Memory at Your Service: Fast Memory Allocation for Latency-critical Services

Aidi Pi, Junxian Zhao, Shaoqi Wang, Xiaobo Zhou
University of Colorado Colorado Springs
{epi,jzhao,swang,xzhou}@uccs.edu

ABSTRACT

Co-location and memory sharing between latency-critical services, such as key-value store and web search, and best-effort batch jobs is an appealing approach to improving memory utilization in multi-tenant datacenter systems. However, we find that the very diverse goals of job co-location and the GNU/Linux system stack can lead to severe performance degradation of latency-critical services under memory pressure in a multi-tenant system.

We address memory pressure for latency-critical services via fast memory allocation and proactive reclamation. We find that memory allocation latency dominates the overall query latency, especially under memory pressure. We analyze the default memory management mechanism provided by GNU/Linux system stack and identify the reasons why it is inefficient for latency-critical services in a multi-tenant system. We present Hermes, a fast memory allocation mechanism in user space that adaptively reserves memory for latency-critical services. It advises Linux OS to proactively reclaim memory of batch jobs. We implement Hermes in GNU C Library. Experimental result shows that Hermes reduces the average and the 99^{th} percentile memory allocation latency by up to 54.4% and 62.4% for a micro benchmark, respectively. For two real-world latency-critical services, Hermes reduces both the average and the 99^{th} percentile tail query latency by up to 40.3%. Compared to the default Glibc, jemalloc and TCMalloc, Hermes reduces Service Level Objective violation by up to 84.3% under memory pressure.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Software libraries and repositories**; **Memory management**.

KEYWORDS

Job co-location, Memory management, Latency-critical services

ACM Reference Format:
Aidi Pi, Junxian Zhao, Shaoqi Wang, Xiaobo Zhou. 2021. Memory at Your Service: Fast Memory Allocation for Latency-critical Services. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464298.3493394

Middleware '21, December 6–10, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3493394

1 INTRODUCTION

Latency-critical services such as key-value store and web search are usually featured with largely varied peak and average resource consumption [26, 40]. For guaranteed performance of latency-critical services, a naive approach is to use a dedicated system for latency-critical services. However, the approach leads to a large amount of idle resources during runtime since the average resource consumption of the services is usually much less than their peak consumption [22, 33, 42]. For instance, SnowFlake system found that the average memory utilization on its servers is only ~ 19% [42]. To improve the utilization of resources, it is a common practice that best-effort batch jobs are co-located with latency-critical services to exploit transient resources in datacenters [22, 29, 30, 43, 44, 47].

Although co-location with memory sharing increases resource utilization, it often significantly degrades the latency particularly the tail latency of latency-critical services. Latency-critical services like cloud-native key-value store and web search commonly distribute requests across many servers, thus the end-to-end response time is determined by the slowest individual latency [9, 20, 22, 48]. We find the root cause of long tail latency is due to the very diverse goals of job co-location and the GNU/Linux system stack. On one hand, job co-location leverages idle resources for batch jobs while maintaining the performance of latency-critical services. On the other hand, the GNU/Linux stack tries to accommodate as many submitted processes as possible while only offering few knobs to prioritize processes. As a result, although co-located latency-critical services and batch jobs may both survive, the performance of latency-critical services is significantly degraded under memory pressure, which jeopardizes Service Level Objective (SLO).

There are mainly two categories of research on improving performance for latency-critical services. Studies [9, 13, 34, 48] improve performance for latency-critical services by leveraging their runtime characteristics. For example, ROLP [13] is a runtime object lifetime profiler for efficient memory allocation and garbage collection for latency-critical services. However, these studies do not take job co-location into consideration. Studies of the other category [29, 30, 43, 47] target co-location of latency-critical services with other jobs. For example, PerfIso [29] and Dirigent [47] are two representative approaches that leverage multicore systems to efficiently share CPU resource between processes. Our work falls into the second category.

While existing efforts try to push the resource utilization to the limit, memory management for latency-critical services still faces significant challenges. First, the runtime behavior of a job is difficult to predict. In particular, it is difficult to obtain the amount of memory that will be requested by a job in the future. Second, it is expensive to reclaim physical memory that is occupied by a process. If a process requests more memory when the node memory

is almost used up, swapping will be triggered to make space for the requested memory. However, swapping is an expensive operation that takes a long period of time (tens of milliseconds to seconds) or even leads to thrashing. In such cases, the performance of latency-critical services are significantly degraded.

Since the original purpose of a dedicated system is for sole use by latency-critical services, ideally their performance should not be affected by batch jobs. In a shared environment, memory is frequently allocated and reclaimed due to provisioning of various workloads. However, the memory reclaim mechanism in Linux OS significantly degrades the performance of latency-critical services under memory pressure, which makes co-location inefficient or even ineffective. In light of the challenges, we tackle the problem from a new perspective: resource slacks should be reserved for latency-critical services in case of a burst of resource requests. PerfIso [29] is a preemptive approach that adopts this principle to achieve CPU sharing between latency-critical services and batch jobs. However, memory sharing is very different from CPU sharing since tasks on a core can be easily preempted and later rescheduled [18]. Data in memory can only be preempted by swapping them onto disks, which is a very expensive operation.

We aim to materialize the principle to achieve fast memory allocation for latency-critical services in a multi-tenant system. Our experiments find that memory allocation latency takes up to 97.5% of a whole query latency. Thus, we focus on reducing the memory allocation latency for latency-critical services. The design should meet the following requirements:

- **R1** Latency-critical services have the highest priority. This is the primary principle. Best-effort batch jobs can share idle resources only if they do not affect the performance of latency-critical services.
- **R2** Memory should be allocated in a fast manner. This is the key to achieving low latency for latency-critical services when they request memory.
- **R3** The design should be generally applicable to all applications written in a popular language such as C / C++. That is, the source code of applications should not be modified.
- **R4** The overhead should be low. In other words, it should consume little resource of a node.

In this paper, we make the following contributions. First, we analyze the current memory management in GNU C Library (a.k.a. Glibc) and Linux OS, and show that it is inefficient for memory sharing between latency-critical services and batch jobs. In particular, 1) it adopts an on-demand physical memory allocation mechanism in order to accommodate as many processes as possible without prioritization. Though this mechanism works well in a dedicated system with sufficient memory, it significantly degrades job performance or even causes thrashing under memory pressure. 2) It uses a reactive algorithm to reclaim file cache even if no process accesses the cache. The design expects the cache will be accessed again in the near future. The reactive algorithm introduces significant delay on latency-critical services since a memory reclaim routine is invoked before requests are served. In summary, the design of the current GNU / Linux stack contradicts the goal of co-location and memory sharing of latency-critical services and batch jobs.

Second, we present Hermes, a library-level mechanism for fast memory allocation for latency-critical services in multi-tenant systems. Hermes maintains one dedicated memory pool for each latency-critical service (**R1**, **R2**). Upon receiving requests from a latency-critical service, memory can be immediately allocated from the memory pool to the service. Hermes uses a lightweight heuristic to determine the size of the memory pool (**R4**). It advises Linux OS to release file cache pages occupied by batch jobs under memory pressure so as to make more available memory for latency-critical services (**R1**). We implement Hermes in library Glibc. It is a library-level mechanism without modification to applications (**R3**) or Linux OS. Note that Hermes could be implemented into Linux OS, but the modification may affect other processes, incur security issues, and importantly violate Linux monolithic kernel generality.

We conduct experiments for Hermes with a micro benchmark and two real-world services under a multi-tenant system. Compared to the default Glibc, Hermes reduces the average and the 99^{th} percentile memory allocation latency by up to 54.4% and 62.4% under memory pressure, respectively. The allocation latency is as low as $4\mu s$ for small requests and $1ms$ for large requests.

Furthermore, we use Redis [4] and Rocksdb [5] as two real-world services to examine the query latency. Results show that Hermes reduces both the average and the 99^{th} percentile tail query latency by up to 40.3%. Compared to the default Glibc, jemalloc and TCMalloc, Hermes reduces the SLO violation by up to 84.3% under memory pressure. Hermes achieves significantly improved system throughput. Results also show that Hermes achieves similar or slightly better query latency under a dedicated system. The overhead of Hermes is negligible.

The rest of the paper is organized as follows. Section 2 introduces the default GNU stack and its problems. Sections 3 and 4 present the design and implementation of Hermes, respectively. Section 6 discusses Hermes. We present the related work in Section 7 and conclude the paper in Section 8.

