Towards Increased Datacenter Efficiency with Soft Memory

Megan Frisella* — Brown University
Shirley Loayza Sanchez* — Brown University
Malte Schwarzkopf — Brown University

Abstract

Memory is the bottleneck resource in today's datacenters because it is inflexible: low-priority processes are routinely killed to free up resources during memory pressure. This wastes CPU cycles upon re-running killed jobs and incentivizes datacenter operators to run at low memory utilization for safety. This paper introduces *soft memory*, a software-level abstraction on top of standard primary storage that, under memory pressure, makes memory revocable for reallocation elsewhere. We prototype soft memory with the Redis key-value store, and find that it has low overhead.

ACM Reference Format:
Megan Frisella, Shirley Loayza Sanchez, and Malte Schwarzkopf. 2023. Towards Increased Datacenter Efficiency with Soft Memory. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595902

1 Introduction

Memory is the bottleneck resource in many of today's data centers [22, 24]. This is because memory is in high demand, but also *inflexible*, satisfying developer expectations of persistent allocations during program execution. Processes may consume large amounts of memory and must explicitly free memory before the OS can re-assign it to another process. However, typical processes only actively use a small fraction of their allocated memory at any time; many of them have cold memory or extensive application-specific caches [12]. The inflexibility of memory allocation means that infrequently-accessed or unimportant memory can't be re-purposed to needier applications.

A common way to make memory flexible is to swap out memory content to more abundant, higher-latency storage, such as flash or disk. This temporarily frees up space for new allocations until the program swaps old memory back when accessing its content. However, swapping is transparent to the application, so it introduces performance non-determinism because developers cannot detect when the program has swapped out pages and, e.g., follow a less aggressive caching strategy.

This paper proposes *soft memory*, an opt-in software-level abstraction on top of standard primary storage, which makes memory allocations revocable under memory pressure for reallocation in other applications. Soft memory investigates what would happen if memory was a more fungible resource—like CPU or I/O time, resources that OS kernels reallocate dynamically between processes—and what abstractions are needed to make this work. Soft memory differs from swapping by actually *revoking* and dropping memory contents, rather than moving them to slower storage. This makes sense when the data stored loses its utility once no longer in memory, as, e.g., with in-memory caches.

Soft memory has two major benefits. First, it reduces the number of low-priority job terminations due to memory pressure. Such *evictions* waste cluster resources on incomplete executions, including resources scarcer than memory, such as accelerators (e.g., GPUs, TPUs). Second, rather than failing `malloc` when there is insufficient memory on a machine, soft memory allows an allocator to retrieve it from other processes. Since most applications cannot handle failed memory allocations gracefully, soft memory avoids crashes by satisfying the immediate memory need.

In a data center that deploys soft memory, a developer uses soft memory for caches, look-up tables, temporary requests queues, and data structures with similar non-essential purposes. They continue to use standard memory to store critical state that enables the correct functioning of the application (i.e., authentication records, data structure metadata, etc.). For example, the entries in database caches may employ soft memory, but table schemas and active user metadata would remain in traditional memory.

In large-scale computing clusters, jobs typically have a memory limit above which they get terminated by the scheduler. Granting a soft memory budget on top of the traditional memory limit achieves two goals: first, it allows productive use of memory left idle because other jobs are operating below their limit; and second, it incentivizes the use of soft memory, which provides additional, free resources to an application. The scheduler can continue to constrain traditional

*Both authors contributed equally to the work.

HotOS '23, June 22–24, 2023, Providence, RI, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0195-5/23/06...$15.00
https://doi.org/10.1145/3593856.3595902

memory by maintaining limits on it, but developers are encouraged to use soft memory to take maximum advantage of extra available resources.

We build a prototype Soft Memory Allocator (SMA) that manages soft memory on an application level, Soft Data Structures (SDSs) that take care of soft allocation and freeing under the hood and offer familiar data structure APIs, and a Soft Memory Daemon (SMD) that serves as a machine-wide memory manager for soft memory requests. We add soft memory support to the Redis key-value store to investigate the practicality and performance of soft memory. Preliminary results show comparable allocation performance to traditional allocators without memory pressure and low memory reclamation overheads during memory pressure.

2 Why Soft Memory

Soft memory presents an opportunity to improve resource management under widespread data center characteristics.

