Harvesting Idle Memory for Application-Managed Soft State with Midas

Yifan Qiao Zhenyuan Ruan[‡] Haoran Ma Adam Belay[‡] Miryung Kim Harry Xu

UCLA [‡]*MIT CSAIL*

Abstract

Many applications can benefit from data that increases performance but is not required for correctness (commonly referred to as *soft state*). Examples include cached data from backend web servers and memoized computations in data analytics systems. Today's systems generally statically limit the amount of memory they use for storing soft state in order to prevent unbounded growth that could exhaust the server's memory. Static provisioning, however, makes it difficult to respond to shifts in application demand for soft state and can leave significant amounts of memory idle. Existing OS kernels can only spend idle memory on caching disk blocks—which may not have the most utility—because they do not provide the right abstractions to safely allow applications to store their own soft state.

To effectively manage and dynamically scale soft state, we propose *soft memory*, an elastic virtual memory abstraction with *unmap-and-reconstruct* semantics that makes it possible for applications to use idle memory to store whatever soft state they choose while guaranteeing both safety and efficiency. We present Midas, a soft memory management system that contains (1) a runtime that is linked to each application to manage soft memory objects and (2) OS kernel support that coordinates soft memory allocation between applications to maximize their performance. Our experiments with four real-world applications show that Midas can efficiently and safely harvest idle memory to store applications' soft state, delivering near-optimal application performance and responding to extreme memory pressure without running out of memory.

1 Introduction

A wide range of applications can benefit from storing *soft state* in memory, including web applications [43], databases [32], key-value stores [30], CDN services [12, 34], and model serving frameworks [9]. Data is considered soft state when it is helpful for efficiency, but discarding it does not impact correctness because it can easily be reconstructed if it is later needed. For example, caches and memoization are both common forms of soft state. Soft state enables applications to trade extra memory consumption for better performance, and these gains generally increase with the amount of memory available [45, 47]. A significant fraction of memory is left idle in today's datacenters [27, 48], suggesting there is a large untapped opportunity to improve overall efficiency by using idle memory to store soft state.

While spending memory on soft state can improve performance, it must not compete with the need to store regular application data. For example, if too much memory is spent on soft state, this could lead to swapping to disk or worse still, out-of-memory errors, which can result in failures. Because of this, developers often limit their storage of soft state to a small static amount, for fear that they may run out of memory. In other words, it is a challenge to allocate enough soft state to consume all available idle memory, but to not go beyond the point where it would cause performance issues or failures.

Existing OS abstractions for elastically responding to changes in available idle memory are too limited. For example, the Linux Kernel maintains a *page cache* that automatically fills idle memory but it can only be used to cache disk blocks. This constrains idle memory to storing just a single type of soft state which may or may not provide the most utility for applications.

An ideal abstraction would instead democratize access to idle memory so that each application could choose how to best spend it (*i.e.*, the type of soft state that is most beneficial). For example, suppose an application does not rely much on local storage, but frequently accesses objects stored in a key value store over the network. Instead of being limited to the page cache, idle memory could be spent on caching the key-value store's objects locally, resulting in a much greater benefit.

This problem is further complicated in today's multi-tenant cloud. It is common for each server to run multiple applications, and they may come from different users and exhibit dramatically different performance sensitivity to the amount of soft state. At the same time, adding memory to one application can lead to reductions in the performance of others. Consequently, determining how to dynamically balance the soft state needs of different applications in a way that maximizes overall memory utility/performance is a challenge.

Insight. In this paper, we aim to answer the following question: can we provide a new *virtual memory abstraction* for soft state (herein referred to as "soft memory") that developers can use to coordinate with the kernel so that they can take full advantage of all available memory? In other words, our goal is to no longer limit idle memory to the page cache, and to instead allow its use to be customized by each application in a way that maximizes overall utility.

Unlike existing systems that perform caching entirely in the user space [2, 9, 30], we propose *Midas*, a system that coordinates with the kernel to dynamically provision soft memory between applications. The advantage of this approach is two-fold. First, application developers can program with the illusion of an "unlimited cache", and are thus freed of the burden of manually managing their soft state. To avoid running out of memory, the kernel responds to memory pressure by rapidly *unmapping* soft memory pages. To transparently recover any lost soft state, later accesses will automatically trigger the application to *reconstruct* the missing soft state. Second, the kernel has global visibility of all applications, their memory usage, and the amount of idle memory, making it possible to understand each application's sensitivity to memory size and automatically coordinate soft memory allocation between applications. Midas also incorporates the page cache by treating it as another source of soft memory.

Challenges. Midas is a soft memory management system that achieves (1) programming flexibility and (2) dynamic memory provisioning, with *unmap-and-reconstruct* semantics, to guarantee both safety and efficiency. Realizing these benefits requires overcoming four major challenges:

First, what interfaces shall we expose to developers? To improve usability, Midas provides developers with a *soft memory pointer* abstraction (similar to C++ smart pointers) to access soft memory easily and safely (see §4.1). Midas offers a set of high-level key-value store APIs, which are similar to those of popular cache services (such as Memcached [30], Redis [2], CacheLib [9], *etc.*), but enhanced to allow the exposure of more semantics to the runtime. A critical interface we expose to developers is *data structure reconstruction*—developers not only register soft memory objects but also specify their (re)construction logic, so soft state can be transparently regenerated if it is later accessed after it was evicted.

Second, how shall soft memory be managed? Program data is allocated as objects on the heap but the kernel cannot recognize them, as it is only aware of memory pages. As a result, if we let the kernel manage soft memory alone, it could only reclaim space in coarse-grained units without knowledge of what objects the space contains. For example, reclaiming hot (*i.e.*, frequently accessed) objects in a soft-state cache can lead to significant slowdowns. In addition, it is undesirable to reclaim space from the programs that would benefit the most from soft state when others need it less, but such performance sensitivity information is invisible to the kernel.

To solve the problem, we propose a runtime library that can be linked into each application to recognize object behaviors, letting the runtime and the kernel *co-manage* soft memory. The Midas runtime offers a log-structured allocator [41] and a concurrent evacuator that continuously identifies and compacts hot objects into a small soft memory space. This information (of hot and cold regions) is shared with the kernel so that it can focus its reclamation on regions with cold objects (see §4.2).

Third, how can we coordinate soft memory allocation between applications? The runtime can only see each application's individual behavior without any global knowledge of the server's available memory and other applications' needs. Furthermore, the runtime can only manage objects in user space, but *cannot* dynamically add/remove memory between applications. To overcome this challenge, we propose a global coordinator inside the Linux kernel. The coordinator periodically probes each application by communicating with the runtime to request information regarding the application's sensitivity to cache size. Cold regions of soft memory from applications that are less sensitive to size changes will be reclaimed and memory will be given to those that are more sensitive (and hence benefit more from a larger cache) by the kernel (see §4.3).

Finally, how can the kernel quickly reclaim soft memory without disrupting a running application? Since the kernel operates at page granularity, a natural idea is to swap out pages that contain soft-state data. Unfortunately, swapping is disruptive—swapping out a page blocks all incoming memory allocations and hence all threads of the application; frequently swapping pages can introduce significant overheads that prevent applications from reaching service-level agreements (SLA) [40] (see §2).

To maintain high efficiency, Midas instead uses the kernel to unmap pages directly (which is much faster than swapping them to disk). When pages are unmapped, their underlying data is lost—this is acceptable for soft state because it can be regenerated. Without coordination, however, the kernel cannot distinguish soft state from application data, making unmapping potentially unsafe.

To solve this problem, our runtime is designed in a way that is resilient to data loss. A soft pointer-based interface detects data loss through segmentation faults that are triggered by the runtime's functions. These functions are carefully designed to capture faults and transparently invoke a *reconstruction* interface to regenerate the needed data (see §4.2.3).

Compared to paging, Midas does not freeze the execution when shrinking soft memory, resulting in less disruption to the application. Furthermore, reconstruction focuses on recovering the individual objects that are needed and hence is much more fine-grained and can be more efficient than swapping, which brings back entire pages. Reconstruction may require more computation than paging (the amount of computation depends on exact soft state data). Therefore, Midas provides a profiling tool that warns developers when reconstruction incurs a high cost (discussed in §4.4).

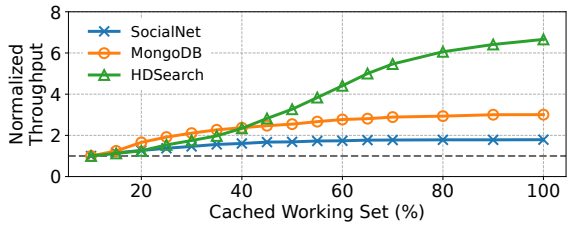


Figure 1: The throughput of all three applications increases by caching more soft state, but the benefit varies: SocialNet is 1.8× faster by caching 70% of its working set, while HDSearch, in contrast, achieves a 3.3× throughput increase by caching 50% state.

Results. With Midas, one can easily allow applications to take advantage of soft state that do not currently support it. It is also easy to port legacy code that uses an existing cache system to use Midas instead. Our evaluation shows that Midas can efficiently and safely harvest idle memory to store applications' soft state and achieve near-optimal performance while reacting to extreme memory pressure quickly enough to avoid running out of memory. By effectively granting soft memory to the applications that benefit the most, Midas achieves 1.34× higher overall throughput than Cliffhanger (a state-of-the-art caching system). Midas is available at https://github.com/uclasystem/midas.

2 Motivation

Many types of applications can benefit from soft state. For example, a web frontend could cache content locally after loading it from a backend to reduce network traffic and improve response times; a database could cache the results of user queries to reduce disk I/O and improve throughput; and a data analytic or machine learning system could memoize intermediate computation results to eliminate redundant computations.