2 BACKGROUND AND MOTIVATIONS

2.1 Memory Management in Glibc

The famous `malloc` function call in Glibc is a unified interface for programs to allocate memory from Linux OS. A process conveniently obtains the address of the memory space without knowing the underlying mechanism by calling `malloc`. The function call uses two Linux system calls `brk` and `mmap` to serve memory requests of different sizes. Figure 1(a) shows the simplified address space of a process that includes memory chunks allocated by both system calls. We focus on the mechanisms in Glibc that manipulate the main heap space and mmapped memory chunks. Both kinds of memory are dynamically allocated at runtime.

System call `brk`. Each process has exactly one main heap that is a continuous virtual address space. Glibc divides the main heap into two areas: the allocated area and the top chunk. Glibc keeps track of the used and free space in the allocated area. It is worth noting that the allocated area and the top chunk in Glibc are transparent to Linux OS. Following the allocated area lies the top chunk that is a continuous free address space. The end address of the top chunk is the program break returned by the `sbrk` wrapper function which calls the `brk` system call. Upon a request for a small size of memory

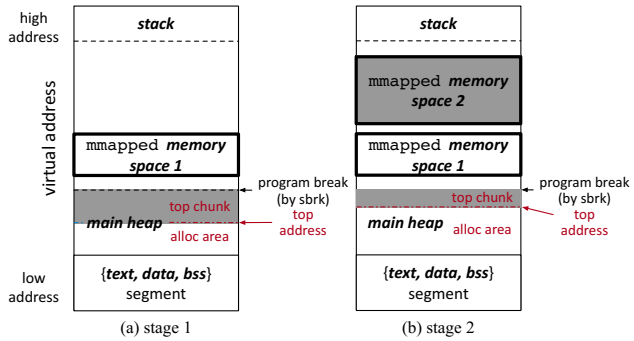


Figure 1: Process address space in Linux. Shaded areas represent allocated virtual memory whose physical pages do not reside in RAM. Red fonts represent variables on Glibc.

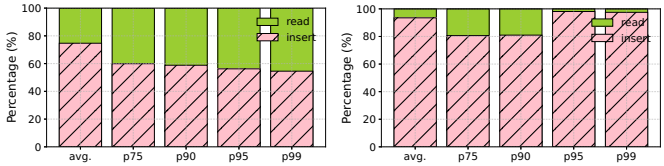
(< 128 KB by default), Glibc first tries to find a free space in the allocated area. If it cannot satisfy the request, space is taken from the beginning of the top chunk and added to the allocated area. Once the top chunk is used up, Glibc expands the main heap by calling `sbrk` with the exact requested size. If the top chunk is greater than a certain threshold, Glibc shrinks the main heap by passing a negative number to `sbrk`.

System call `mmap`. Besides the main heap, a process can have multiple disjoint memory chunks allocated by `mmap`. This system call can either map a file to process address space or allocate anonymous pages. Glibc leverages the anonymous page usage to handle large memory requests (≥ 128 KB by default). Upon success, it returns the starting address of the newly allocated mmapped memory chunk. Glibc gives the memory chunk to the process after a bookkeeping operation. When a process frees a memory space allocated by `mmap`, Glibc releases it directly back to Linux OS.

Upon return of both system calls, a process gets a virtual memory space while the corresponding physical memory does not necessarily reside in RAM at the moment. Linux OS constructs the virtual-physical address mapping only when the process accesses (i.e., writes or reads or executes) the allocated memory for the first time. For example, in Figure 1(a), the process has a main heap and a mmapped memory space 1. In Figure 1(b), the process allocates a new mmapped memory space and writes data in the main heap. The newly mmapped memory space does not have corresponding physical pages yet. Thus, the virtual-physical mapped space in the main heap expands.

Two benefits come with the on-demand mapping construction. For Linux OS, physical memory pages are loaded for the actually used memory since physical memory is a scarce resource. For the process, it accelerates the memory allocation routine. The reason is that the mapping construction for all the virtual addresses requires loading all the physical pages at once, which takes a longer time than only returning the virtual address.

While usually fast, the on-demand virtual-physical mapping construction can be significantly delayed when there is insufficient physical memory in the node, which is common in a multi-tenant system. At this point, Linux OS starts to reclaim physical pages by either directly freeing them or swapping them onto disks.



(a) Small (1KB) requests. (b) Large (200KB) requests.

Figure 2: The percentage breakdown of the insert and read operations in Rocksdb.

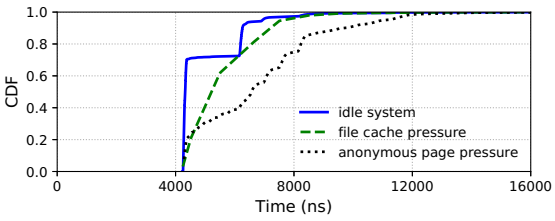


Figure 3: The CDF of the memory allocation latency.

2.2 Case Studies

In real-world latency sensitive services, latency spent in memory allocation during data insertion takes a large portion of latency of a whole workload. We take Rocksdb as a case study to illustrate that memory allocation latency is much higher compared to data read latency using both small (1KB) and large (200KB) requests. We use Glibc as the memory allocator and execute Rocksdb queries without any memory pressure. Each query is a data insertion operation (involving memory allocation) followed by a read operation. Figure 2 shows the percentage breakdown of the query latency at specified percentiles. For small requests, the average (99^{th} percentile) query latency is the $15\mu s$ ($29\mu s$). Data insertion latency is 74.7% (54.5%) of the average (99^{th} percentile) overall query latency. For large request, the average (99^{th} percentile) query latency is the $1730\mu s$ ($14069\mu s$). Data insertion latency is 93.5% (97.5%) of the average (99^{th} percentile) overall query latency. The impact of memory allocation is significant, and even more in large requests. As for data update requests, it renders similar results compared with read quests since they do not incur memory allocation.

We use another case study to demonstrate the memory allocation latency degradation under anonymous page pressure and file cache pressure. We use a micro benchmark that continuously sends 1KB-size memory requests until a total amount of 1 GB, using the default Glibc in a node with 128 GB RAM. We repeat the experiment under a dedicated system with sufficient memory, under anonymous page pressure, and under file cache pressure, respectively. The details of the micro benchmark and the node are described in Section 5.1. Figure 3 shows the CDF of the memory allocation latency under the dedicated system and two kinds of memory pressure.

Anonymous page pressure. To generate anonymous page pressure, we run a program that continuously sends memory allocation requests until the available memory in the node becomes about 300 MB. Note that, the available memory could not further drop below 300 MB due to the indirect and direct reclaim mechanisms of Linux

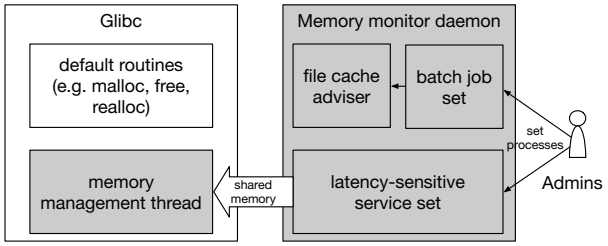


Figure 4: The architecture of Hermes.

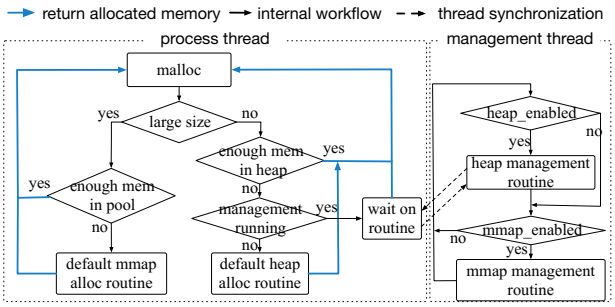


Figure 5: The workflow of the modified Glibc routines.

OS. At this point, new memory allocation requests from the micro benchmark trigger the memory reclaim routine and cause swapping. Figure 3 shows that the memory allocation latency significantly increases under anonymous page pressure. The average and the 99^{th} percentile allocation latency under anonymous page pressure are prolonged by 35.6% and 46.6% compared to those without memory pressure, respectively.

File cache pressure. We generate file cache pressure by loading 10 GB files and sending memory allocation requests to occupy the rest of the system memory until free memory drops to about 300 MB. In this case, memory reclaim routine starts but not necessarily trigger swapping since the file cache can be directly released without accessing the disk. Figure 3 shows that the memory allocation latency under file cache pressure is lower than that under anonymous page pressure, but it is still higher than that under a dedicated system. The average and the 99^{th} percentile allocation latency under file cache pressure are prolonged by 10.8% and 7.6% compared to those without memory pressure, respectively.

Memory pressure significantly prolongs memory allocation latency, which has non-trivial impact on SLO violation. We target both kinds of memory pressure and aim to reduce the memory allocation latency of latency-critical services in a co-located system as well as in a dedicated system.