Memory Under-utilization. Cluster schedulers allocate initial resources to jobs in response to requirements specified by the developer or to predicted peak load. However, engineers are notoriously bad at estimating actual resource consumption and workload requirement estimation remains fairly conservative in data centers as deployments provision for peak load [3, 19]. This leads to widespread resource under-utilization, as identified in large computing cluster traces [4, 14, 22]. At the core of the problem, there is a trade-off between maximizing the use of hardware resources and avoiding performance degradation and disruption when requests exceed availability. Large-scale schedulers such as Google's Borg decide to terminate lower-priority jobs when they receive memory requests that cannot be satisfied otherwise [23]. This is undesirable, as often work completed by the evicted job must be recomputed at a later time. Soft memory eliminates the utilization-performance trade-off for the memory resource, opening the doors to maximizing memory utilization without risking process terminations. A soft memory allocator can allocate memory even when the memory on a machine is fully utilized, because it can revoke older soft memory allocations, transferring allocated memory between jobs without triggering evictions.

Shifting Resource Consumption Patterns. Even though compute clusters run many heterogeneous workloads, some universal resource consumption patterns exist in most large data centers. For example, low nocturnal user interaction with web services leads to reduced utilization of pre-assigned resources [1, 24]. In particular, spare CPU resources exist when the load on long-running services is low, even though their memory footprint remains the same. Soft memory helps applications and datacenter operators scale out during low-utilization periods. Extra workloads can reclaim the soft memory in under-utilized services and use it productively, which reduces CPU stranding. This suggests a two-level memory scheduling strategy: a cluster scheduler primarily decides a-priori on traditional resource memory allocations, while a lower-level soft memory scheduler redistributes revocable memory while jobs run. This increased fungibility allows reclaimed soft memory to be repurposed when needed, and we expect that jobs employing soft memory will benefit from higher likelihood of being scheduled.

Example Use-case: Key-Value Store. Consider a datacenter where a long-running web service uses Redis [16] as an in-memory cache to reduce tail-latency. During nocturnal lulls in traffic, the web service can operate on a much smaller cache footprint without harming tail latency. Redis can put the cache in soft memory, so that when batch jobs in the datacenter scale up at night, they can reclaim part of the cache memory. The cache can be scaled back up during the day when latency is critical and batch jobs have finished.

Example Use-case: Machine Learning Training Cache. The input data pipeline is a bottleneck in machine learning (ML) training jobs as accelerators process data faster than a dataset batch gets loaded into memory [10]. Storage caches for deep learning maintain a partial set of the training dataset in memory and provide significant speedups through informed replacement policies (i.e., guaranteeing the randomness and uniqueness properties of batches of training data) [11]. Increasing cache size via soft memory can provide performance gains while productively using otherwise idle memory. Once this memory is needed again, the soft memory subsystem re-configures the cache to its original size. This slows down the ML training, but makes memory available for other workloads like latency-critical service jobs.

3 Design

We now explore one possible design for soft memory. A custom Soft Memory Allocator (SMA) manages soft memory on an application level. Developers can opt-in to using soft memory by using API calls `soft_malloc` and `soft_free`. We imagine that, in practice, many applications will use soft memory through pre-provided Soft Data Structures (SDSs), which hide the details of soft memory behind a familiar data structure API. A Soft Memory Daemon (SMD) is a machine-wide memory manager for soft memory requests, and arbitrates soft memory requests across applications.

3.1 Soft Memory Allocator

The main contribution of the Soft Memory Allocator is its ability to reclaim memory upon a demand by the Soft Memory Daemon. Figure 1 illustrates soft memory reclamation. The Soft Memory Allocator provides each SDS with its own heap and set of memory pages. Each SDS has a *context* in charge of tracking the SDS's heap and a user-defined priority. The SMA manages a *global free pool* of free pages that it assigns to SDS heaps upon memory requests and replenishes when a SDS transfers pages back to the pool after freeing allocations. The SMA has a soft memory *budget* assigned by the SMD upon startup. When an application makes soft memory requests to the SMD, its budget maintained in the SMA increases, while fulfilling soft memory reclamation demands from the SMD decreases the budget. Soft memory reclamation within an application is two-tiered—the SMA chooses SDSs and the SDSs choose allocations to give up.