To gain a high-level understanding of how much improvement can be achieved by storing soft state, we experimented with three datacenter applications: SocialNet (from DeathStarBench [18]), MongoDB [32], and HDSearch (from µSuite [46]). Each of these applications are capable of using soft state. SocialNet [18] is a web forum built using microservices; it employs Memcached and Redis to cache user data in its frontend services. MongoDB [32] is a NoSQL database; it has a built-in, in-memory caching engine that caches recently queried data. HDSearch is an image search service that memoizes the feature vectors of the images in its corpus, generated by a GPU-based DNN.

Figure 1 shows the throughput of each application with varying amounts of soft state. The *x*-axis represents the percentage of each application's working set cached in memory, and the *y*-axis shows the normalized throughput (to its performance without soft state). Soft state is helpful to all applications but the amount of benefit it provides varies. SocialNet is the least sensitive to its soft state size; however, it still sees a 1.8× speedup by storing 70% of its soft state. HDSearch, in contrast, is more sensitive to the soft state size—its throughput increases by more than 3× with only 50% of its soft state.

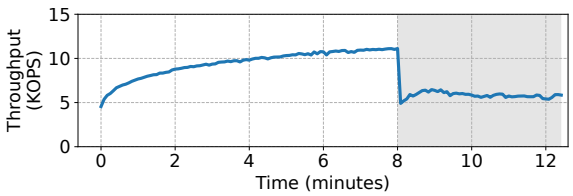


Figure 2: SocialNet starts to swap when it caches excessive data and exhausts all available memory at $t = 8$min and it experiences a throughput collapse.

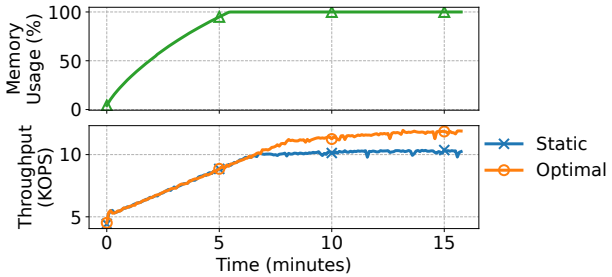


Figure 3: Statically provisioning the cache space for SocialNet is sub-optimal. During t =0min–5min, the cache is overprovisioned which wastes memory. After that, the cache becomes underprovisioned which limits performance.

Real-world datacenter applications can access a massive amount of data. For example, a web forum like Twitter generates petabytes of new data every day [49]. Thus, blindly storing soft state in memory without a proper limit can hurt application performance. An example of this problem is shown in Figure 2. Storing soft state increases the throughput of SocialNet up to a point. However, when idle memory becomes exhausted, the kernel begins to swap out pages (at $t = 8$min), leading to a severe collapse in throughput.

A simple strawman solution is to statically provision a limited memory capacity for storing soft state so that memory use does not grow unbounded. However, provisioning the right capacity is extremely challenging in practice.

First, for each application, we must find its sweet spot of cache capacity; underprovisioning limits performance while overprovisioning wastes memory. In addition, datacenter applications often have phased behaviors and load variability [7, 8], making it impossible to have a simple static configuration that is optimal at all times. For example, Figure 3 shows the results of SocialNet when statically provisioning it with 4 GiB for storing soft state. It takes about 5 minutes to fully fill this memory, leading to a suboptimal utilization during this period. Performance increases with more usage until it exhausts the soft state limit. After that, performance flattens out despite the possibility of higher throughput if additional soft state memory were available (the optimal line).

Second, as shown by Figure 1, different applications gain different amounts of benefits through caching. To achieve optimal overall performance with a limited amount of memory,

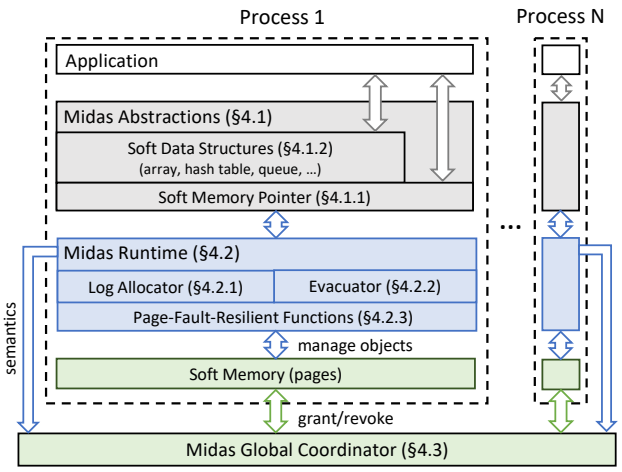


Figure 4: Midas enables developers to utilize soft memory easily and efficiently with three major components: a familiar programming abstraction, an application-integrated runtime, and a global soft-memory budget coordinator.

one must grant space correctly to the applications that benefit the most. For example, initially MongoDB's performance is most sensitive to the amount of soft state (the left side of Figure 1), and thus we should prioritize its need. However, the return diminishes quickly after caching 30% of its state. To make the best use of the remaining memory, we should respond by granting memory to HDSearch.

These problems call for a new system that can provide *elastic* access to soft state for applications and *dynamically coordinate* usage among applications in response to each one's execution phase and sensitivity to soft state size. To be efficient, soft state should be able to quickly scale up and down its capacity with little disruption. To be safe, the system should be resilient to data loss caused by scaling down. To be responsive, the system should conduct coordination among applications quickly. Finally, to be practical, the system should provide familiar programming abstractions for developers to store and access soft state.

3 Midas Overview

As shown by Figure 4, Midas consists of three main components: a programming abstraction for using soft memory (§4.1), an application-integrated runtime that manages soft-memory objects (§4.2), and a global coordinator that arbitrates soft memory usage across different applications (§4.3).

Midas provides programming abstractions that enable simple and efficient use of soft memory through familiar APIs. At a low-level, programmers can interact with Midas through *soft memory pointers*, an abstraction that provides object ownership similar to C++ smart pointers. However, it differs in that underlying objects can be forcibly released when under memory pressure, even if still in scope. If a released object is later accessed, a *reconstructor* function is invoked to regenerate the missing object (*e.g.*, by fetching it from a database over the network).

Building upon soft memory pointers, Midas provides a higher-level library of familiar STL-style *soft data structures*—including arrays, hash tables, and queues. These hide the complexity of managing individual soft memory pointers, and can be used as drop-in replacements for existing data structures. For example, a developer building a key-value store similar to Memcached could use a soft hash table to store soft memory objects. Midas's high-level interface is generally sufficient for most use cases, but developers are free to build their own custom soft data structures through use of soft memory pointers.

Midas manages soft memory objects through a runtime that is linked as a library with the application. It serves as an allocator for soft memory objects. It works cooperatively with the coordinator (discussed next) to determine the best memory to release (*i.e.*, idle memory first, then cold objects, and finally hot objects). To achieve this, the runtime provides a moving allocator that embraces the idea of log-structured memory [41] to organize soft memory into different segments. An evacuator thread scans and compacts logs to segregate hot objects, cold objects, and dead objects. This helps both to coordinate which memory should be freed and to reduce fragmentation.

However, the runtime is not trusted for correct operation. If it fails to respond quickly enough or if memory pressure becomes too severe, pages will be unmapped in an uncoordinated fashion to avoid swapping. In the event such forcible revocation happens, the runtime is designed to safely tolerate page faults when accessing unmapped memory. To achieve this, we developed a set of page-fault-resilient functions and used them as primitives to build our runtime.

Midas's global coordinator dynamically adjusts the soft memory budget among applications to optimize their overall performance. It periodically probes the marginal utility of soft memory for each application by granting a small amount of additional memory and observing the effect on performance. Using this information, the coordinator can optimize the allocation of soft memory by granting it to the applications that benefit the most. The coordinator defines the global utility function as the weighted average of all applications' performance and employs a hill-climbing algorithm to approach the global optimal point. Midas allows operators to specify the weight of each application to indicate relative significance, similar to the `nice` interface of Linux.

4 Design

4.1 Soft Memory Abstraction

Soft memory is a new type of memory that can be revoked under memory pressure. In Midas, soft memory is backed by physical pages that can be unilaterally unmapped and reclaimed by the OS kernel. Accessing reclaimed soft memory will trigger a *reconstruction event* to rebuild the missing data. Midas provides a smart-pointer-like API to enable developers to easily use soft memory, hiding the complex details of soft memory allocation/deallocation, page-fault handling, and data

```
template <typename T, typename... ReconArgs>
class SoftMemPool {
  SoftMemPool(std::function<T(ReconArgs...)>
    reconstructor);
  SoftUniquePtr<T, ReconArgs...> new_unique();
  SoftSharedPtr<T, ReconArgs...> new_shared();
};

template <typename T, typename... ReconArgs>
class SoftUniquePtr {
  ~SoftUniquePtr();
  T read(ReconArgs... args);
  void write(T newval);
  bool cmpxchg(const T &oldval, T newval);
};
```

Listing 1: Midas's soft memory pool and unique pointer interface.

```
1  template <typename T> class SoftArray {
2    SoftArray(size_t size, std::function<T(size_t)>
3      reconstructor);
4    T read(size_t idx);
5    void write(size_t idx, T t);
6    bool cmpxchg(size_t idx, const T &oldval, T newval);
7  };
8
9  class BlockCache {
10   BlockCache(size_t sz) : array_(sz, [](size_t idx) {
11     return read_from_storage(idx); }) {}
12   Block read(size_t idx) { return array_.read(idx); }
13   void write(size_t idx, Block block) {
14     array_.write(idx, block);
15     write_to_storage(idx, block);
16   }
17
18   SoftArray<Block> array_;
19 };
```

Listing 2: Midas's soft array interface and a simple user-level storage block cache (similar to Linux's page cache) built using soft array.

reconstruction (§4.1.1). Furthermore, Midas offers high-level data structure libraries as composable building blocks (§4.1.2).