2.3 Memory Reclaim in Linux OS

Linux OS emulates an LRU-like (Least Recent Used) algorithm for physical memory page reclaim by keeping four lists: `active_anon` and `inactive_anon` for anonymous pages, and `active_file` and `inactive_file` for file cache pages. The two active lists contain recently used pages while the two inactive lists contain pages that are not recently used. Under memory pressure, Linux OS scans through these four lists, updates page usage status, moves pages between lists, and selects pages to reclaim. Specifically, Linux OS keeps three memory watermarks (i.e. high, low and minimum) to instruct memory reclaim routine. When available memory drops below the low watermark, a page reclaim thread is started until available memory is larger than the high watermark. When available memory further drops below the minimum watermark, each memory request goes through a synchronous direct memory reclaim routine before the physical memory is allocated.

However, the page reclaim algorithm in Linux OS is inefficient for latency-critical services in a multi-tenant system. The watermarks are conservatively set at around 1‰ of a memory zone. For example, the total capacity of a memory zone in one of our physical nodes is 60 GB. The low and high watermarks are 53 MB and 64 MB, respectively. Since both latency-critical services and batch jobs tend to consume hundreds of megabytes or gigabytes of memory, the watermarks are too small to timely trigger the indirect memory reclaim thread. The direct memory reclaim routine even causes more delays on memory requests. After a process finishes, all of its anonymous pages are reclaimed immediately. However, the file cache pages loaded by the process are not reclaimed by Linux OS but remain in memory. They are only reclaimed upon memory pressure by the reclaim routine, which prolongs new memory requests. The memory pressure cannot be relived even if we increase the watermarks. Although, Linux OS triggers memory reclaim routine when there is still much free memory with higher watermarks, it does not distinguish latency-critical services and batch jobs. Memory from both kinds of workloads can be reclaimed. The performance of latency-critical services is still degraded.

[Summary] There are two drawbacks of the current GNU / Linux system stack that make the memory allocation of latency-critical services inefficient in a multi-tenant system. 1) Glibc only keeps a small chunk of physically mapped memory in the main heap, which is much less than the total size of memory requests from latency-critical services. 2) The on-demand virtual-physical memory mapping construction causes significant delay under memory pressure due to the conservative memory page reclaim mechanism in Linux OS.

3 HERMES DESIGN

3.1 Overview

In this paper, we propose and develop Hermes, a library-level mechanism to memory management that addresses the identified problems in the GNU/Linux system stack and reduces memory allocation latency of latency-critical services in a multi-tenant system. Hermes is transparent to applications and it does not make modification to Linux OS. As shown in Figure 4, Hermes consists of two major components: a memory management thread woken per f milliseconds in Glibc and a memory monitor daemon independently running on the same physical node. A system administrator sends the process IDs of batch jobs and latency-critical services to the memory monitor daemon. Upon memory pressure, the file cache adviser advises Linux OS to free the file cache owned by batch jobs. In Glibc, if a process is a latency-critical service, the memory management thread is started for memory reservation and virtual-physical address mapping.

3.2 Memory Management Thread

The goal of the memory management thread is to reserve memory and construct its virtual-physical address mapping in advance for latency-critical services. Figure 5 outlines the workflow of the management thread and the modified Glibc. The management thread periodically checks the current amount of reserved memory and decides whether to reserve more memory or release reserved memory back to Linux OS. When a process thread calls `malloc`, Hermes first tries to return the reserved memory to the process. If the reserved memory is insufficient, it uses the default routine to serve the request. Though sharing the same principle, the management thread uses different approaches to manage the main heap memory and mmapped memory chunks since they are allocated by two different system calls.

3.2.1 Heap Memory Management. Small-sized memory requests are allocated from the main heap, as shown in the `no` branch of the `large_size` statement in Figure 5. If there is sufficient memory in the main heap, Hermes immediately allocates it to the requests. Otherwise, if the management thread is running, the requests wait on it. If memory in the main heap is insufficient, the requests are allocated by the default allocation routine in Glibc. We show the heap management routine in Algorithm 1. In every round of the execution, the routine first updates the memory allocation metrics including the total size of all small memory requests (i.e. requests <128 KB) and the number of requests in the last interval. It then updates all the thresholds based on the collected memory allocation metrics (function UpdateThreshold). For example, the target amount of reserving memory is the total amount of memory requests in the last interval multiplying a reservation factor *RSV_FACTOR*. If the top chunk is smaller than the reservation threshold *RSV_THR*, it expands the current program break and immediately constructs the virtual-physical mapping for the newly allocated memory. Otherwise, if the free space in the top chunk exceeds the trim threshold *TRIM_THR*, it shrinks the top chunk by setting the program break to a lower memory address.

Algorithm 1 Heap management routine.

```
1:  RSV_THR: a threshold below which more memory should be reserved;
2:  TGT_MEM: the target free size in the top chunk at which the memory reservation
    stops;
3:  TRIM_THR: a threshold above which memory is released;
4:  MEM_CHUNK: memory reserved on each sbrk() call;
5:  top_free: current free memory in the top chunk;
6:  UpdateThreshold( );
7:  if top_free < RSV_THR then
8:      mem_to_reserve ← (TGT_MEM − top_free);
9:      reserved ← 0;
10:     while reserved < mem_to_reserve do
11:         Lock(heap);
12:         address ← sbrk(MEM_CHUNK);
13:         ConstructMapping(address);
14:         reserved ← (reserved + MEM_CHUNK);
15:         Unlock(heap);
16:     end while
17: else if top_free > TRIM_THR then
18:     extra ← (top_free − TRIM_THR);
19:     Lock(heap);
20:     sbrk(−extra);
21:     Unlock(heap);
22: end if
```

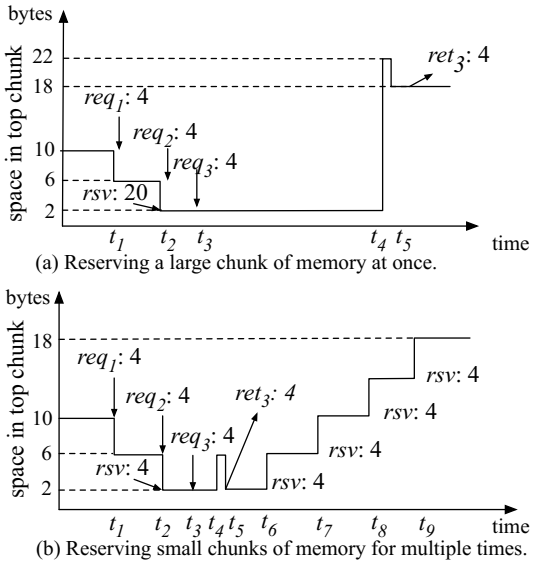


Figure 6: Illustration of gradual reservation.

A naive approach. The challenge of expanding the main heap lies in how to determine the amount of memory to be reserved. Intuitively, simply reserving a large amount of memory at once would boost process performance since the memory is immediately available for processes. However, our experiments find that this approach even degrades the performance of latency-critical services in terms of tail latency. The latency of the default on-demand virtual-physical mapping construction is near proportional to the size of the constructed memory. Since there is only one program break for each process, the manipulation on the program break must be synchronized.

A burst of memory requests in the process thread may be blocked for a long time due to the mapping construction for a large chunk of memory in the management thread. Figure 6 (a) illustrates this scenario. There are initially 10 bytes in the top chunk. At t_1 and t_2, the user process sends two memory requests req_1 and req_2 of 4 bytes, respectively. The requests return immediately. Then, there are only 2 bytes left in the top chunk. The management thread is now invoked to expand the top chunk by 20 bytes and construct the virtual-physical mapping. At t_3, there is another request req_3 of 4 bytes from the user process. Since the running management thread locks the program break, req_3 is blocked. It can only be served at t_5 after the top chunk is expanded at t_4, which incurs significant delay on the request. Although a large number of memory requests do not compete with the main heap expansion, it is the competing ones that lead to prolonged tail latency.

Gradual reservation. We propose gradual reservation that expands the program break by a small size at a time for multiple times (lines 10 ~ 16 in Algorithm 1). For example, instead of expanding the program break for 20 bytes at once, gradual reservation expands the program break for 5 times, each time for 4 bytes, as shown in Figure 6(b). Before req_3 arrives, a reservation of a small memory chunk has already been sent to Linux OS at t_2 by the management thread.