The goals of the SMA design are the flexibility, efficacy, and non-disruptiveness of memory reclamation.

Flexibility. Soft memory is opt-in; developers can choose to allocate soft memory *or* traditional memory depending on application semantics. A developer might use soft memory for caches, lookup tables, request queues, etc., as memory loss may impact performance but will not affect correctness in these use-cases. The developer would use traditional memory where reclamation is not tolerable, e.g., when storing crucial application state. The developer is most knowledgeable about the semantics of allocated soft memory blocks, so we allow them to communicate semantic information to the SMA in the form of a user-defined SDS priority. A SDS's priority influences the likelihood that it will be instructed to reclaim memory by the SMA during soft memory reclamation.

Efficacy. Memory must be returned to the operating system at page granularity, but applications think in terms of allocations; once all allocations in a page are freed, the page can be returned. Accordingly, the SMA faces a trade-off between space and the number of allocation frees required to free up entire pages for reclamation. A policy where allocations are freed arbitrarily from the heap until enough entire pages are free would result in large numbers of allocation frees to fulfill a reclamation quota. A policy where each allocation gets its own page permits straightforward reclamation (an entire page is freed by one allocation free) but wastes copious amounts of space if most allocations are small (as is commonly the case [13]). We manage memory on the level of data structures to balance this trade-off; a SDS receives pages from the SMA and manages its own memory within these pages. During soft memory reclamation, the SMA distributes a reclamation quota among its SDSs. A SDS frees allocations from its own heap until it has satisfied the SMA's reclamation demand. Localizing allocation frees within a SDS's pages increases the likelihood of producing the required number of entirely-free pages for reclamation from the least amount of data structures. The downside is possible heap fragmentation. Such fragmentation will be proportional to the number of soft data structures (each with their own heap); recent work on sharded data structures that give a separate heap to each shard (a quantity much larger than the number of data structures) suggests that this overhead is acceptable in practice [17].

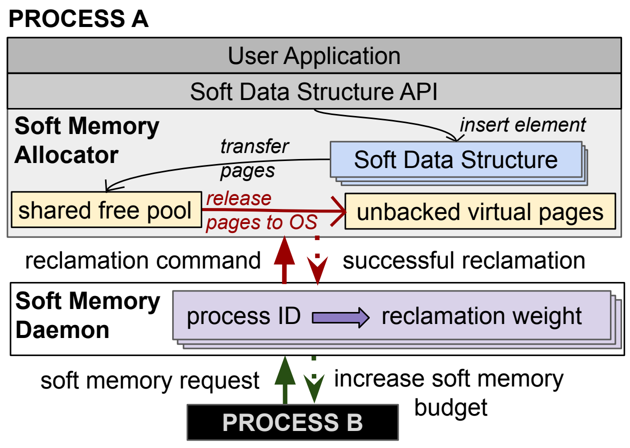

Figure 1: Soft Memory Reclamation: Process B requests soft memory from the SMD when the system is under memory pressure. The SMD selects Process A as a target for reclamation.

Non-Disruptiveness. Freeing a block of allocated memory is disruptive. Depending on the role of the memory, data may need to be re-fetched or recomputed down the line. We design our SMA's reclamation protocol to avoid freeing in-use blocks when possible and, if impossible to avoid, freeing low-priority blocks first. Consider an application with two soft linked lists that each have hundreds of 2 KB elements. Suppose the application receives a 12 KB reclamation demand (roughly three pages). If the application has excess soft budget or pages in its global free pool, it first exhausts these. If not, it begins with the lowest priority soft linked list and frees list elements from oldest to newest until the page quota is fulfilled. In our example application, two 2 KB list elements fit in a 4 KB page, so the quota is fulfilled by freeing the first six list elements. Before a list element is freed, the SMA invokes a developer-defined callback on the memory. This is a last-chance for the developer to interact with the memory before it is given up, e.g., to tag the data for future re-computation or store the data elsewhere.