4.1.1 Soft Memory Pointer

Listing 1 shows Midas's soft memory pool and pointer interface. To use soft memory, developers first need to create a soft memory pool which can later be used to allocate soft memory pointers. The pool abstraction conceptually groups together soft pointers whose objects can be reconstructed in a similar way. Midas exposes the pool as a C++ template class whose parameters consist of two parts: `T`, which is the object type of soft pointers to allocate, and `ReconArgs`, which are the types of arguments used for reconstructing a missing object. Developers can initialize a pool with a `reconstructor` function and then allocate pointers using `new_unique` (for soft unique pointers, similar to C++'s `std::unique_ptr`) and `new_shared` (for shared pointers).

Soft memory pointers support automatic lifetime management through reference counting. Developers can use its `read` API to get the value of the pointed object. In case the underlying soft memory has been reclaimed, Midas will automatically reconstruct the missing object using the reconstruction arguments passed into `read` (we will show a concrete example soon in §4.1.2). Midas hides the raw reference and returns the value by *copying*. This is critical as the underlying reference may become invalid any time when the soft memory gets reclaimed. With copying, Midas restricts potential faulting sites to stay inside Midas's internal code, thereby freeing developers from handling complicated page faults in the application code. The copying design incurs negligible performance overheads (only a few additional cache accesses). `write` enables developers to update the object value. However, different from `read`, `write` does not require reconstruction arguments as Midas can directly rebuild the object using the new value. Soft pointers also support atomic operations like compare-and-exchange, enabling developers to atomically update object values to support multi-threaded applications.

With its smart pointer design, Midas is able to capture rich application semantics for effectively managing soft memory. For example, since all soft object accesses go through the `read/write` API, Midas can accurately track the hotness information of each object which can be leveraged by Midas runtime for making intelligent object placement and eviction decisions (details in §4.2). Soft pointer's automatic lifetime management enables cascading eviction, improving the efficiency of using soft memory. For instance, in a web forum application, a forum post object may contain a soft unique pointer to an attached picture. Under memory pressure, Midas may decide to evict the post object in which case the reference count of the picture pointer will automatically drop to zero and trigger evicting the dangling picture object cascadingly.

4.1.2 Soft Data Structures

To further reduce the programming effort of using soft memory, Midas offers high-level data structures as convenient building blocks. Midas's built-in data structures include soft arrays, soft hash tables, and soft queues; developers can also easily build more based on the soft pointer abstraction.

Listing 2 presents the interface of soft array (lines 1-8). Developers can create a soft array by specifying its size and reconstructor (which rebuilds the array element of a given index). Soft array supports standard read, write, and atomic operations by index. Under the hood, a soft array is simply implemented via an ordinary array of soft pointers.

Lines 10-21 present a user-level storage block cache as a simple illustrative application, similar to Linux's page cache. `BlockCache` internally wraps a soft array whose elements are storage blocks (line 20). This enables it to efficiently leverage idle memory to cache storage blocks in a best-effort manner. For each block read request, it simply retrieves the result from the soft array (line 14). Upon an element miss, the array automatically reconstructs the element by reading the block back from the storage device (lines 12-13). For each block write request, `BlockCache` updates both the cache in array and the data in storage.

4.2 Application-Integrated Runtime

Midas runtime is the key component that manages soft objects to enable efficient use of soft memory. It includes a log-structured memory allocator that serves memory allocation

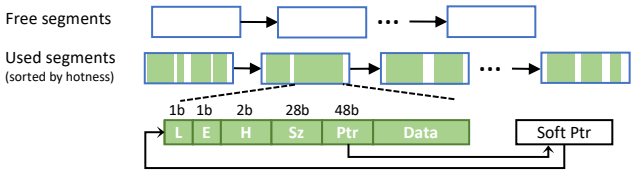


Figure 5: Midas organizes soft memory using a free segment list and a used segment list (sorted by segment's hotness, useful for Midas's evacuator in §4.2.2). It employs a log-structured allocator to serve memory allocation requests. Each object has a 10-byte header, which includes a liveness bit, an evacuating bit, hotness bits, an object size field, and a reversed pointer field.

requests and organizes objects into a list of segments (§4.2.1), a concurrent evacuator that constantly compacts hot objects and releases cold and dead objects (§4.2.2). Page faults can happen in Midas runtime when the soft memory it is accessing gets reclaimed and unmapped because of memory pressure. To ensure robustness, we carefully built the runtime using a set of page-fault-resilient functions which are able to capture page faults and gracefully recover from them (§4.2.3).

In Midas, the runtime as well as the soft memory it manages are linked directly into each application's address space. Compared to traditional cache services (*e.g.*, Memcached) that run in a separate process, our design offers several important advantages. First, it provides direct and efficient soft memory accesses for applications, eliminating the inter-process communication (IPC) overhead. Second, it enables our runtime to profile the application and collect semantics, greatly facilitating semantics-aware optimizations. Third, since each application has its own runtime, we can easily enforce soft memory isolation among applications and adaptively customize the memory management policy of each application.

4.2.1 Log-structured Soft Memory Allocator

Midas embraces the idea of log-structured memory [41] to manage soft memory; it reduces memory fragmentation through compaction, thereby achieving higher efficiency in utilizing soft memory.

Midas's log-structured allocator organizes soft memory using a free segment list and a used segment list, illustrated in Figure 5. Segments are the units for Midas to perform evacuation to compact objects and reclaim space (details in §4.2.2). The total size of all segments (used and free) equals the soft memory budget that the linked application receives from the global coordinator (§4.3). For each memory allocation request, the allocator allocates space from a free segment; if the current one is full, it will pop a new one from the free list. Midas backs each segment using a 2 MiB huge page; this reduces TLB pressure and page table walk cost. While small objects reside in only one segment (i.e., they do not cross the segment boundary), big objects whose sizes are larger than 2 MiB span across multiple segments. Since the free list does not provide any address contiguity guarantee for segments, Midas breaks the big object into smaller pieces—each one fits into a single segment—and chains them together using segment headers. The decomposition is transparent to application developers; upon object read, Midas automatically reads all segregated pieces and stitches them back. This is possible thanks to Midas's pass-by-copy interface (§4.1).

Each allocated object has a 10-byte header inlined with its data, used for tracking the object's runtime information. This includes 1) a liveness bit, indicating whether the object has been deallocated; 2) an evacuating bit, marked by the evacuator to synchronize evacuation with object accesses; 3) hotness bits, a counter that will be incremented (or unchanged when it has reached the maximum) each time the object gets accessed; 4) a size field, indicating the total size of the object; 5) a reverse pointer field, used by the evacuator, if it moves the object, to rewrite the soft pointer.

4.2.2 Soft Memory Evacuator

As the allocation goes on, the application may eventually deplete the free segment list. It is the responsibility of Midas's evacuator to constantly release cold and dead objects, ensuring the best use of soft memory by only storing hot objects. In addition, the evacuator tracks segments in order of hotness in a used list (see Figure 5), to simplify the design and improve the speed of memory reclamation, in which the kernel forcibly unmaps application's soft memory pages under intense memory pressure (§4.3).

Midas's background thread continuously monitors the free segment ratio and triggers evacuation if it falls below a configurable threshold (our default value is 90%). The evacuation mainly consists of three stages:

Scanning Stage. The evacuator first scans through all objects in the used segment list. For each scanned object, it decrements the embedded hotness counter (similar to the CLOCK algorithm [15]). The evacuator treats objects with a zero pre-scanning hotness value, in addition to deallocated objects, as *dead* objects; they will be released in the compaction stage. The evacuator calculates the *live ratio* of each segment (*i.e.*, the percentage of live bytes) during scanning, and then uses it to sort all scanned segments to decide their priority for compaction. The segment with the lowest live ratio will be compacted first as it yields the largest benefits (in terms of the reclaimed space).

Compaction Stage. The evacuator compacts one segment at a time. For each live object, it first relies on the evacuation bit to synchronize with application threads to avoid data race (similar to AIFM [40]). It then copies the object into a new segment and leverages the reversed pointer field to rewrite the address of the corresponding soft pointer. After evacuating all live objects, it moves the segment from the used list into the free list.

Sorting Stage. After compaction, the evacuator calculates the segment-level hotness value for all segments in the used list, defined as $\sum_{\forall obj \in seg} SIZE(obj) \cdot HOTNESS(obj)$. It finally sorts the used list by segment-level hotness in ascending order.

```
1  for each segment S to compact {
2    D = pick_destination_segment();
3    for each object O in S {
4      try {
5        // A wrapper around our PF-resilient memcpy
6        copy_object_into(O, D);
7      } catch (SoftMemUnmapped &exception) {
8        if (exception.fault_addr belongs to O)
9          break; // Skip S as it has gone
10       else // It must belong to D
11         goto line2; // Pick a new D and restart
12     }
13   }
14 }
```

Listing 3: Midas implements its evacuator's compaction code using a page-fault-resilient memory copy function.

4.2.3 Page-Fault-Resilient Functions

Midas runtime directly manipulates soft memory during allocation and evacuation. Since the kernel may unmap soft pages to reclaim memory under pressure (details in §4.3), the runtime has to be aware of page faults and be able to recover from them gracefully. We carefully built the runtime to achieve this goal. First, we stored the important metadata (e.g., the free and used segment lists) in normal memory instead of soft memory, therefore it will not be lost under memory pressure. This is viable as the metadata only consumes little memory (less than 10 MiB). Second, we introduced *page-fault-resilient functions* and used them as primitives to build the runtime.

A page-fault-resilient function is able to capture any internal page fault that stems from dereferencing unmapped soft memory and respond to it by reverting all side effects and throwing a `SoftMemUnmapped(fault_addr)` exception to the caller. As a concrete example, in Midas we internally implemented a page-fault-resilient memory copy function, which is used to build the evacuator's compaction code to withstand page faults (see Listing 3). Page faults can happen when copying objects from the old segment into the new segment. To deal with this case, Midas uses its resilient memory copy function (line 7) to capture and handle the potential exception (lines 8-13).