After the reservation returns, req_3 can be immediately served. Finally, the management thread sends four more small reservation operations until the reserved memory reaches 18 bytes. Based on our observation and other studies [11, 26], continuous memory requests from latency-critical services are usually of a similar or constant size. Hermes uses the average memory request size during the previous interval as the size of each memory chunk in gradual reservation. Compared with the default on-demand virtual-physical mapping construction, Hermes serves memory requests faster even if the program break is locked by the management thread, because the virtual-physical mapping construction already starts in advance and returns shortly.

3.2.2 Mmapped Memory Management. Large memory requests are allocated from mmapped memory chunks, as shown in the `yes` branch of the `large_size` statement in Figure 5. Management for mmapped memory is asynchronous since a process can have multiple chunks of mmapped memory space. In other words, the process thread and the management thread can simultaneously allocate two different chunks of mmapped memory space. Thus, incoming requests do not wait on the management thread but uses the default memory allocation routine when the reserved memory is insufficient. Algorithm 2 shows the management routine for mmapped memory. Since the addresses of mmapped memory space are not necessarily adjacent, each chunk of space needs to be managed separately. We use a segregated free list as the memory pool to keep track of the addresses of mmapped memory space (line 14). The function calculates the target bucket based on the size of a mmapped memory chunk using formula 1.

Glibc parameter *min_mmap_size* is the minimum memory request size that can use `mmap` system call, which is 128 KB by default. We use parameter *table_size* to represent the maximum number of buckets in the segregated free list. In implementation, we empirically set *table_size* to 8 (1 MB / 128 KB) since the size of a single memory request is usually less than 1 MB.

$$bucket(chunk\_size) = MIN(\left\lfloor \frac{chunk\_size}{min\_mmap\_size} \right\rfloor, table\_size) \quad (1)$$

Upon a request for a large chunk of memory (i.e., requests ≥ 128 KB) from the process, the modified allocation routine first tries to find the best-fit bucket in the list by calculating the bucket based on the requested size. The hash code of the best-fit bucket is calculated by equation $MIN(bucket(request\_size) + 1, table\_size)$. If there is no such a chunk, the allocation routine uses the largest chunk in the memory pool and expands the chunk to the requested size. If this step still fails due to an empty memory pool, it falls back to the default allocation using `mmap` system call. By this design, the user process gets requested memory immediately as long as they are available while asynchronous shrinking avoids memory wastage. If memory requests are served by expanding an existing small chunk, the delay is still shorter than that of the default allocation routine. The reason is that small chunks already have their virtual-physical mapping constructed. Additional mapping constructions only need to be done for the memory space that exceeds the size of the original memory chunks.

Algorithm 2 Mmap management routine.

```
1:  RSV_THR: a threshold below which more memory is reserved;
2:  TGT_MEM the target free size of mmapped space at which reservation stops;
3:  TRIM_THR: a threshold above which memory is released;
4:  MEM_CHUNK: memory reserved on each mmap() call;
5:  memory_pool a segregated free list that keeps track of the allocated mmapped
    space;
6:  alloc_set: a set of allocated mmapped chunks by the process thread;
7:  DelayRelease(alloc_set);
8:  UpdateThreshold( );
9:  if memory_pool.total_size < RSV_THR then
10:     reserved ← 0;
11:     while reserved < TGT_MEM do
12:         address ← mmap(MEM_CHUNK);
13:         ConstructMapping(address);
14:         memory_pool.add(address);
15:         reserved ← (reserved + MEM_CHUNK);
16:     end while
17: end if
18: while memory_pool.total_size > TRIM_THR do
19:     to_release ← memory_pool.smallest_space;
20:     munmap(to_release);
21: end while
```

3.3 Memory Monitor Daemon

The memory monitor daemon is running on a physical node that adopts job co-location. The daemon keeps the process IDs of latency-critical services in shared memory. The memory management thread adopts a lazy initialization mechanism. When a process detects its process ID is in the shared memory, it initializes the memory management thread. Otherwise, the process behaves as it uses the default Glibc.

Proactive reclamation. The memory monitor daemon is responsible for proactively advising Linux OS to release file cache pages upon memory pressure. The daemon keeps track of all batch jobs and their loaded data files. When the system memory usage exceeds threshold `adv_thr`, the monitor daemon advises Linux OS to release file cache pages in a largest-file-first order until the percentage of file cache drops below the threshold or no file cache is from the specified batch jobs. The largest-file-first paging order makes a large chunk of memory available at once for latency-critical services. It also reduces the number of calls to the advising routine.

Proactive reclamation is an effective approach to accelerating memory allocation. Although Hermes reserves physical memory in advance, the reservation can still be delayed if it triggers the direct reclaim routine due to insufficient memory. Proactive reclamation reduces the chance by which the direct reclaim routine is triggered. Note that solely relying on proactive reclamation is insufficient since it only tries to make free space for new memory requests but it does not contribute to virtual-physical mapping construction.

4 IMPLEMENTATION

We implement Hermes in Glibc-2.23 with about 1,200 lines of C code. We empirically set the invocation interval (f) of the memory management thread to 2 ms. There could be transient changes in terms of memory space of a process during the interval. In the next wakeup, the monitoring thread is able to capture the change and the memory management thread can manage the available memory accordingly. Recall that we use a reservation factor *RSV_FACTOR* to determine the amount of memory to be reserved. A larger value results in more reserved memory and faster memory allocation.

However, the reserved memory is wasted if it is never used by latency-critical services. In the rest of the paper, we set this value to 2 if not otherwise specified, which balances between memory allocation speed and memory wastage. We also set the minimum amount of memory *min_rsv* that should be reserved after each execution of the management thread even if there is no newly incoming memory request. It allows that a burst of memory requests after an idle period can be quickly served. The value depends on the characteristics of latency-critical services. Empirically, we set this value to 5 MB. We use `mlock` system call to delegate virtual-physical mapping construction to kernel space.

There are two choices to implement the virtual-physical mapping construction function, 1) iterating through the allocated virtual memory addresses and filling them with '0', and 2) using the `mlock` system call to delegate the construction to the kernel space. We choose the second one for two reasons. First, our experiments find that using `mlock` system call is at least 40% faster than the iteration approach for both heap memory and mmapped memory. Second, the `mlock` system call guarantees newly reserved physical memory not to be swapped into disks, which further accelerates memory allocation. After a chunk of reserved memory is allocated to a process, the `munlock` system call will be called on that address space to allow swapping on the chunk.

The memory monitor daemon takes about 500 lines of C code. It is responsible for bookkeeping latency-critical services and batch jobs, and advising Linux OS to release file cache pages. It communicates with the modified Glibc with a shared memory area. Specifically, it uses the shared memory to store all the process IDs of latency-critical services specified by a system administrator. With the modified Glibc, a process examines whether its process ID is in the shared memory. If so, the modified Glibc initializes the memory management thread. When a process is no longer a latency-critical service, the administrator can simply remove its process ID. The monitor daemon keeps track of the data files loaded by batch jobs by calling the `lsof` command. It uses the C library call `posix_fadvise()` to release file cache pages, which is a wrapper function of the underlying system call `fadvise64()`. Hermes then adopts the default memory management in Glibc for this process.

Hermes is open sourced at https://github.com/EddiePi/Hermes.

5 EVALUATION

5.1 Evaluation Setup

We use both a micro benchmark and real-world latency-critical services to evaluate the performance of Hermes, and compare it to Glibc, jemalloc [21], and TCMalloc [6]. Glibc is the most popular memory allocator in C/C++. Jemalloc is the default memory allocator for Redis [4]. TCMalloc is Google's customized implementation of `malloc()` function. All experiments are executed on a server that has two 2.4 GHz 8-core Intel Xeon E5-2630 CPUs, 128 GB DRAM, and 2 TB 7200 rpm HDD disks. The server is installed with Ubuntu 16.04 with Linux kernel-4.4.0. For all experiments, we pin latency-critical services and background processes onto different cores to avoid CPU interference.

Micro benchmark. We implement a micro benchmark in C, which continuously calls `malloc` function to request memory until the total amount of requested memory reaches a specified threshold. We run the experiments in two settings referred as dedicated system and memory pressure. For the dedicated system setting, we run the micro benchmark alone on the nodes with sufficient memory. For the memory pressure setting, we generate the memory pressure for the micro benchmark by loading the node with either anonymous pages or file cache pages. We measure the memory allocation latency due to the three approaches.

Real-world services. We evaluate Redis [4] and Rocksdb [5] as real-world latency-critical services under different memory pressure levels in Section 5.3. We measure three metrics in the experiments: 1) query latency of latency-critical services, 2) SLO violation of latency-critical services, and 3) throughput of batch job. The memory pressure is computed as `virtual memory of batch job / memory capacity of the server`. To generate different levels of memory pressure, we configure the maximum logically available memory of batch jobs to 50%, 75%, 100%, 125% and 150% of the memory capacity of the node. For example, on a node with 128 GB DRAM, 150% memory pressure level suggests batch jobs can oversubscribe 192 GB (128 GB $\times$ 1.5) of DRAM. In addition, we also conduct the experiment on a dedicated server, i.e., 0% memory pressure level.