3.2 Soft Data Structures

Soft data structures provide a familiar API for utilizing soft memory and handle details such as soft memory contexts and reclamation under the hood. SDSs are required to implement a `reclaim` method to handle reclamation demands from the SMA. Protocols for SDS reclamation are designed by data structure engineers. Our prototype, for example, provides implementations of a `SoftArray` and `SoftLinkedList` (see Listing 1). Our soft array gives up all of its soft memory upon

```
typedef void (*reclaim_callback_t)(void*);

class SoftLinkedList {
    SoftLinkedList(size_t priority,
                   reclaim_callback_t callback);
    size_t reclaim(size_t sz); // callback invoked here

    // standard linked list API...
}
```

Listing 1: Sketch of a `SoftLinkedList` SDS API.

a reclamation demand because an array is a single, contiguous memory block. Our soft linked list prioritizes newer entries over older entries when giving up list elements to fulfill a reclamation demand. A SDS engineer may choose a different policy, e.g., one that prioritizes infrequently-accessed elements for reclamation. As part of the SDS's reclaim logic, the SDS optionally invokes an application-provided callback to allow the application to react to reclamation.

3.3 Soft Memory Daemon

The Soft Memory Daemon manages soft memory resources across processes on a single machine. Soft memory differs from a world where applications individually free memory because the SMD helps the system escape memory pressure by maintaining global state and making informed reclamation decisions for in-use memory. The SMD tracks each process's soft memory budget and utilization. The SMD increases a process's soft memory budget by approving a soft memory request by that process and decreases a process's soft memory budget by issuing a reclamation demand to that process. Our SMD is designed to almost never deny a process's soft memory request, while not unfairly burdening other processes with reclamation demands.

When there is excess unassigned soft memory or excess soft memory budget in any process, the SMD can approve a soft memory request with minimal disturbance. When there is memory pressure and no excess soft pages exist, the SMD must demand the reclamation of allocated pages from SMAs on the machine. In this scenario, the SMD selects a capped number of processes in decreasing order of *reclamation weight* until it fulfills its reclaim page quota or hits the cap on the number of processes to consider. If the SMD does not reach the page quota, it denies the soft memory request that triggered the reclamation. This limits the number of applications that can be disturbed by a soft memory request.

The metric for calculating a process's reclamation weight should be designed to incentivize soft memory use. In particular, (i) the larger the (soft and traditional) memory footprint of the process, the higher its reclamation weight should be; and (ii) soft memory usage should increase the reclamation weight proportional to the traditional memory usage of a process. The latter criterion is important, as it ensures that processes with high soft-to-traditional memory ratio avoid getting disturbed disproportionally often, which would be a disincentive for soft memory use. The higher the reclamation weight of a process, the more likely it becomes that the SMD picks that process as a reclamation target. For example, suppose applications A and B each use the same number of soft memory pages and T_A and T_B pages of traditional memory, respectively. If $T_A < T_B$ then A has a lower reclamation weight than B because its soft memory footprint is the same as B's but the proportion of soft to traditional memory is greater in A than B. Application A chose to put more of its data into soft memory, which increases flexibility of the overall system, so it has a lower chance of receiving a reclamation demand than application B, which tied up more memory.

Soft memory is a *reactive* abstraction because the SMD coordinates the reallocation of in-use memory once under memory pressure. The SMD would ideally distribute reclamation requests across processes so that the re-computation cost of such entries is less than the cost of killing a process and restarting it, which incurs the cost of recomputing its entire state in traditional and soft memory. The SMD does not manage traditional memory and leaves satisfying the traditional memory amount required to achieve application goals (i.e., performance, liveness, etc.) to the data center scheduler.

4 Prototype

We implemented a prototype memory allocator and daemon in 3,031 lines of C++. Each SDS has an isolated heap and periodically transfers free pages back to the global free pool of transferable, on-demand soft memory. When the memory allocator releases pages back to the operating system upon a reclamation demand, it tracks the released virtual pages to re-back them with physical pages before extending the heap.

When a soft memory request requires reclamation, the SMD considers different processes as possible reclamation targets. The daemon has an upper bound on the number of reclamation targets and selects them in descending reclamation weight, but biases towards targets that will experience little or no disturbance from the reclamation. Suppose a reclamation target has allocated all its memory to SDSs. In that case, the daemon will try other reclamation targets to find a process in a more flexible memory state (i.e., with an unused memory budget); only when the SMD cannot find a better option, it will return to the first target and trigger reclamation. The SMD demands a fixed memory percentage upon reclamation, which may exceed the immediate soft memory request, in order to amortize reclamation costs.