Midas registers its own signal handler to facilitate capturing and handling all soft-memory-related page faults. Additionally, a page-fault-resilient function satisfies the following requirements to ensure resilience:

- It embeds a fault recovery code block for aborting the partial execution and rolling back side effects. Midas runtime maintains a mapping from resilient functions to their recovery blocks so that when page fault happens the handler can invoke the corresponding recovery code.
- All of its inner non-resilient functions have to be inlined to prevent the control flow from jumping out of its scope. Otherwise, the page fault handler is unable to find the corresponding recovery code.
- It preserves its stack frame base pointer (by disabling the compiler optimization) so that the fault handler can easily unwind its stack and throw an exception back to its caller.

4.3 Global Soft Memory Coordinator

Midas's global coordinator is responsible for granting server's idle memory to applications as soft memory and coordinating the budget across applications to optimize the overall performance.

4.3.1 Soft Memory Management Mechanism

The coordinator maps idle memory pages directly into an application's address space as soft memory segments. For each application, the coordinator dynamically maps or unmaps pages to readjust its soft memory budget. To facilitate the management, the application's runtime shares its free segment list and used segment list with the coordinator.

To grant more soft memory to an application, the coordinator maps more pages to it and inserts them into the free segment list. Similarly, to reclaim memory from an application, the coordinator unmaps pages. The coordinator first tries to pop out and unmap the segments from the free list; since they do not hold any useful live objects, unmapping them does not incur any impact on the application's performance. Meanwhile, the runtime strives to avoid the exhaustion of the free list by triggering evacuation (§4.2.2).

The synergy between the runtime and the coordinator is able to handle moderate memory pressure (*i.e.*, the common case). However, under severe pressure, the evacuation may fall behind, leading to an empty free segment list. To avoid depletion, the coordinator reacts by unmapping used segments which may induce performance disruption in two folds. First, when the application later tries to access an unmapped object, the runtime will experience a page fault which incurs overhead. Second, the runtime has to spend additional time reconstructing the missing object. To alleviate this issue, the coordinator prioritizes cold segments over hot segments. Thanks to the evacuator, the segments in the used list have been ordered by their hotness (§4.2.2). Therefore, the coordinator can realize prioritization by simply unmapping segments based on their order in the list.

4.3.2 Coordination Policy

Midas continuously adjusts each application's soft memory budget by solving the following optimization problem:

$$\underset{m}{\text{maximize}} \sum_{\forall i \in APPS} w_i \Gamma_i(m_i) \text{ , subject to } \sum_{\forall i \in APPS} m_i = M$$

For each application i, w_i denotes its weight (which is either specified by the operator or uses the default value 1) and Γ_i denotes its performance utility when assigned soft memory of size m_i. The server-wide overall utility is defined as the weighted sum of all application's utilities. M denotes the server's total idle memory.

By default, the coordinator estimates Γ_i as $-RCOST_i$, where $RCOST_i$ is the application's CPU usage spent on reconstructing missing objects. Midas's runtime can easily

collect this per-application information and report it to the coordinator. Developers can also plug in the real performance metric reported by applications—which already exists in many datacenter applications [11]—for a more faithful Γ_i.

Midas solves the optimization problem using the hill climbing approach [42]. It periodically probes every application's marginal utility benefit $\frac{\partial \Gamma_i(m_i)}{\partial m_i}$ by additionally assigning a small portion of memory Δ_{m_i} and monitoring the change of utility Δ_{Γ_i}. Midas regrants the soft memory budget from the application with the lowest marginal utility benefit to the one with the highest benefit.

In contrast, Cliffhanger (a recent cache service) [13] adopts a coordination policy that optimizes for the overall cache hit rate, but this does not necessarily optimize the overall performance. For example, caching objects that are frequently accessed may not be helpful if they can be cheaply reconstructed. Midas avoids this issue by using both access frequency and reconstruction cost as metrics for optimization.

4.4 Discussion

Though Midas is mainly designed for caching hot data and memoizing intermediate computation results, developers have the freedom to put any data into soft memory as long as it is reconstructible. However, storing data that is expensive to reconstruct but infrequently accessed can lead to performance issues. Midas provides a profiling tool that generates runtime warnings if such cases are detected. In addition, Midas offers a debugging mode where we validate the reconstruction logic by calling the user-defined reconstruction function and comparing its result with the actual cached object using the object's comparison operator. Bugs are reported if these objects are not identical.

Midas also incorporates Linux's page cache by simply treating it as another per-application soft memory pool. For each application, Midas's shim layer intercepts all POSIX file operations and caches the file data using a soft hash table, whose keys are file inode numbers along with block-aligned offsets and values are file blocks. The reconstructor rebuilds the missing block by performing the actual file read.

5 Implementation

Midas is implemented in C++ and includes bindings for C. Our implementation has 2,814 LOC for the soft memory abstraction (§4.1), 3,866 LOC for the runtime (§4.2), and 1,029 LOC for the global coordinator (§4.3).

Soft data structures store their metadata (*e.g.*, a hash table's bucket array that stores indices) in normal memory and store their data payload (*e.g.*, a hash table's key-value pairs) in soft memory using soft pointers.

The log-structure allocator enforces 16-byte alignment for allocated data to make it GCC-compatible. The evacuator adopts a concurrent pauseless design similar to AIFM [40]. The evacuator ensures atomicity when evacuating or reconstructing large objects that span across multiple soft memory segments. Midas registers its own `SIGSEGV` handler. For each segmentation fault, the handler checks whether the faulting memory address belongs to a soft memory region and whether the faulting program counter (PC) belongs to a page-fault-resilient function; for faults that do not meet these conditions, the handler treats them as unrecoverable exceptions and aborts the program. To facilitate the PC check, Midas leverages a linker script to place all resilient functions into a separate code segment whose layout is known at compile time.

During each application's initialization, the runtime registers itself to the global coordinator using `ioctl` and uses `mmap` to create a shared memory region for exposing information—including its free segment list and used list (implemented as arrays) and the application's reconstruction cost (implemented as a counter)—to the coordinator.

We implemented the global coordinator as a user-space daemon (that runs the coordination policy) and a privileged kernel module (that executes the coordination decision by mapping/unmapping pages to/from user processes directly). Every 5 seconds, the coordinator probes the marginal utility of each application and makes a new adjustment to soft memory budgets. It probes an application by either granting or revoking 64 MiB soft memory and monitoring its performance change. In each adjustment, it regrants up to 256 MiB soft memory from the application with the lowest marginal utility to the one with the highest utility. To avoid oscillation, it refrains from granting more soft memory to the application until it has consumed the additional memory offered in the previous round.

6 Programming with Midas

We present general guidelines of programming with Midas (§6.1) followed by concrete examples of porting four real applications (§6.2).

6.1 Guidelines

When is it safe to use soft memory? Developers can generally use soft memory to store any application data that follows the unmap-and-reconstruct semantics. To support evacuation, developers have to implement copy constructors for objects stored in soft memory.

When is it beneficial to use soft memory? Developers should generally consider using soft memory when applications can opportunistically benefit from having additional memory. Typical use cases include caching in web applications and memoization in data analytics systems. They often have unknown marginal utility and unbounded memory footprint, making them hard to handle efficiently through static provisioning. Midas can benefit them by automatically rightsizing their soft memory budget and harvesting idle memory.

Applications	Abstractions used	Porting effort (LOC)	CPU cores	Normal mem. (GiB)	Peak soft mem. (GiB)	Reconstruction cost (µs/obj.)	Dataset
HDSearch [46]	Soft hash table	36	12	1.7	13.6	1244.2	`OpenImg` [25], 1.9M images
WiredTiger [33]	Soft pointer	332	12	3.7	21.3	20.6	Facebook `USR` [8], 50M KV
Storage Server [24]	Soft array	29	4	1.1	20.4	10.5	`multilate` [23], 16 GiB disk
SocialNet [18]	Soft hash table Soft queue	175	20	1.3	12.2	99.1–3227.7	Socfb-Penn94 [39], 41.5K nodes, 1.4M edges

Table 1: We ported four applications into Midas with low programming effort. All four applications extensively use soft memory while their data reconstruction costs vary drastically.

How to migrate from traditional cache services? Existing applications that employ local cache services (*e.g.*, Memcached [30] or Redis [2]) can directly use Midas as a drop-in replacement. Existing applications that employ distributed cache services (*e.g.*, AWS ElastiCache [1]) can use Midas as a fast local cache tier to reduce the overhead of accessing remote cache.

6.2 Application Case Studies

We ported four applications to Midas. They cover a range of CPU usage, normal and soft memory usage, data reconstruction cost, and Midas's abstraction usage (see Table 1).

HDSearch [46] is an image search service based on content similarity. For each query, a feature extraction backend transforms the input image into a feature vector via a DNN (running on GPU), and then caches the result along with a hash of the image (using Memcached for memoization). To port this application, we replaced Memcached with our soft hash table, which only involves 36 LOC changes. It has 1.7 GiB normal memory usage and 13.6 GiB peak soft memory usage. Reconstructing KV pairs is expensive (1244.2 µs per object) as it requires re-performing transformation on GPU.

WiredTiger [33] is a NoSQL key-value storage engine used by MongoDB [32]. It persists all key-value pairs in storage indexed via an in-memory B+ tree. It has a built-in in-memory caching engine that caches the data of B+ tree's internal nodes and leaf nodes to reduce expensive storage I/Os. To port WiredTiger, we implement its caching engine using Midas's soft memory pool and pointer abstractions; we created a soft memory pool with a reconstruction method that wraps WiredTiger's existing code for handling cache misses, and replaced ordinary B+ tree pointers with soft memory pointers allocated from the pool. This only involves 332 LOC changes. With our port, WiredTiger has 3.7 GiB normal memory usage and 21.3 GiB peak soft memory usage. Reconstructing a tree node object requires reading its content from the disk and rebuilding the index, which takes 20.6 µs.