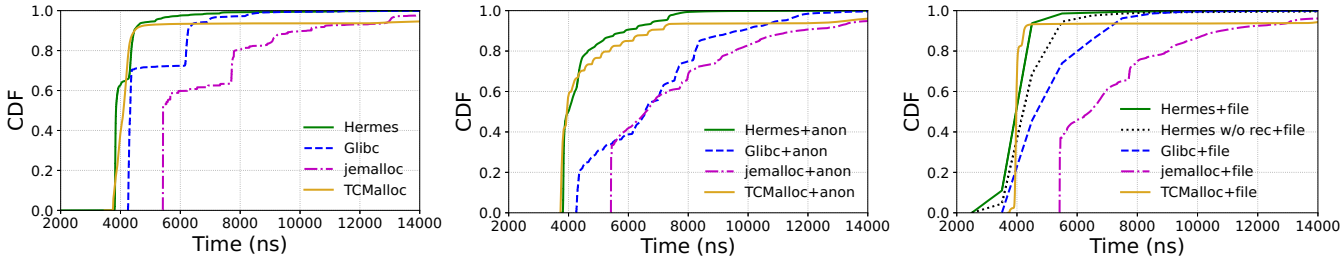
Parameter sensitivity. We conduct experiments to evaluate parameter sensitivity in Section 5.4. Specifically, we run the micro benchmark and evaluate its latency under different values of reservation factor *RSV_FACTOR*. We evaluate the overhead of Hermes in Section 5.5.

5.2 Micro Benchmark

We evaluate the performance of Hermes under three scenarios: a dedicated system with sufficient memory, anonymous page pressure, and file cache pressure. Under file cache pressure, we also show the performance of Hermes when it is disabled with proactive reclamation, denoted as "Hermes w/o rec', to demonstrate the performance gain due to proactive reclamation. The anonymous page pressure is made by a process that keeps allocating memory until the system available memory drops below 300 MB. The file cache pressure is made by a process that repeatedly reads 10 GB files and occupies the rest of the system memory with anonymous pages. We develop the micro benchmark by continuously sending fix-sized memory requests until the total requested memory reaches 1 GB. We use 1KB-size and 256KB-size memory requests to evaluate the allocation latency of heap memory and mmapped memory.

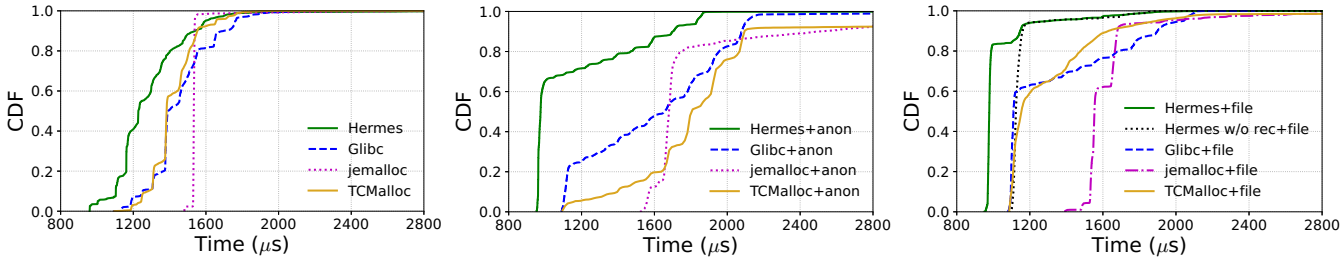
Figure 7(a)-(c) and Figure 8(a)-(c) show the CDFs of memory allocation latency of 1KB-size and 256KB-size requests under a dedicated system, anonymous page pressure ("+anon" suffix), and file cache pressure ("+file" suffix), respectively. For small memory requests, Hermes achieves the lowest latency at every percentile compared to Glibc and jemalloc in all three cases. TCMalloc presents low latency on average. However, it has very high tail latency in all three cases. As for large memory requests, jemalloc presents longer but more stable latency under a dedicated system. However, Hermes outperforms both Glibc, jemalloc and TCMalloc when the system is under memory pressure. Jemalloc and TCMalloc present very long tail latency under memory pressure.

Specifically, we show the latency reduction of Hermes at each percentile for small requests and large requests compared to Glibc



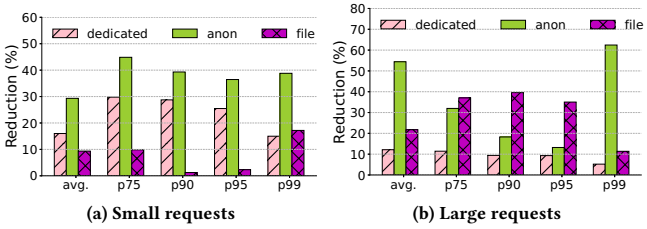
(a) Dedicated system. (b) Anonymous pages pressure. (c) File cache pressure.

Figure 7: The memory allocation latency for small (1KB-size) memory requests.



(a) Dedicated system. (b) Anonymous pages pressure. (c) File cache pressure.

Figure 8: The memory allocation latency for large (256KB-size) memory requests.



(a) Small requests (b) Large requests

Figure 9: The latency reduction by Hermes.

in Figures 9, since Glibc outperforms jemalloc in most cases. For 1KB-size requests, Hermes reduces the average latency by 16.0%, 29.3%, 9.4%, and the 99^{th} percentile latency by 15.0%, 38.8%, 17.2% in the three scenarios, respectively. For 256KB-size requests, Hermes reduces the average latency by 12.1%, 54.4%, 21.7%, and the 99^{th} percentile latency by 5.2%, 62.4%, 11.4%, respectively. Hermes outperforms the default Glibc at each percentile in all scenarios. The allocation latency is as low as $4\mu s$ for small requests and $1ms$ for large requests. For 1KB-size requests, `brk` is called 1,053,952 times. For 256KB-size request, `mmap` is called 4099 times.

By comparing the "dedicated" and "file" bars in Figure 9(a) to those in Figure 9(b), the performance gain by Hermes under a dedicated system and under file cache pressure for large requests is more significant than that for small requests. The reason is that large requests take a long time to be allocated in the default Glibc. Hermes allocates the requests and constructs the virtual-physical mapping in advance. Thus, memory is immediately available for incoming requests. By comparing the "anon" bar to the "dedicated" and "file" bars in Figure 9(a) or Figure 9(b), we observe that Hermes generally achieves more performance improvement under anonymous pressure for both small and large requests compared to those under file cache pressure. The reason is that it is faster to reclaim file cache pages in the default Linux kernel since unmodified file cache pages are directly released without I/O operations. For anonymous pages, however, each of them must be swapped into disks before released, causing much longer delay due to I/O operations.

Proactive reclamation. Figures 7c and 8c show that "Hermes w/o paging" achieves similar memory allocation latency at low percentiles compared with the default Glibc, but it significantly reduces the latency at high percentiles. Full Hermes further improves the average latency over "Hermes w/o paging".

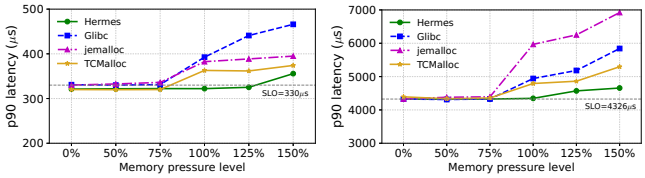
5.3 Two Real-world Latency-critical Services

5.3.1 Query latency and SLOs. We evaluate the query latency reduction on real-world latency-critical services by Hermes compared to Glibc, jemalloc and TCMalloc under different memory pressure. We use Redis-5.0.5 [4] and Rocksdb-6.4.0 [5] as two representative real-world services. Redis is an in-memory key-value store for fast data access. Rocksdb is a disk-based persistent key-value store for fast storage environments. It uses memory as data cache. These services are usually used for intermediate or temporary data storage. Thus, they frequently allocate and release memory.

For both Redis and Rocksdb, we implement a program to continuously generate requests. One request consists of one insertion

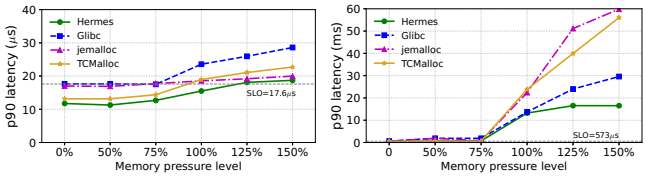
Table 1: The number of system calls invoked.