5 Preliminary Results

We use our prototype to investigate the practicality and performance of soft memory. To do so, we added soft memory support to the popular Redis key-value store [16]. Redis is a

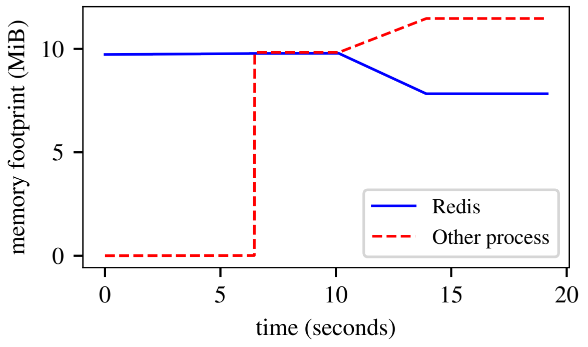

Figure 2: Under memory pressure, reclaiming soft memory from the Redis key-value store reduces its memory footprint and moves memory to another process without crashing either application.

single-threaded application, and stores data in an in-memory hash table. We modified this hash table to store the elements of its buckets in soft memory, turning it into an SDS. We added 25 lines of code to Redis (out of 258K LoC total).

We investigate Redis's response to memory pressure and soft memory reclamation. We set up the Redis server with 130K key-value pairs all allocated in soft memory (10 MiB total). We then simulate another process that allocates 12 MiB of soft memory, which exceeds the 20 MiB available soft memory on the machine, requiring the SMD to reclaim soft memory from Redis. Figure 2 shows a timeline of events: at $t = 10.13$s, the test process makes a request that exceeds its budget. The SMD detects memory pressure and initiates a reclamation. At $t = 13.88$s, the reclamation finishes and Redis has relinquished 2 MiB of soft memory.

We find that the reclamation time of 3.75s is spent almost exclusively in Redis code, invoked via the callback, that cleans up associated traditional memory for the reclaimed entries. Requests for the key-value pairs removed from Redis will now return "not found" to the client; in a caching setup, the client would re-fetch these entries from a database if it needed them. Without soft memory, Redis would crash under memory pressure. The cost of such a termination is a minimum of 12ms of downtime for Redis to restart, with an additional, load-dependent period of increased tail latency while the cache refills.

To understand our prototype's performance in more detail, we stress-test the SMA and SMD in three settings with 1 KiB allocation size: (1) one process makes 977K soft memory allocations with sufficient budget from the SMD; (2) one process makes the same number of soft memory allocations, but the SMA grows its soft memory budget by communicating with the SMD; and (3) two processes each make 977K soft memory allocations, then one process makes another 500k allocations that require reclaiming and moving soft memory from the other process.

We measure the total time to make all allocations (cases (1) and (2)) and the time taken to make the additional 500k allocations (case (3)) under memory pressure. We also measure, for each test case, the time it takes to create the same number and size of allocations using the system allocator, and compare our SMA's performance to this baseline. A good result would show competitive performance when the SMA allocates available soft memory, and low overhead when it reclaims soft memory from another process. Making 977K allocations with the SMA (case (1)) takes 1.22× as long as with the system allocator. The communication needed for managing the soft memory budget has negligible effect on performance (case (2)): the SMA still takes 1.23× as long as the baseline, as communication with the memory daemon to increase resource budget is amortized over many allocations. Finally, reclamation—which requires extra work to redistribute memory among processes—is still fast, making the 500k allocations under memory pressure take 1.44× as long as making 500k allocations without memory pressure.

It is worth noting that our current prototype SMA is a simple textbook memory allocator without optimizations; adding soft memory functionality to a state-of-the-art allocator such as jemalloc [6] or TCMalloc [9] would likely further improve performance.

6 Related Work

Soft memory is closely related to work on far memory [8, 12, 18, 25], and our approach applies many techniques inspired by recent far memory systems. AIFM [18] supports far data structures that swap part of their memory into far memory, with APIs similar to our soft data structures. The main difference between AIFM and soft memory is that AIFM is a swapping mechanism that stores data remotely to be swapped back in the future, while soft memory deletes reclaimed memory content after invoking a callback. Soft memory works well for caching use cases, whereas AIFM targets applications that prioritize returning the data to the program.

zswap [12] proactively compresses cold memory pages and maintains their compressed version in DRAM. This transfer from near to far memory is transparent to the application. By contrast, soft memory is explicit about memory reclamation via its callback mechanism and SDSs reactively reclaim pages under memory pressure to avoid process disruption. Soft memory also allows developers to opt into using it selectively, while zswap is an OS-level solution and may swap out *any* cold memory pages.

