]. Available: https://doi.org/10.1145/773039.512451

[9] Y. Afek, D. Dice, and A. Morrison, "Cache index-aware memory allocation," *SIGPLAN Not.*, vol. 46, no. 11, p. 55–64, jun 2011. [Online]. Available: https://doi.org/10.1145/2076022.1993486

[10] S. Schildermans, K. Aerts, J. Shan, and X. Ding, "Ptlbmalloc2: Reducing tlb shootdowns with high memory efficiency," in *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 2020, pp. 76–83.

[11] O. Mutlu, "Memory scaling: A systems architecture perspective," in *2013 5th IEEE International Memory Workshop*, 2013, pp. 21–25.

[12] R. Bryant, "Data-intensive supercomputing: The case for disc," 05 2007.

[13] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch, "In-memory performance for big data," *Proceedings of the VLDB Endowment*, vol. 8, pp. 37–48, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2735465

[14] A. S. Foundation, "Apeche spark," https://spark.apache.org/, 2022, online. Accessed: 2022-10-7.

[15] R. Ltd., "Redis," https://redis.io/, 2022, online. Accessed: 2022-5-14.

[16] J. Chakraborty, I. Jimenez, S. A. Rodriguez, A. Uta, J. LeFevre, and C. Maltzahn, "Skyhook: Towards an arrow-native storage system," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 81–88.

[17] D. Elias, R. Matias, M. Fernandes, and L. Borges, "Experimental and theoretical analyses of memory allocation algorithms," in *Proceedings of the 29th annual acm symposium on applied computing*, 2014, pp. 1545–1546.

[18] D. Leijen, B. Zorn, and L. d. Moura, "Mimalloc: Free list sharding in action," in *Asian Symposium on Programming Languages and Systems*. Springer, 2019, pp. 244–265.

[19] S. Ghemawat and P. Menage, "Tcmalloc: Thread-caching malloc," 2009.

[20] S. Poyarekar, "The gnu c library version 2.31 is now available," https://sourceware.org/legacy-ml/libc-announce/2020/msg00001.html, 2020, online. Accessed: 2022-5-14.

[21] D. McCracken, "Object-based reverse mapping," in *Linux Symposium*, 2004, p. 357.

[22] github, "The top programming languages," https://octoverse.github.com/2022/top-programming-languages, 2023, online. Accessed: 2023-11-24.

[23] M. M. Yuliia Panasenko, "Rust market overview: reasons to adopt rust, rust use cases, and hiring opportunities," https://yalantis.com/blog/rust-market-overview/, 2023, online. Accessed: 2023-11-24.

[24] M. Kerrisk, "ld.so(8) — linux manual page," https://man7.org/linux/man-pages/man8/ld.so.8.html, 2021, online. 2021-08-27.

[25] T. A. S. Foundation, "Apache arrow," https://arrow.apache.org/, 2022, online. Accessed: 2022-5-14.

[26] B. Alliance, "Wasmtime," https://wasmtime.dev, 2023, online. Accessed: 2023-11-30.

[27] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[29] W. Wang and S. Diestelhorst, "Quantify the performance overheads of pmdk," in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 50–52.

[30] cppreference.com, "realloc," https://en.cppreference.com/w/c/memory/realloc/, 2022, online. Accessed: 2022-7-8.

[31] MallocInternals, "Glibc wiki," https://sourceware.org/glibc/wiki/MallocInternals, 2022, online. Accessed: 2023-11-30.

[32] N. Amit, "Optimizing the TLB shootdown algorithm with page access tracking," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 27–39. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

[33] J. Evans, "mremap with modern linux kernel," https://jemalloc.net/mailman/jemalloc-discuss/2014-April/000757.html, 2015, online. Accessed: 2023-11-30.

[34] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*, H. G. Baler, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116.

[35] M. Arutyunyan, A. Berestovskyy, A. Bratschi-Kaye, U. Degenbaev, M. Drijvers, I. El-Ashi, S. Kaestle, R. Kashitsyn, M. Kot, Y.-A. Pignolet *et al.*, "Decentralized and stateful serverless computing on the internet computer blockchain," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 329–343.

[36] alexcrichton, "wasmtime/examples/linking.rs," https://github.com/bytecodealliance/wasmtime/blob/main/examples/linking.rs, 2022, online. Accessed: 2023-12-5.

[37] J. Huang, M. K. Qureshi, and K. Schwan, "An evolutionary study of linux memory management for fun and profit," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 465–478.

[38] S. Park, M. Bhowmik, and A. Uta, "Daos: Data access-aware operating system," in *HPDC 2022*, 2022. [Online]. Available: https://www.amazon.science/publications/daos-data-access-aware-operating-system

[39] J. Huck and J. Hays, "Architectural support for translation table management in large address space machines," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, p. 39–50, may 1993. [Online]. Available: https://doi.org/10.1145/173682.165128

[40] M. Talluri, M. D. Hill, and Y. A. Khalidi, "A new page table for 64-bit address spaces," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 184–200. [Online]. Available: https://doi.org/10.1145/224056.224071

[41] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 48–59, jun 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815970

[42] A. H. Hunter, C. Kennelly, D. Gove, P. Ranganathan, P. J. Turner, and T. J. Moseley, "Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[43] M. Maas, C. Kennelly, K. Nguyen, D. Gove, K. S. McKinley, and P. J. Turner, "Adaptive hugepage subrelease for non-moving memory allocators in warehouse-scale computers," in *International Symposium on Memory Management (ISMM) 2021*, 2021.

[44] M. Ramakrishna, J. Kim, W. Lee, and Y. Chung, "Smart dynamic memory allocator for embedded systems," in *2008 23rd International Symposium on Computer and Information Sciences*, 2008, pp. 1–6.

[45] Z. Ali, S. Khaf, Z. H. Abbas, G. Abbas, F. Muhammad, and S. Kim, "A deep learning approach for mobility-aware and energy-efficient resource allocation in mec," *IEEE Access*, vol. 8, pp. 179 530–179 546, 2020.