	Redis 1KB	Redis 200KB	Rocksdb 1KB	Rocksdb 200KB
brk	0	0	52,429	5
mmap	35	8	34	8,397



(a) Small requests (b) Large requests

Figure 10: The 90^{th} percentile query latency of Redis.



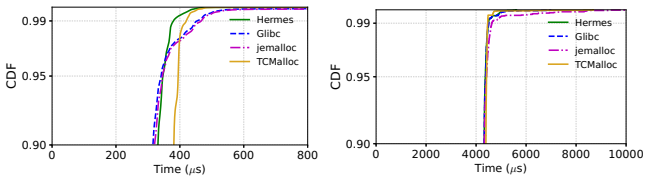
(a) Small requests (b) Large requests

Figure 11: The 90^{th} percentile query latency of Rocksdb.

operation followed by one read operation. We use 1KB-size and 200KB-size data records referenced as small and large memory requests, respectively. For each data insertion execution, we insert the data until it reaches 2 GB. Table 1 summarizes the number of the two system calls invoked during the insertion exectuion. To inject memory pressure, we run Spark Kmeans and Spark PageRank as batch jobs on the host node. The jobs are from HiBench-6.0 [27] using its default huge data size. We run Spark-2.3.0 on Hadoop-2.7.3 [41].

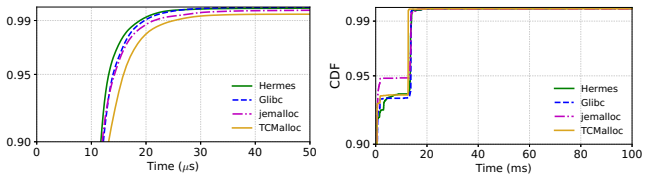
Since there is not a magic value to define the SLO of each service, we adopt the 90^{th} percentile latency by the default Glibc under a dedicated system (w/o memory pressure) as the SLO, which is a rather strict value. The rational is that latency-critical services like web search commonly distribute requests across many servers. The end-to-end response time is determined by the slowest individual latency [9, 20, 22, 48]. Thus, the 90^{th} percentile latency is a critical metric in measuring the SLO of latency-critical services.

Latency reduction. Figures 10 and 11 show the 90^{th} percentile query latency under different memory pressure levels for Redis and Rocksdb, respectively. Under memory pressure level 0%, 50% and 75%, memory is not a scarce resource. Under memory pressure level 100%, 125% and 150%, memory become a scarce resource. The horizontal dash line represents the target SLO in each situation. In Redis, the SLOs are 330μs and 4, 326μs for small and large requests, respectively. In Rocksdb, the SLOs are 17μs and 573μs for small and large requests, respectively.



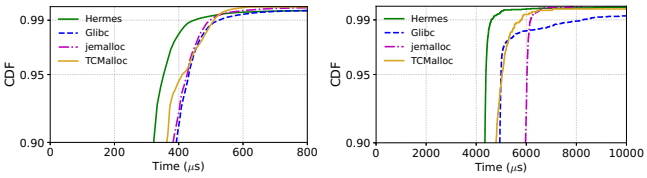
(a) Small requests w/o batch jobs (b) Large requests w/o batch jobs

Figure 12: Redis latency under a dedicated system.



(a) Small requests w/o batch jobs (b) Large requests w/o batch jobs

Figure 13: Rocksdb latency under a dedicated system.

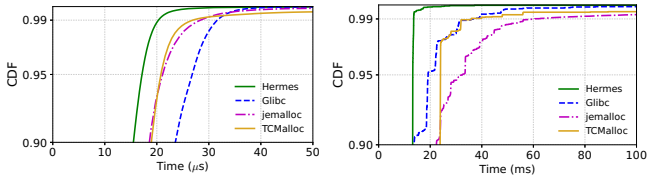


(a) Small requests w/ batch jobs (b) Large requests w/ batch jobs

Figure 14: Redis latency under 100% memory pressure.

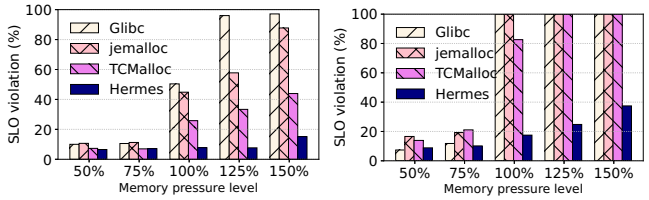
The results show that Hermes outperforms Glibc, jemalloc and TCMalloc in reducing the 90^{th} percentile query latency in all scenarios for both Redis and Rocksdb. Specifically, with a dedicated system (0% memory pressure) or a low memory pressure level (50% and 75%), Hermes achieves similar or slightly lower 90^{th} percentile latency compared to Glibc, jemalloc and TCMalloc. With a moderate memory pressure level (100% and 125%), Hermes can meet the SLO targets for small requests while Glibc, jemalloc and TCMalloc incur significant SLO violation. With a severe memory pressure level (> 125%), all three approaches incur non-trivial SLO violation but Hermes significantly outperforms the others. We observe that large requests in Rocksdb under high memory pressure experience tens of milliseconds of latency. Note that Rocksdb is a disk-based KV store with memory cache. Under severe memory pressure, data are written into disks more frequently, causing high latency.

Under a dedicated system and job co-location. We evaluate the performance of real-world services Redis and Rocksdb by Hermes, Glibc and jemalloc under a dedicated system with sufficient memory. Figures 12 and 13 plot the CDF of the query latency for the two real-world services under the dedicated system, respectively. Compared to Glibc and jemalloc, Hermes renders similar or slightly better average, 90^{th}, and 99^{th} percentile query latency.



(a) Small requests w/ batch jobs (b) Large requests w/ batch jobs

Figure 15: Rocksdb latency under 100% memory pressure.



(a) Small requests (b) Large requests

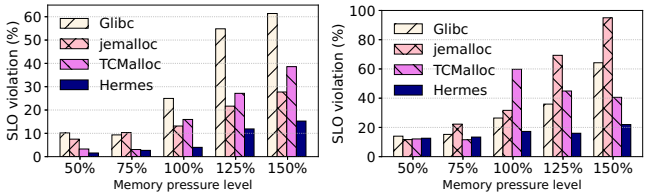
Figure 16: The SLO violation ratio of Redis requests.

Under job co-location, severe memory pressure is usually addressed by a system administrator while memory pressure around the 100% level is more likely to happen due to the dynamic memory consumption of latency-critical services and batch jobs. Thus, we plot the CDF of the query latency under such a scenario for Redis and Rocksdb in Figures 14 and 15, respectively. Hermes achieves the lowest latency for both services. Compared to Glibc, it reduces the average (99^{th} percentile) latency by up to 17.0% (40.6%) for Redis and 20.6% (63.4%) for Rocksdb.

SLO violation. Figure 16 and Figure 17 show the ratios of SLO violation with Hermes, Glibc, jemalloc and TCMalloc under different memory pressure levels for Redis and Rocksdb, respectively. For Redis, Hermes achieves the SLO violation ratio lower than 10% under a low memory pressure level (i.e., 50% and 75%). The results for Rocksdb are similar. The reason is that Hermes builds the virtual-physical mapping in advance such that incoming memory requests can be immediately served. The most significant results are those under 100% or higher memory pressure levels which usually happen in a multi-tenant system. Under such a memory pressure level, compared to the default Glibc, jemalloc, and TCMalloc, Hermes reduces the SLO violation of Redis by up to 83.6%, and reduces the SLO violation of Rocksdb by up to 84.3%.

5.3.2 Batch job throughput. We examine the throughput of batch jobs co-located with latency-critical services. We submit Spark Kmeans jobs and keep three concurrent job instances in the node. Each Kmeans job runs in eight Yarn containers and requests around 40GB memory. This generates the 100% memory pressure level. We send data insertion, read, and deletion requests to the latency-critical services such that stored data size varies from 20GB to 40GB. The co-location experiment runs for 24 hours in each of the three scenarios: Default, Hermes, and Killing.

- **Default.** We co-locate batch jobs and latency-critical services with the default GNU/Linux stack.



(a) Small requests (b) Large requests

Figure 17: The SLO violation ratio of Rocksdb requests.

Table 2: The throughput of batch jobs.