Storage Service is an NVMe-based block storage service similar to Reflex [24]. It exposes a standard block I/O interface using RPC to support accessing 4KiB storage blocks remotely. Its original design uses SPDK [3] to communicate with the storage block device, which bypasses Linux's page cache. To port it, we cache the block data using a soft array, similar to the `BlockCache` design in Listing 2. This requires adding 29 LOC. With our port, it uses 1.1 GiB normal memory and 20.4 GiB peak soft memory. Reconstructing an array element requires a block I/O which takes 8.5 µs to finish.

SocialNet is a twitter-like latency-critical web service from DeathStarBench [18]. It is built using 12 microservices with sophisticated fan-out patterns and call dependencies. Its original design uses Memcached/Redis to cache users' data and memoize results of certain queries, and employs pools to cache TCP connections/RPC sessions. Since each microservice has its own binary and runs within its own process, Midas treats SocialNet as 12 different applications. To port it, for each microservice, we replace its Memcached/Redis usage with Midas's soft hash table and connection pool with Midas's soft queue; this involves 175 LOC changes. With our port, it uses 1.3 GiB normal memory and 12.2 GiB peak soft memory. It takes 99.1–3227.7 µs to reconstruct an object depending on its type; for example, it takes only 99.1 µs to re-establish an RPC session but requires 3227.7 µs to re-fetch a user's post.

7 Evaluation

Our evaluation seeks to answer the following questions:

1. Can Midas judiciously coordinate soft memory among applications to optimize overall performance? (§7.1)
2. Can Midas quickly and reactively harvest available idle memory to improve utilization and performance? (§7.2)
3. Can Midas quickly react to memory pressure to avoid out-of-memory killing while maintaining good performance? (§7.3 and §7.4)
4. How does the data reconstruction cost of an application affect its performance? (§7.4)

Setup. We conducted experiments on one server that equips a 48-core Intel Xeon Gold 6252 CPU and 128 GiB memory. The server ran Ubuntu 20.04 with Linux 5.14. In line with prior work [37], we enable hyperthreading, but disable dynamic CPU frequency scaling, transparent huge pages, and kernel mitigations for transient execution attacks. For interactive services (*e.g.*, SocialNet), we use a separate server to generate load, which connects to the application server via a 10 GbE network. For all four applications, we generated requests with Zipfian distribution, consistent with the study of real datacenter workloads [9].

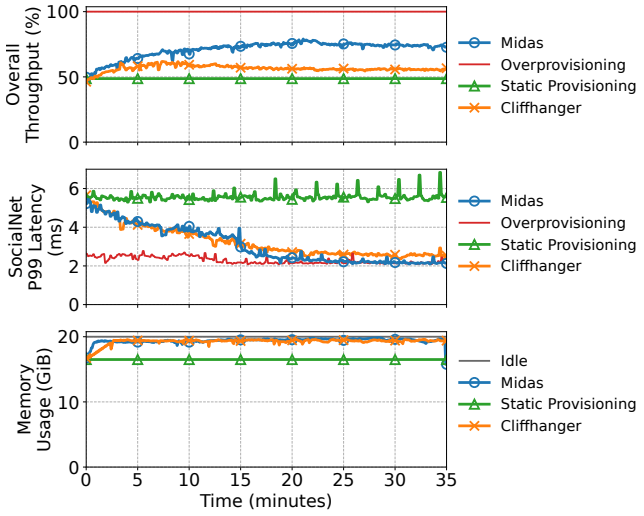


Figure 6: When co-running four applications with 20 GiB idle memory, Midas dynamically coordinates their soft memory budgets and reaches an equilibrium in around 20 minutes. Overall, it harvests 19.6 GiB idle memory as soft memory and achieves 75.0% of the ideal throughput (measured by overprovisioning soft memory for all applications regardless of the 20 GiB total budget constraint).

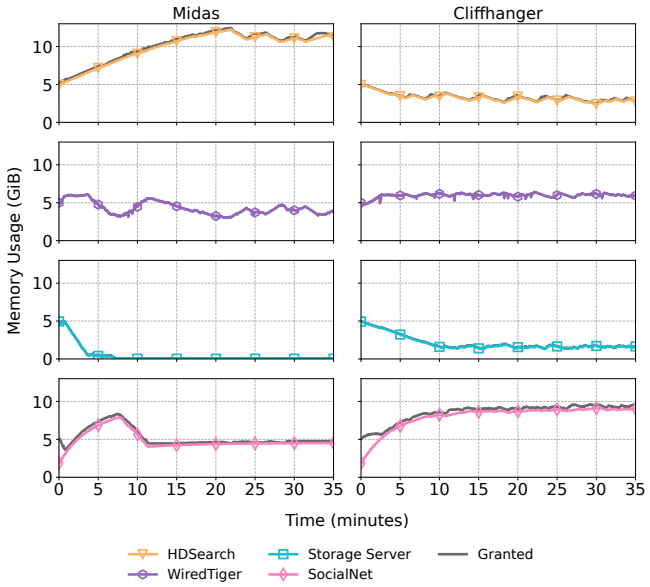


Figure 7: Midas and Cliffhanger converge to different allocations of soft memory between applications because of fundamental differences in their coordination policies.

7.1 Coordinating Soft Memory

In this experiment, we investigated whether Midas can judiciously coordinate soft memory usage among applications to optimize overall performance.

We provisioned the server with 20 GiB idle memory and co-ran all four applications (§6) using Midas. Initially, all applications start with the same amount of soft memory (*i.e.*, 5 GiB), but Midas will dynamically adjust it. SocialNet has 12 loosely-coupled microservices and we start by evenly splitting the 5 GiB budget across them. We measured the overall throughput (defined as the average of all applications' throughput normalized to their ideal throughput) and the soft memory usage. We compared Midas with three different baselines. The first baseline *overprovisions* soft memory for each application to cache all of possible soft state. This leads to a 67.5 GiB soft memory usage that is impossible to achieve under 20 GiB idle memory; thus, this represents the ideal throughput. The second baseline limits itself to the 20 GiB soft memory budget and *statically partitions* it across four applications in an even manner (*i.e.*, each application gets 5 GiB soft memory). The third baseline is *Cliffhanger* [13]. Similar to Midas, it dynamically coordinates soft memory among applications. However, it adopts a different coordination policy of maximizing the global cache hit rate as opposed to maximizing the overall performance utility. As the original version of Cliffhanger only supports Memcached, we emulated Cliffhanger by implementing its coordination policy atop Midas.

A good result for Midas would show that it quickly reaches an equilibrium by judiciously coordinating soft memory usage among applications and achieves good overall throughput close to the ideal throughput (of the overprovisioning baseline). In contrast, the overall throughput of the static provisioning baseline should be suboptimal, as it equally treats all applications and fails to prioritize the soft memory need of applications that can benefit the most. On the contrary, Cliffhanger does coordinate soft memory among applications, but it optimizes for the overall cache hit rate which does not guarantee optimal overall performance (§4.3.2). Therefore, we expect Cliffhanger to achieve overall throughput better than the static baseline but worse than Midas.

Figure 6 shows the results. The top figure presents the overall throughput of four systems normalized to the ideal value. The bottom figure presents soft memory usage; we leave out the usage of the overprovisioning baseline as it is much higher (67.5 GiB) than the amount of idle memory (20 GiB). Midas's overall throughput converges in around 20 minutes and achieves 75.0% of the ideal throughput by harvesting 98.0% idle memory. It also reduces SocialNet's 99th percentile latency by 58.4% from 5.5ms to 2.3ms. In contrast, the static provisioning baseline only achieves 48.7% of the ideal throughput and fails to improve SocialNet's tail latency due to the lack of coordination. It also uses 3.1 GiB less soft memory than Midas as some microservices of SocialNet fail to fully use their statically-provisioned soft memory budgets due to small soft memory footprints. Cliffhanger uses a similar amount of soft memory to Midas. Due to its coordination policy, it converges on the overall cache hit rate (not shown due to the space constraint) but oscillates in terms of the overall throughput. Therefore, it only achieves 56.0% throughput on average.

Figure 7 presents the per-application soft memory usage of Midas and Cliffhanger. For each application, the gray line

represents the soft memory budget it receives, while the colorful line represents the amount of soft memory it uses. Because of the difference in their coordination policies, Midas and Cliffhanger make very different allocations of soft memory between applications except for the storage server. For example, since it is time-consuming to reconstruct HDSearch's objects (as it involves recomputing the feature vectors of images), Midas scales up HDSearch's soft memory to cache more objects. However, since HDSearch has a relatively low request skewness (compared to other applications) and consequently a lower cache utility (in terms of hit rate), Cliffhanger deprioritizes it by scaling down its soft memory, significantly impacting its performance (and therefore the overall performance).

In summary, the experiment demonstrates that Midas can efficiently utilize available memory as soft memory and judiciously coordinate soft memory among applications, achieving high overall performance close to the ideal one that requires 3.4× more memory.

7.2 Harvesting Available Idle Memory

In this experiment, we investigated whether Midas can quickly and reactively harvest additional idle memory—whenever it is available—to improve memory utilization and application performance.

We ran an application using Midas and dynamically added idle memory to the server. A good result for Midas would show that it quickly detects any new idle memory and reactively grants it to SocialNet as additional soft memory to improve performance. Additionally, we expect that the marginal benefit decreases as SocialNet uses more soft memory and caches more hot items.