VM ballooning [20] involves redistributing free memory resources among multiple VMs. This operation is comparable to process-level soft memory reclamation of unused memory budget, which precedes the reclamation of in-use data structure memory. However, VM ballooning cannot reclaim *in-use* memory. Memory-harvesting VMs (MHVMs) [7] and Deflatable VMs [21] are a new type of elastic VM that can have its memory size reduced under pressure. But MHVMs reclaim memory at the granularity of entire containers running inside the VM, and terminate containers if insufficient reclaimable memory exists, which replicates precisely the job eviction situation that soft memory tries to avoid. Deflatable VMs trigger existing application-level mechanisms (JVM GC, memcached cache eviction) when deflating memory, rather than co-designing the memory allocator with the deflation mechanism (as in soft memory).

Under memory pressure, managed language garbage collectors free inaccessible objects to make space for new allocations [5]. By contrast, the SMA may revoke live allocations when free memory is unavailable. Prioritized garbage collection realizes space-aware caches ("Saches") via a soft-reference-style API that allows the garbage collector to eagerly reclaim objects in caches [15]. This realizes one key use case for soft memory, but realizes it in the context on a managed language with GC.

7 Open Questions

Integrating soft memory across the systems stack raises interesting questions for research that touches application APIs, language runtimes, OS abstractions, and hardware features.

Handling Reclamation. When a soft allocation gets reclaimed, all pointers into it become invalid. Our prototype handles reclamation by invoking an application-specific callback, which gives the application a chance to remove such pointers (e.g., by rewriting them to `NULL`). But finding all pointers into an allocation is difficult in an unmanaged language like C/C++, as the pointers could be anywhere and there could be any number of them. Even though approaches to locating all pointers, e.g., by scanning the whole heap [2] exist, they are prohibitively expensive, particularly at the timescales on which soft memory reclamation needs to happen. This could be solved by requiring pointers into soft memory to be created via a runtime that keeps track of these pointers, an approach that requires code changes, but may work within SDS implementations.

Concurrency. With concurrency, the SMA's reclamation of a soft allocation can race with another thread that is accessing the memory. We expect that ideas from far memory systems, such as the thread-safe smart far pointers in AIFM [18], could help realize safe soft memory abstractions under concurrency with reasonable cost. AIFM's smart pointers impose a cost of only five x86-64 instructions per pointer dereference on the fast path, but require developers to wrap their accesses to the data pointed to into dereference scopes, custom syntactic constructs that notify a runtime that a thread is currently accessing an allocation. Like with AIFM's far-memory data structures, we imagine that much of this complexity can be hidden within the soft datastructure implementation, and that application developers need not worry about it for the most part.

Language Integration. Managed language runtimes could leverage soft memory in effective ways, since they often already have ways of locating all pointers to an object for garbage collection (GC). We believe that co-designing a language runtime with a soft memory system could be a fruitful direction for future research; indeed, soft-memory-like abstractions already exist in some managed languages, e.g., in the form of Java's `WeakReference`. Managed language runtimes with GC also already have means of handling race conditions between the runtime and application threads, such as read/write barriers.

Policies for Soft Memory. Our design has the SMD arbitrate between different applications' soft memory needs by managing their budgets. Many different policies are imaginable, and the question of what soft memory reclamation strategy is fair or desirable is itself challenging. Should processes that have a larger soft memory footprint, and thus benefit the most from soft memory, be called upon to give up more soft memory when memory is tight? While doing so intuitively seems fair, it also provides a disincentive from using soft memory. Similarly, there are open questions about whether a good policy should take into account a process's total memory footprint, and whether the SMD should let a process reclaim its own (older) soft memory. We expect the answers to these questions to be determined empirically for different use cases and applications of soft memory.