	Default	Hermes	Killing	Dedicated
Redis	212	194	123	N/A
Rocksdb	380	364	267	N/A

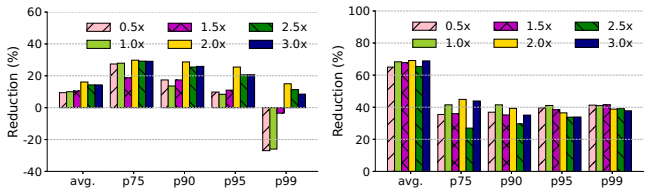
- **Hermes.** We co-locate batch jobs and latency-critical services with Hermes.
- **Killing.** Upon Default, we kill the latest launched container of a batch job when node memory is insufficient, which frees up memory. Killing the container results in the least progress loss of the batch job.

Table 2 gives the number of the finished batch jobs in the three co-location scenarios as well as in a dedicated system where there is no throughput of batch jobs. Both Default and Hermes achieve much higher throughput than that of Killing. Hermes achieves slightly lower throughput to that of Default. In return, it significantly reduces the query latency and SLO violation of latency-critical services, the principle requirement of job co-location. We notice that the throughput of co-location with Rocksdb is higher than that of Redis. The reason is that Redis is a memory-based KV store that keeps all data in DRAM. Rocksdb is a disk-based KV store that has much lower memory consumption than Redis. Thus, more memory can be allocated to batch jobs. Experimental results find that job co-location due to Hermes renders about 98.5% average node memory utilization.

5.4 Parameter Sensitivity

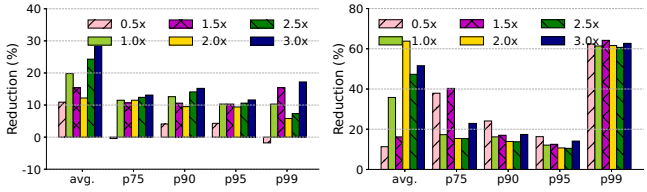
We evaluate the impact of parameter sensitivity. Specifically, we change the value of reservation factor *RSV_FACTOR* raging from 0.5 to 3, and evaluate the memory allocation latency under each value for both small and large memory requests using the micro benchmark. We run the micro benchmark under a dedicated system and under anonymous page pressure, respectively. We use the same settings as those in Section 5.2 to generate the memory pressure. Figures 18 and 19 show the percentage of latency reduction at specific percentiles for small and large requests, respectively.

Under a dedicated system, a small value of *RSV_FACTOR* significantly increases the 99^{th} percentile tail latency for small requests, as shown in Figure 18(a). The reason is that the reserved memory under such a *RSV_FACTOR* value is too small. When a burst of memory requests are sent by the processes, the reserved memory quickly runs out. In this case, the incoming memory requests are blocked by the memory reservation routine. As the value of



(a) Dedicated system (b) Anonymous pressure

Figure 18: Latency reduction for small requests.



(a) Dedicated system (b) Anonymous pressure

Figure 19: Latency reduction for large requests.

RSV_FACTOR is increased, the 99^{th} percentile tail latency becomes better than that by the default Glibc. For large memory requests, the incoming memory requests are not blocked but served by the default allocation routine in Glibc since there can be multiple mmapped memory chunks for a process. Thus, Hermes achieves more allocation latency reduction for large requests under a dedicated system as shown in Figure 19(a).

Under anonymous page pressure, Hermes achieves much more significant latency reduction compared with that under a dedicated system. Specifically, it reduces the average and the 99^{th} percentile latency by up to 69.1% and 41.6% for small requests, respectively. It reduces the average and the 99^{th} percentile latency by up to 63.8% and 64.2% for large requests, respectively. Overall, setting *RSV_FACTOR* to a value larger than 2 does not achieve more performance gain. The reserved memory exceeds the total amount of memory requests and causes more memory wastage. Thus, we empirically set the value of *RSV_FACTOR* to 2 since it achieves good reduction in the memory allocation latency while resulting in the least memory wastage.

5.5 Hermes Overhead

Hermes introduces about 0.4% CPU usage overhead to the application due to the management thread in the modified Glibc. The downsides using Hermes is that there is some memory which is reserved but not actually used by the application. We profile such memory in the micro benchmark for both small (1KB-size) and large (256KB-size) memory requests. The reserved memory at runtime is about 6 MB~6.4 MB, which is negligible compared to the memory capacity of a physical node. In addition, the memory monitor daemon requires about 2 MB memory including the memory occupied by the daemon process and the shared memory space. It introduces about 2.4% CPU usage as it keeps monitoring the latency-critical services and available memory in the OS.

6 DISCUSSIONS

Reservation factor. Users need to set an appropriate value for the reservation factor *RSV_FACTOR* in Hermes. We find that a value of 2 achieves good performance gain for both the micro benchmark and two real-world services while introducing the least memory wastage. However, the value setting depends on various factors such as the characteristics of latency-critical services and the underlying multi-tenant system. If a latency-critical service does not require much memory at runtime, *RSV_FACTOR* can be set to a small value. Otherwise, it should be set to a relatively large value.

Reservation heuristic. Hermes relies on two simple but effective heuristics, heap management routine and mmap management routine, to reserve memory for latency-critical services. Since Hermes is developed at the library level, the reservation heuristics incur low overhead. Users can also implement their own heuristics.

Query latency. Hermes aims for fast memory allocation. Once the reserved pages are obtained by a process, Hermes calls `munlock` system call on the pages. The pages can be swapped into disks when the available system memory is low. Queries to the latency-critical services will be delayed if the physical pages reside in the swap area. A simple solution is to return the pages to a process without calling `munlock`. In this case, the pages occupied by latency-critical services are never swapped, resulting in low latency for queries. The simple solution meets the design goal that batch jobs should not affect the performance of latency-critical services. However, it may incur out-of-memory errors if the locked memory is not well managed under extreme memory pressure. Thus, no page is eligible for reclaim but killing processes is the only choice.

Default Linux mechanisms. Linux provides mechanisms by which applications can instruct the system to construct the virtual-to-physical memory mapping. For example, the `MAP_POPULATE` flag in the `mmap()` system call and the `mlock()` system call. However, using these mechanisms in applications has two drawbacks. First, using these mechanisms requires modification to application source code. By doing so, applications bypass the memory management routine in the library and need to manage the memory by themselves, which puts much more burden on software developers. Second, when an application tries to allocate memory under memory pressure, using these mechanisms does not help accelerate memory allocation since Linux OS still needs to reclaim/swap physical memory before the new allocation request.

Cgroups and VMs. Linux provides the cgroup mechanism to control resource utilization of processes. Its memory subsystem can be used to dynamically set memory limits for batch jobs. However, there are two limitations by using the cgroup mechanism. First, after the memory of batch jobs is reclaimed by setting a smaller limit in cgroup, the reclaimed memory can be allocated to multiple latency-critical services. This may lead to memory competition and degrades latency. Second, the cgroup mechanism cannot proactively construct the virtual-to-physical mapping. The idea of Hermes could be applied to cgroup such that it allows prioritization of memory allocation between cgroups. However, it requires OS-level modification. In this paper, latency-critical services and batch jobs run in the same OS. Hermes is still effective if latency-critical services and batch jobs run in separate VMs on the same host server since it reserves physical memory for latency-critical services.

Fragmentation. The current Glibc does not round up the size of heap memory chunks to power of two. Thus, freed memory chunks of any size can be coalesced to neighboring chunks, which does not incur high memory waste through fragmentation. Hermes inherits the heap management algorithm from Glibc for small memory requests allocated from heap. Thus, the impact of fragmentation on heap memory is the same as that in Glibc. Hermes uses its own segregated free list to manage large memory chunks allocated by `mmap` system call. Since most memory requests from latency-critical services are of the same size, freed large memory chunks may exactly fit incoming requests, incurring no fragmentation. In the worst case where significant memory waste through fragmentation occurs, memory compaction can be done through `mremap` system call. This is a rare case since modern CPUs support hundreds of gigabytes of memory address space.

Applicability. Currently, Hermes supports C/C++ programs. Many popular key-value stores [2–5] are implemented in C/C++. A potential limitation of using Hermes is that it might clash with other C/C++ libraries. In the future, we plan to do more compatibility experiments for Hermes. The principle and design of Hermes can be applied to other language runtimes. For example, for programs running on Java Virtual Machines (JVMs), JVMs could reserve a chunk of memory and construct the virtual-physical mapping in advance for fast memory allocation.