Figure 8 presents the results of SocialNet. The results of other applications show similar trends and can be found in Appendix A. Initially, the server has 2 GiB idle memory (the dark gray line). With Midas, SocialNet fully utilizes them as soft memory (the blue line) and achieves 13 MOPS throughput (the pink line). At $t = 5$min, we added 4 GiB more idle memory to the server. Midas immediately detects this change and rapidly ramps up its soft memory usage; it only takes around 3 minutes for SocialNet to reach a new steady state. Benefiting from more soft memory, SocialNet's throughput increases by 46% from 13 MOPS to 19 MOPS, and its 99th percentile latency decreases by 27% from 5.5ms to 4ms (the light brown line). At $t = 15$min, we again added 4 GiB more idle memory. This time we observed a reduced marginal benefit as SocialNet has already cached most hot items; it takes 15 minutes to reach a new equilibrium (*i.e.*, 8.5 GiB soft memory usage) and yields a 43% improvement of 99th percentile latency (from 4ms to 2.3ms).

In summary, these results highlight Midas can quickly detect idle memory and reactively scale up its soft memory usage to improve memory utilization and application performance.

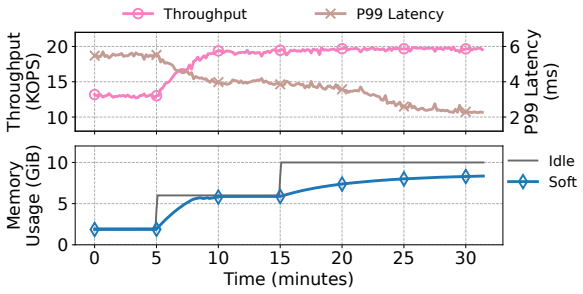


Figure 8: With Midas, SocialNet effectively harvests additional idle memory by scaling up its soft memory usage, improving both throughput and tail latency.

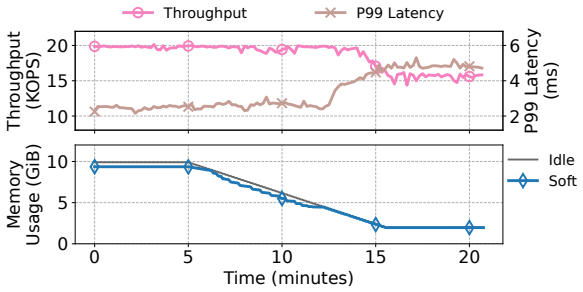


Figure 9: Under moderate memory pressure ($t = 5$min-15min), Midas is able to reactively scale down SocialNet's soft memory usage to avoid running out of memory with moderate performance impact.

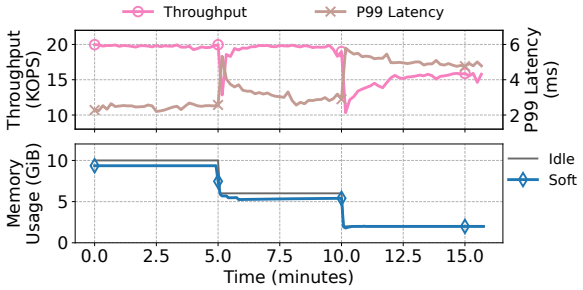


Figure 10: Midas is able to avoid out-of-memory killing even under extreme memory pressure ($t = 5$min and $t = 10$min). SocialNet experiences brief throughput collapses and tail latency spikes but quickly recovers to normal once the pressure is finished.

7.3 Reacting to Memory Pressure

In this experiment, we investigated whether Midas can quickly react to memory pressure to avoid out-of-memory killing and studied its impact on application performance.

Similar to §7.2, we ran SocialNet using Midas, but dynamically decreased the server's idle memory with a colocated memory antagonist. We measured the impact on SocialNet's soft memory usage and performance.

Under moderate memory pressure, ideally, Midas's global coordinator should reactively unmap free soft memory segments while Midas's evacuator should be able to replenish them (by evicting cold objects and evacuating hot objects) to match the coordinator's unmapping rate. A good result for Midas would show that SocialNet's throughput degrades gradu-

ally and mildly as the pressure persists, since Midas prioritizes the eviction of cold and dead objects over hot objects.

Under intense memory pressure, we expect the coordinator to also unmap the used soft memory segments as the evacuator cannot keep up with the high unmapping rate. In this case, SocialNet may experience a sudden throughput collapse due to the loss of hot objects. However, a good result for Midas would show that SocialNet is still able to operate without experiencing any out-of-memory killing. In addition, immediately after the pressure is finished, SocialNet's performance should be able to recover to normal by reconstructing back hotter objects and evicting colder objects.

Figure 9 presents the results under moderate memory pressure. Initially, the server has 10 GiB soft memory. The application uses around 9.6 GiB of it as soft memory and achieves around 20 MOPS throughput and 2.3ms 99th percentile latency. At $t = 5$min, the memory antagonist starts to allocate 8 GiB more memory at a moderate rate of 0.8 GiB/min, resulting in the decrease of idle memory until $t = 15$min. As shown by the bottom figure, Midas is able to reactively scale down SocialNet's soft memory usage through reclamation to avoid running out of memory. As shown by the top figure, SocialNet's throughput and 99th percentile latency remain unaffected in the beginning, as Midas prioritizes the reclamation of cold soft memory. After running below 5 GiB idle memory, SocialNet experiences a mild throughput drop and latency increase, as Midas starts to reclaim hotter soft memory.

Figure 10 presents the results under intense memory pressure. In this case, the antagonist allocates memory as fast as Linux permits (7 GiB/s), making it an extremely challenging case to handle. Despite the high rate, Midas is still able to avoid out-of-memory killing by rapidly scaling down SocialNet's soft memory usage. In this case, Midas has to unmap the used soft memory segments, inevitably causing brief throughput collapses and latency spikes (at $t = 5$min and $t = 10$min). However, once the memory pressure is finished, SocialNet's throughput and latency quickly recovers to the normal level, consistent with the numbers reported in Figure 8 and 9.

The results of other applications also show similar trends (see Appendix B). In summary, these results demonstrate that Midas can always quickly react to memory pressure to avoid out-of-memory killing while maintaining good application performance whenever it is possible.

7.4 Design Drill-Down

Soft Pointer Dereference Cost. We measured the latency of dereferencing a soft pointer and compared it to the latency of dereferencing an ordinary C++ `unique_ptr`, when the pointer and data pointed to are originally in memory (*i.e.*, not in CPU's cache). Table 2 shows the results of small objects (32 B) and large objects (4 MiB), and Appendix D has more results of other object sizes.

For small objects that fit into CPU's cache line (Table 2a), Midas is able to deliver comparable read latency as its extra object copying overhead is negligible. Midas achieves higher write latency (< 200 cycles) as it has to additionally update the metadata in the object header.

[*read*|*write*] Latency (cycles)	Average *read/write*	Median *read/write*	P90 *read/write*
C++ `unique_ptr`	367 / 199	382 / 176	510 / 332
`SoftUniquePtr`	400 / 393	370 / 368	516 / 500

(a) Small objects (32 B).

[*read*|*write*] Latency (Mcycles)	Average *read/write*	Median *read/write*	P90 *read/write*
C++ `unique_ptr`	0.97 / 1.39	0.94 / 1.36	0.99 / 1.37
`SoftUniquePtr`	1.77 / 1.15	1.75 / 1.14	1.77 / 1.18

(b) Large objects (4 MiB).

Table 2: Midas' soft pointer only adds moderate dereferencing cost compared to C++'s ordinary smart pointer.

Live Object Ratio		10%	30%	50%	70%	90%
Reclamation Tput. (MiB/s)	Cooperative	312.5	243.1	173.6	104.2	34.7
	Direct	8268.5				

Table 3: Midas's cooperative reclamation reclaims memory at the throughput of 35 MiB/s-313 MiB/s, depending on the live object ratio of soft memory. Midas's direct reclamation trades off reclamation quality for faster speed; it achieves a throughput of 8269 MiB/s, exceeding the rate at which the Linux kernel can allocate memory.

For large objects (Table 2b), Midas achieves ≈800K cycles (82%) higher read latency since now the additional object copying happens in memory (rather than in CPU's cache). However, Midas achieves lower write latency than `unique_ptr` thanks to its optimized memory copy implementation.

Memory Reclamation Speed. We measured Midas's memory reclamation throughput using a synthetic microbenchmark. Under moderate memory pressure, the coordinator reclaims memory with the cooperation from the runtime (Figure 9); we refer to it as *cooperative reclamation*. Under severe memory pressure, the coordinator directly unmaps soft memory segments (Figure 10); we refer to it as *direct reclamation*.

Table 3 presents the throughput of both reclamation methods. The speed of cooperative reclamation depends on the live object ratio of soft memory; the lower the live ratio, the easier to make room by compacting hot objects, thereby yielding faster reclamation speed. It achieves a throughput of 313 MiB/s under 10% live ratio and 35 MiB/s under 90% live ratio. To handle intense memory pressure, direct reclamation trades off reclamation quality for faster reclamation speed; it achieves a significantly higher throughput of 8269 MiB/s, unrelated to the live object ratio. This exceeds the rate at which the Linux kernel can allocate memory (7-8 GiB/s measured in our machine), therefore Midas can always safely harvest server's idle memory without leading to OOM killing.

Performance Impact of Data Reconstruction. To examine the performance impact of using soft memory, we conducted

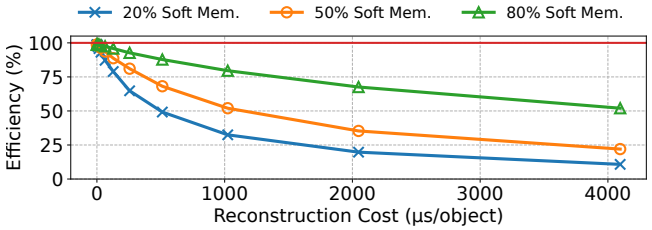


Figure 11: Midas's efficiency (*y*-axis) as a function of data reconstruction cost normalized to the ideal throughput of caching all soft state. Midas's efficiency increases as the reconstruction cost decreases, delivering >80% efficiency for applications with <1024 µs/object reconstruction cost when caching 80% of soft state.

an experiment using a synthetic application; we measured its performance with varying data reconstruction costs under different soft memory ratios (*i.e.*, the ratio of cached soft state). Intuitively, the cheaper the data reconstruction, the lighter the performance impact it incurs.