Soft Data Structures. Nailing down the right APIs for soft data structures is an important challenge, particularly if SDSs are used in composition. For example, in our prototype Redis integration, we changed the hashtable's per-bucket soft linked lists to store their list elements in soft memory. These elements then themselves point to dynamically-allocated heap memory for storing the key and value. If these allocations were also soft, reclamation might reclaim only a key or value, or both key and value, but leave the (now-incomplete) list element in place. In our prototype, this worked out because we left the keys and values in traditional memory and de-allocate them via the reclamation callback function. Better APIs for composition, for grouping soft allocations, and for prioritizing soft allocations would be desirable.

Acknowledgements

We thank the anonymous reviews and the ETOS group at Brown for their helpful feedback on drafts of this paper. This work was supported by NSF award DGE-2039354, by a Google Research Scholar Award, and by a gift from VMware.

References

[1] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. "On the Diversity of Cluster Workloads and Its Impact on Research Results". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. Boston, Massachusetts, USA, 2018, pages 533–546.

[2] Hans-Juergen Boehm. "Space Efficient Conservative Garbage Collection". In: *SIGPLAN Notices* 28.6 (June 1993), pages 197–206.

[3] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 2017, pages 153–167.

[4] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *SIGPLAN Notices* 49.4 (Feb. 2014), pages 127–144.

[5] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. "Garbage-First Garbage Collection". In: *Proceedings of the 4th International Symposium on Memory Management*. Vancouver, BC, Canada, 2004, pages 37–48.

[6] Jason Evans. "A Scalable Concurrent malloc(3) Implementation for FreeBSD". In: *Proceedings of the BSDCan Conference*. Ottawa, Canada, 2006.

[7] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. "Memory-Harvesting VMs in Cloud Platforms". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland, 2022, pages 583–594.

[8] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. "Efficient Memory Disaggregation with INFINISWAP". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, 2017, pages 649–667.

[9] Google Inc. *gperftools: Fast, multi-threaded malloc() and nifty performance analysis tools*. URL: http://code.google.com/p/gperftools/.

[10] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. "Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines". In: *Proceedings of the 4th Conference on Machine Learning and Systems (MLSys)*. Volume 4. 2022, pages 33–51.

[11] Abhishek Vijaya Kumar and Muthian Sivathanu. "Quiver: An Informed Storage Cache for Deep Learning". In: *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA, Feb. 2020, pages 283–296.

[12] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. "Software-Defined Far Memory in Warehouse-Scale Computers". In: Providence, Rhode Island, USA, 2019, pages 317–330.

[13] Per-Åke Larson and Murali Krishnan. "Memory Allocation for Long-Running Server Applications". In: *Proceedings of the 1st International Symposium on Memory Management (ISMM)*. Vancouver, British Columbia, Canada, 1998, pages 176–185.

[14] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. "Imbalance in the cloud: An analysis on Alibaba cluster trace". In: *Proceedings of the 2017 IEEE International Conference on Big Data (Big Data)*. 2017, pages 2884–2892.

[15] Diogenes Nunez, Samuel Z. Guyer, and Emery D. Berger. "Prioritized Garbage Collection: Explicit GC Support for Software Caches". In: *SIGPLAN Notices* 51.10 (Oct. 2016), pages 695–710.

[16] Redis Ltd. *Redis*. URL: https://redis.io/ (visited on 05/23/2023).

[17] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. "Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes". In: *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Apr. 2023, pages 1409–1427.

[18] Zhenyuan Ruan, Malte Schwarzkopf, Marcos Aguilera, and Adam Belay. "AIFM: High-Performance, Application-Integrated Far Memory". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Banff, Canada, Nov. 2020, pages 315–332.

[19] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. "Autopilot: Workload Autoscaling at Google". In: *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*. Heraklion, Greece, Apr. 2020.

[20] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. "Application Level Ballooning for Efficient Server Consolidation". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pages 337–350.

[21] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. "Resource Deflation: A New Approach For Transient Resource Reclamation". In: *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*. Dresden, Germany, 2019.

[22] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: the Next Generation". In: *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*. Heraklion, Crete, Apr. 2020.

[23] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg". In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[24] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. "Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly". In: *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*. San Francisco, California, 2022, pages 210–225.

[25] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. "Carbink: Fault-Tolerant Far Memory". In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, July 2022, pages 55–71.