7 RELATED WORK

Latency reduction. There are extensive efforts on reducing query latency for latency-critical services [9, 14, 15, 19, 23, 25, 26, 29, 32, 34, 35, 48]. Tail-control [34] develops a work-stealing scheduler for optimizing the number of requests that meet a target latency. MittOS [26] tackles the tail latency for distributed file systems where the bottleneck is disk I/O. SDChecker [15] is a tool that characterizes scheduling delay for low-latency data analytics workloads. FastTrack [25] targets mobile devices and improves the response time for foreground apps. PerfIso [29] is an approach that reserves CPU slacks to achieve efficient CPU sharing between latency-critical services and batch jobs. RobinHood [9] dynamically reallocates the cache between cache-rich and cache-poor applications. CurtailHDFS [35] manages the tail latency in distributed file systems. Hermes aims to reduce the latency in the memory allocation phase for latency-critical services in multi-tenant systems and achieve significantly lower tail query latency and higher throughput.

Cluster resource sharing. Modern cluster schedulers [12, 18, 31] launch best-effort jobs with transient resources in a cluster. For example, Apollo [12] is a scalable scheduling framework for cloud computing. Mercury [31] launches jobs with transient resources and kills the jobs when the available resources drop below a threshold. Pado [45] is a data processing engine that aims to harness transient resources. Mos [8] analyzes cloud object stores and proposes independent microstores for the needs of particular types of workloads. Big-C [18] is a preemption-based cluster scheduler that achieves low scheduling latency for heterogeneous workloads. Caladan [22] exclusively relies on CPU scheduling to mitigate resource interference and achieve better QoS in a co-located environment. Hermes targets efficient co-location and specifically efficient memory sharing in physical nodes.

Memory management. Study [36] designs a buffer pool for relational databases in a multi-tenant environment. There are studies on efficient memory management for applications running in JVMs by leveraging the runtime characteristics of applications [13, 17, 24, 37, 38, 46]. Broom [24] proposes to use region based memory management for data analytics applications. Facade [38] statically bounds the number of objects allocated to threads for efficient memory management in JVMs. Yak [37] creates a new region in the JVM heap to store long-lived data. ROLP [13] is an object lifetime profiler for efficient garbage collection. Pufferfish [17] is an elastic memory manager that leverages containers to flexibly allocate memory for data-intensive applications. Charon [16] is a cluster scheduler for oversubscription of opportunistic memory in an on-demand manner that leverages an OS-augmented and user-assisted OutOfMemory killer. Hermes focuses on fast memory allocation for latency-critical services that use C libraries.

Memory allocation libraries. GNU C Library provides ptmalloc [1] as the memory allocator for C/C++ programs. There are other memory allocators [6, 10, 21, 28, 39] that focus on different design objectives. Jemalloc [21] emphasizes fragmentation avoidance. It is the default memory allocator for FreeBSD [7]. Hoard [10] is a scalable memory allocator that largely avoids false sharing and better worst case fragmentation. TCMalloc [6] supports efficient memory allocation for multi-thread processes. McRT-Malloc [28] and SFMalloc [39] focus on non-blocking scalable memory allocation. Especially, McRT-Malloc is specialized for transactional memory while SFMalloc is a general purpose memory allocation. Both memory allocators support concurrent memory manipulation by multiple threads. The major difference between the existing memory allocators and Hermes is that, only Hermes explicitly focuses on physical memory management while the other memory allocators focus on virtual memory management. In addition, Hermes is aware of and mitigates the cost in building virtual-physical mapping. Although Hermes is implemented in Glibc, its principle and the idea of proactively building virtual-physical mapping can be integrated to those memory allocators.

8 CONCLUSION

We present Hermes, a library-level mechanism that enables fast memory allocation for latency-critical services in multi-tenant servers. Hermes constructs the virtual-physical address mapping in advance and quickly serves incoming memory requests from latency-critical services. It proactively advises Linux OS to release file cache occupied by batch jobs so as to make available memory without going through the slow memory reclaim routine. Hermes is implemented in GNU C Library. Experimental results show that Hermes significantly reduces the average and the tail latency of queries for latency-critical services especially under memory pressure, and improves system throughput and memory utilization.

In the future, we plan to extend the principle and design of Hermes to language runtimes Java and Scala.

9 ACKNOWLEDGMENT

This research was supported in part by U.S. NSF grant CCF-1816850. We thank our shepherd Dr. Sameh Elnikety and the anonymous reviewers for their valuable comments.

REFERENCES

[1] GNU C Library. https://www.gnu.org/software/libc/.
[2] Memcached. https://memcached.org/.
[3] MongoDB. https://www.mongodb.com/.
[4] Redis. https://redis.io/.
[5] Rocksdb. https://rocksdb.org/.
[6] TCMalloc. https://gperftools.github.io/gperftools/tcmalloc.html.
[7] The FreeBSD Project. https://www.freebsd.org/.
[8] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proc. of ACM HPDC*, 2016.
[9] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-balter, and Siddhartha Sen. Robinhood: Tail latency-aware caching – dynamically reallocating from cache-rich to cache-poor. In *Proc. of USENIX OSDI*, 2018.
[10] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocatorfor multithreaded applications. In *Proc. of ACM ASPLOS*, 2000.
[11] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, 1994.
[12] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proc. of USENIX OSDI*, 2014.
[13] Rodrigo Bruno, Duarte Patricio, Jose Simao, Luiz Veiga, and Paulo Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proc. of ACM EuroSys*, 2019.
[14] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *Proc. of USENIX FAST*, 2020.
[15] Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou. Characterizing scheduling delay for low-latency data analytic workloads. In *Proc. of IEEE IPDPS*, 2018.
[16] Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou. Os-augmented oversubscription of opportunistic memory with a user-assisted oom killer. In *Proc. of ACM/IFIP Middleware*, 2019.
[17] Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou. Pufferfish: Container-driven elastic memory management for data-intensive applications. In *Proc. of ACM SoCC*, 2019.
[18] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proc. of USENIX ATC*, 2017.
[19] Youmin Chen, Lu Youyou, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proc. of ACM ASPLOS*, 2020.
[20] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of ACM*, 56(2):74–80, February 2013.
[21] Jason Evans. A scalable concurrent mallloc(3) implementation for freebsd. In *Proc. of BSDCan*, 2006.
[22] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proc. of USENIX OSDI*, 2020.
[23] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. Evendb: optimizing key-value storage for spatial locality. In *Proc. of ACM Eurosys*, 2020.
[24] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *Proc. of USENIX HotOS*, 2015.
[25] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong King. Fasttrack: Foreground app-aware i/o management for improving user experience of android smartphones. In *Proc. of USENIX ATC*, 2018.
[26] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, and Riza O. Suminto. Mittos: Supporting millisecond tail tolereance with fast rejecting slo-aware os interface. In *Proc. of ACM SOSP*, 2017.
[27] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proc. of IEEE Data Engineering Workshops (ICDEW)*, 2010.
[28] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mcrt-malloc: A scalable transactional memory allocator. In *Proc. of ACM ISMM*, 2006.
[29] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Semeh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *Proc. of USENIX ATC*, 2018.
[30] Seyyed Ahmad Javadi and Anshul Gandhi. Dial: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing. In *Proc. of IEEE ICAC*, 2017.
[31] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of USENIX ATC*, 2015.
[32] Marios Kogias and Edouard Bugnion. Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proc. of ACM Eurosys*, 2020.
[33] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proc. of ACM Eurosys*, 2014.
[34] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. Work stealing for interactive services to meet target latency. In *Proc. of ACM PPoPP*, 2016.
[35] Pulkit A. Misra, Maria F. Borge, Inigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proc. of ACM EuroSys*, 2019.
[36] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chudhuri. Sharing buffer pool memory in multi-tenant relational databases-as-a-service. In *Proc. of VLDB*, 2015.
[37] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proc. of USENIX OSDI*, 2016.
[38] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proc. of ACM SOSP*, 2015.
[39] Sangmin Seo, Junghyun Kim, and Jaejin Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *Proc. of IEEE PACT*, 2011.
[40] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Metzler Bernard. Unification of temporary storage in the nodekernel architecture. In *Proc. of USENIX ATC*, 2019.
[41] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of ACM SoCC*, 2013.
[42] MidHul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *Proc. of USENIX NSDI*, 2020.
[43] Hailong Yang, Alex Breslow, Jason Mars, and Tang Lingjia. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proc. of ACM ISCA*, 2013.
[44] Xi Yang and Stephen M. Blackburn. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *Proc. of USENIX ATC*, 2016.
[45] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. Pado: A data processing engine for harnessing ransient resources in datacenters. In *Proc. of ACM EuroSys*, 2017.
[46] Junxian Zhao, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou. Flashbyte: Improving memory efficiency with lightweight native storage. In *Proc. of IEEE/ACM CCGrid*, 2021.
[47] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-criticaltasks on shared multicore systems. In *Proc. of ACM ASPLOS*, 2016.
[48] Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. Workloadcompactor: Reducing datacenter cost while providing tail latency slo guarantees. In *Proc. of ACM SoCC*, 2017.