Figure 11 shows the result. When the soft memory ratio is 20%, Midas is able to deliver >80% efficiency when the reconstruction cost is <128 µs/object. When the soft memory ratio is higher, Midas can tolerate a higher cost as reconstruction happens less frequently; thus, it is able to provide >80% efficiency for applications with <256 µs/object reconstruction cost under 50% memory ratio and <1024 µs/object under 80% memory. This suggests that Midas can still achieve high performance with moderate data reconstruction costs.

8 Related Work

Resource Harvesting and Deflation. Datacenters today suffer from low resource utilization [6, 17, 50]. To make use of vacant resources, major cloud providers now offer spot VMs [5, 20, 31], which run at a low priority and get evicted under resource pressure. Others propose new VM designs to gracefully adjust VMs' resource usage. Harvest VM is a new type of VM that grows and shrinks according to the amount of unallocated resources at its underlying server, including CPU [6], memory [17], and storage [38]. Similarly, deflatable VM [45] codesigns the hypervisor, VM, and the application to reclaim resources from applications under memory pressure. These approaches focus on VMs only, and take minutes to re-configure a VM to release resources.

Resource Disaggregation and Remote Memory. Resource disaggregation and remote memory systems are another trending approach for improving utilization, thanks to faster datacenter networking [19, 26, 29]. Their key idea is to break the server hardware boundary with a fast network interconnection to exploit stranded resources on a remote server. Various systems have established the viability of disaggregated storage [22, 24], accelerators [35, 51], and memory [4, 21, 44, 54]. While some provide remote memory transparently via OS paging, it is also possible to use a library-based approach that modifies the application to bypass the OS. AIFM [40] proposes remote-able data structures to build remote-memory-aware applications. Semeru [52], Mako [28], and MemLiner [53] co-design the JVM with the kernel to offer transparent remote memory for Java programs. Like Midas, these systems adopt customized pointer formats for their remote-able objects. Unlike Midas, they do not consider the unmap-and-reconstruct semantics and suffer from swapping or out-of-memory killing under intense memory pressure.

Cache Services. Improving cache performance is important to datacenter applications, especially in a shared setting [10, 36]. Fairride [36] and RobinHood [10] provide fair and latency-aware cache-sharing policies, and CliffHanger [13] uses a hill climbing method to incrementally optimize cache allocation across applications. Memshare [14] further improves the cache partitioning with a log-structured allocator for higher hit rates. However, existing cache service systems still rely on static memory allocation, and cannot efficiently use idle memory. CacheLib [9] provides a library-based approach for caching, but it again relies on static provisioning and lacks global coordination, hindering its ability to manage memory across multiple applications.

Cooperative Memory Revocation. In parallel with our work, researchers are also exploring the benefits of soft state by managing it at the application level [16]. Midas instead uses kernel coordination and unmap-and-reconstruct semantics, which enables it to reclaim pages even if applications do not cooperate or are slow to respond. This makes it possible to react to severe memory pressure without running out of memory.

9 Conclusion

In this paper, we presented Midas, a system that efficiently and safely harvests idle memory to store the soft state that is most beneficial to each application, improving both memory utilization and application performance. Midas provides familiar high-level programming abstractions and maximizes overall performance through coordination between an application-integrated runtime and a global coordinator. Our evaluation demonstrates that Midas is able to effectively use soft memory to achieve near-optimal performance and can respond to extreme memory pressure fast enough to avoid running out of memory.

Acknowledgement

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Sanidhya Kashyap for his feedback. This work is supported by NSF grants CNS-1907352, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, CCF-1764077, CHS-1956322, CCF-1460325, CCF-2106404, and CNS-2104398, an Amazon Ph.D. fellowship, a gift from Amazon, a contract from Samsung, and a contract from Cisco.

References

[1] Amazon elasticache. https://aws.amazon.com/elasticache/, 2023.

[2] The redis database. https://redis.com/, 2023.

[3] Storage performance development kit. https://spdk.io, 2023.

[4] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[5] Amazon Elastic Compute Cloud. Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot, 2022.

[6] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini. Providing SLOs for Resource-Harvesting VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, Nov. 2020.

[7] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, Apr. 2018. USENIX Association.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[9] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, Nov. 2020.

[10] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, Oct. 2018. USENIX Association.

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[12] M. Calder, R. Gao, M. Schröder, R. Stewart, J. Padhye, R. Mahajan, G. Ananthanarayanan, and E. Katz-Bassett. Odin: Microsoft's scalable Fault-Tolerant CDN measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 501–517, Renton, WA, Apr. 2018. USENIX Association.

[13] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, Mar. 2016. USENIX Association.

[14] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.

[15] F. Corbató. *A Paging Experiment With the MULTICS System*. Project MAC. Massachusetts Institute of Technology, 1968.

[16] M. Frisella, S. L. Sanchez, and M. Schwarzkopf. Towards increased datacenter efficiency with soft memory. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 127–134, New York, NY, USA, 2023. Association for Computing Machinery.

[17] A. Fuerst, S. Novaković, I. n. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. 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 '22, page 583–594, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[19] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, pages 249–264, 2016.

[20] Google Cloud. Preemptible vm instances. https://cloud.google.com/compute/docs/instances/preemptible, 2022.

[21] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[22] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP ≈ RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, Feb. 2020. USENIX Association.

[23] Jacob Leverich. Mutilate: High-performance memcached load generator. https://github.com/leverich/mutilate, 2023.

[24] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash ≈ local flash. In *ASPLOS*, pages 345–359, 2017.

[25] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *IJCV*, 2020.

[26] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[27] C. Lu, K. Ye, G. Xu, C. Xu, and T. Bai. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *Big Data*, pages 2884 – 2892, 2017.

[28] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, 2022.

[29] Mellanox. RDMA aware programming manual (rev. 1.7). https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[30] Memcached. A distributed memory object caching system. http://memcached.org, 2020.

[31] Microsoft Azure. Azure spot virtual machines. https://azure.microsoft.com/en-us/pricing/spot, 2022.

[32] MongoDB. https://www.mongodb.com/, 2022.

[33] MongoDB. Wiredtiger storage engine. https://www.mongodb.com/docs/manual/core/wiredtiger/, 2023.

[34] U. Naseer and T. A. Benson. Configanator: A data-driven approach to improving CDN performance. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1135–1158, Renton, WA, Apr. 2022. USENIX Association.

[35] Nvidia. Virtual gpu (vgpu) | nvidia. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[36] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. FairRide: Near-Optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, Mar. 2016. USENIX Association.

[37] Y. Qiao, C. Wang, Z. Ruan, A. Beley, Y. Zhang, M. Kim, and G. H. Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Apr. 2023.

[38] B. Reidys, J. Sun, A. Badam, S. Noghabi, and J. Huang. BlockFlex: Enabling storage harvesting with Software-Defined flash in modern cloud platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 17–33, Carlsbad, CA, July 2022. USENIX Association.

[39] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[40] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, Nov. 2020.

[41] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for DRAM-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, Feb. 2014. USENIX Association.

[42] S. J. Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[43] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 455–468, 2016.

[44] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[45] P. Sharma, A. Ali-Eldin, and P. Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[46] A. Sriraman and T. F. Wenisch. μsuite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.

[47] K. Tati and G. M. Voelker. Shortcuts: Using soft state to improve dht routing. In C.-H. Chi, M. van Steen, and C. Wills, editors, *Web Content Caching and Distribution*, page 44–62, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[48] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: The next generation. In *EuroSys*, 2020.

[49] Twitter Inc. Processing billions of events in real time at twitter. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2021/processing-billions-of-events-in-real-time-at-twitter-, 2021.

[50] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

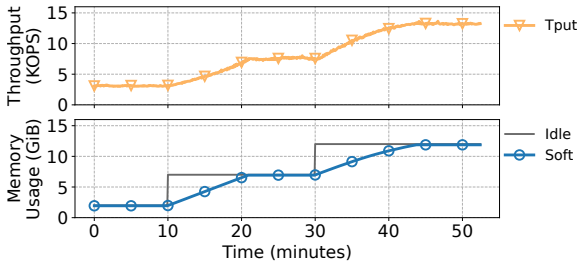
[51] L. Vilanova, L. Maudlej, S. Bergman, T. Miemietz, M. Hille, N. Asmussen, M. Roitzsch, H. Härtig, and M. Silberstein. Slashing the disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 352–367, New York, NY, USA, 2022. Association for Computing Machinery.

[52] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, Nov. 2020.

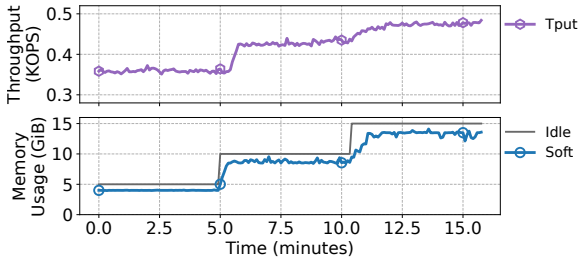
[53] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. MemLiner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, 2022.

[54] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*, 2023.

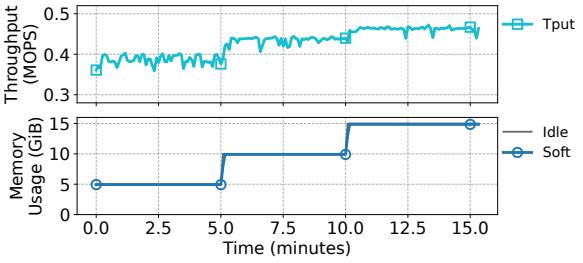
A Harvesting Available Idle Memory



(a) HDSearch.



(b) WiredTiger.

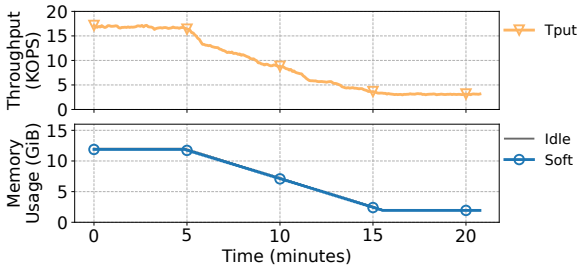


(c) Storage server.

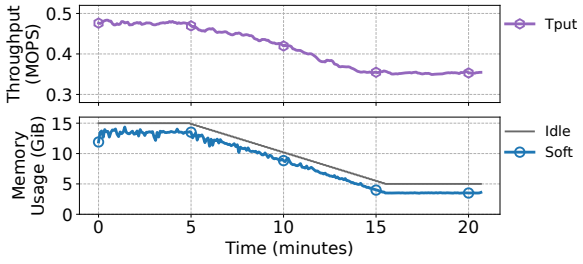
Figure 12: With Midas, applications can reactively scale up their soft memory usage to harvest additional idle memory and improve performance.

In this section, we evaluated the other three applications individually under the same setting as in §7.2 to show how Midas can harvest available idle memory to improve memory utilization and application performance. Figure 12 presents the results. Similar to Figure 8, Midas can quickly detect any idle memory and reactively grant it to the application to improve its performance. As applications expose different allocation speeds and utilities of their soft state, the average time to scale up the soft memory usage as well as the performance gain also varies across applications. HDSearch takes longer to fully utilize the additional soft memory because it needs expensive GPU computations to re-construct a cache-missed object. It also enjoys higher throughput increases by memoizing computation results with additional soft memory. On the contrary, both WiredTiger and storage server can quickly utilize all additional soft memory, but they only get marginal performance improvement after caching most hot blocks at $t = 10$min.

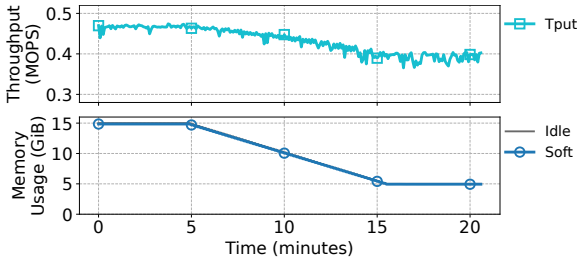
B Reacting to Memory Pressure



(a) HDSearch.



(b) WiredTiger.



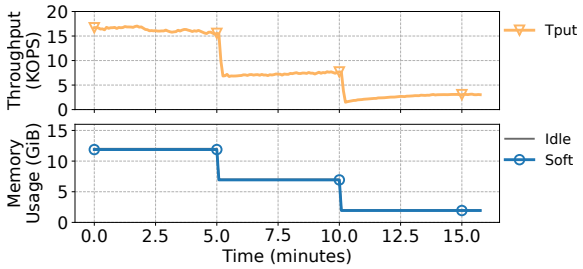
(c) Storage server.

Figure 13: Under moderate memory pressure (t = 5min-15min), Midas is able to reactively scale down applications' soft memory usage to avoid running out of memory while minimizing its impact on their throughput.

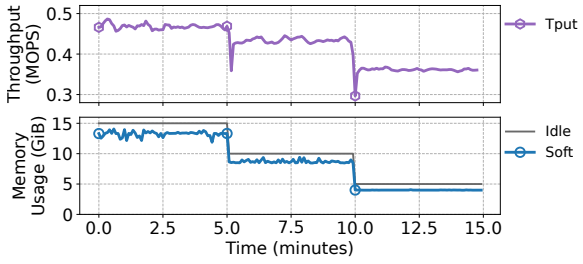
In this section, we further investigated whether Midas can quickly react to memory pressure by running the other three applications individually under the same setting as in §7.3. Similarly, we measured memory utilization and application throughput under moderate memory pressure and intense memory pressure, respectively.

Figure 13 presents the results of each individual application under moderate memory pressure. Similar to Figure 9, At t = 5min, the memory antagonist starts to allocate 10 GiB more memory with a moderate rate of 1.0 GiB/min, leading to the decrease of idle memory until t = 15min. As shown by the bottom figure, for all three applications, Midas reactively scaled down their soft memory usage and avoided out-of-memory killing. As shown by the top figure, application throughput drops gradually and mildly as the reclamation goes on and never experiences any severe disruption.

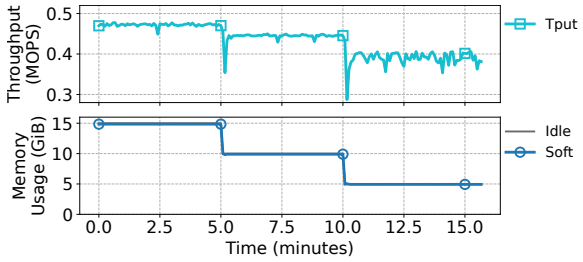
Figure 14 shows the results of each individual application under intense memory pressure. Similar to Figure 10, the



(a) HDSearch.



(b) WiredTiger.



(c) Storage server.

Figure 14: Midas is able to avoid out-of-memory killing even under extreme memory pressure (t = 5min and t = 10min). The victim application experiences brief throughput collapses but quickly recovers to normal once the pressure is finished.

memory antagonist intensely allocated 5 GiB memory at $t = 5$min and $t = 10$min. Despite the high memory allocation rate, Midas is still able to rapidly reclaim application's soft memory and avoid running out of memory. Because Midas has to unmap the used soft memory segments in this case, both WiredTiger and Storage server experience brief throughput collapses. However, once the memory pressure is finished, their throughput can quickly recover to the normal level, consistent with the numbers reported in Figure 12 and 13. HDSearch has a relatively lower request rate, therefore it is more tolerable to the enforced soft memory unmapping and does not experience severe throughput collapse at all.

C SocialNet Microservices Memory Usage

We have reported the overall soft memory usage of SocialNet in Figure 7. Among SocialNet's 12 microservices, two microservices used the most soft memory, namely UserTimeline and PostStorage. Figure 15 reported the detailed memory usage for each of them.

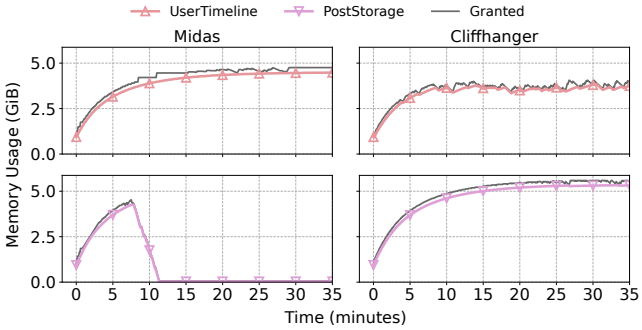


Figure 15: Memory usage of two major microservices in SocialNet. Midas dynamically coordinates memory between the microservices to achieve high memory utilization and optimal performance for SocialNet.

UserTimeline is the frontend microservice that handles user requests. It fetches a group of user posts from the storage backend and composes them as a timeline webpage. It caches composed user timelines in soft memory to reduce backend storage accesses. PostStorage is the backend database microservice that stores user posts. It handles post requests from UserTimeline with MongoDB and caches hot posts using soft memory. As shown in 15, at first, Midas reactively grants soft memory to both microservices to quickly recover SocialNet's throughput and latency. As UserTimeline gets more soft memory, it caches more hot timelines and consequently reduces its request rate to PostStorage. At $t = 8$min, Midas's profiling reveals that PostStorage is no longer frequently accessed and therefore has relatively low cache utility, so Midas reactively scales down PostStorage's soft memory. At $t = 20$min, SocialNet reaches a new equilibrium, where UserTimeline consumes most of the soft memory budget and PostStorage only keeps a small portion of soft memory. Cliffhanger, in contrast, only profiles the cache hit rate of each microservice regardless of their cache access rate and performance sensitivity. Therefore, it continuously grants soft memory to PostStorage, resulting in overprovisioning soft memory to SocialNet.

D Soft Pointer Dereference Cost

In this section, we reported the detailed results of soft pointer dereference cost when reading and writing large objects in various sizes and compared them with the cost of dereferencing an ordinary C++ `unique_ptr`. Similar to Table 2, we measured the P90 latency and throughput of accessing large objects (Figure 16) in various sizes.

As shown in Figure 16, reading a large object whose size is smaller than 512 KiB with Midas soft pointer has similar latency and throughput compared to dereferencing a C++ `unique_ptr`, although dereferencing a soft pointer incurs an additional memory copy. This is because the object and its

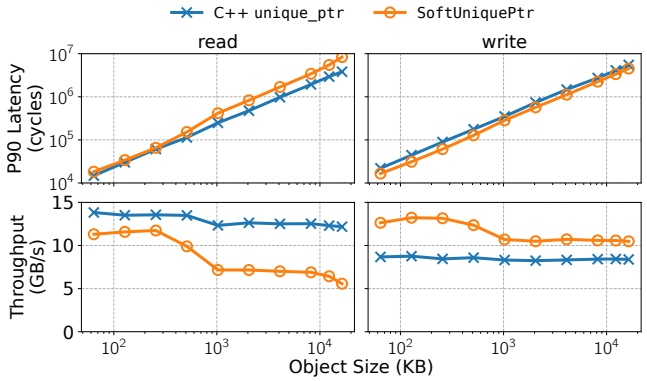


Figure 16: Midas's soft pointer achieves similar performance compared to C++'s ordinary smart pointer when objects can fit into CPU L2 cache, and it only adds moderate dereferencing cost otherwise.

copy can both fit into the CPU L2 cache and hence the second copy is fast. For all object sizes, soft pointer offers lower write latency and higher write throughput than `unique_ptr` thanks to Midas's optimized memory copy implementation.