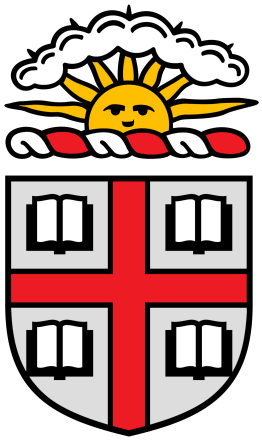
Hypno-Go: High-Performance Soft Memory for Golang Applications

Katie Li

Advisor: Malte Schwarzkopf

Reader: Akshay Narayan



Department of Computer Science

Brown University

Providence, RI

April 2026

Contents

Acknowledgments

Thank you:

To my advisor, Malte, who helped me develop a love for computer systems and believed in my ability to do research long before I believed in it myself.

To Howie, who patiently answered every question, humored every idea, and taught me a lot in our debugging sessions together.

To Nick, for being such a dedicated mentor and teacher both for this project and in many of my CS classes at Brown.

To Akshay, for being my reader and raising incisive feedback and questions about this work and about Hypno as a whole.

To Megan and Shirley, who provided me with guidance and mentorship when I first joined the soft memory project.

To the ETOS group, for being such a supportive and fun community.

To my family and friends, for their unwavering support throughout the years.

Abstract

Modern memory-intensive applications increasingly rely on large in-memory caches for performance purposes. At the same time, physical memory remains a rigid and overprovisioned resource. Golang application developers often statically partition memory between application heaps and manually managed caches, leading to significant underutilization and increased risk of out-of-memory failures.

Prior work introduces soft memory, a software-level abstraction on top of DRAM that, upon memory pressure, allows for memory to be safely and transparently revoked for reallocation elsewhere. This thesis presents Hypno-Go, a system that solves the underutilization caused by static partitioning by integrating Hypno, and existing soft memory system, into Go applications. Hypno-Go enables cache memory to be transparently reclaimed under memory pressure and reconstructed on demand via application-defined recomputation logic.

Our design allows for Golang applications with manually managed caches to avoid the memory underutilization inherent to static partitioning. We demonstrate that soft memory systems can be effectively integrated into garbage-collected languages without modifying the runtime, enabling dynamic cache resizing and improved memory utilization while maintaining performance and strong safety guarantees.

1 Introduction

Memory is in high demand; applications ranging from ML training to key-value caches see improved performance when these applications allocate more memory ([5, 6]). However, memory is also an inflexible resource; once allocated, it cannot be used elsewhere until explicitly freed. As a result, developers often overprovision the amount of memory available to applications to avoid OOM terminations ([10]). While this prevents applications from crashing, it also creates significant waste in large-scale settings.

1.1 Soft Memory

Prior work introduces soft memory—memory that can be transparently revoked when machines face memory pressure ([3, 8]). At allocation time, applications can allocate non-critical memory—i.e., memory that is useful but not strictly necessary for the application to run—in soft memory, indicating to the soft memory system that the memory can be revoked if needed for use elsewhere.

Memory that is critical to application logic will still be allocated in traditional memory. Likewise, data that is critical to application logic still requires more persistent storage, e.g. on disk. For example, in a key-value store application, specific key-value pairs that are frequently accessed could be cached in soft memory, while application logic that stores and fulfills requests would still use up traditional memory. A copy of the key-value pairs might persist in a database so that the application can re-access the data after the soft memory copy has been reclaimed.

This provides several benefits: first, it prevents out-of-memory terminations because soft memory can be revoked to keep total memory usage beneath the machine's overall memory limit. This results in a second benefit: soft memory also reduces memory underutilization because applications are no longer at risk of out-of-memory terminations, meaning that they can more freely allocate memory and thereby boost application performance.

Similar solutions for preventing out-of-memory terminations involve transparently swapping memory to disk when memory is needed elsewhere. However, this swapping is transparent to the application, meaning that developers cannot control *which* data is swapped out. This makes soft memory a preferable solution to swapping because certain data—e.g., caches—lose their utility once no longer in memory. By giving the application control over which data is freed, applications using soft memory can improve performance by ensuring that cache data stays in memory.

1.2 Memory Underutilization in Golang Applications

Golang is a popular language for developers building performance-critical applications; it balances performance and ease of use by providing efficiency and lightweight concurrency while also offering convenient features like garbage collection. However, many developers writing Go applications that store large amounts of data in memory—e.g., for caching

purposes—have identified that allocating memory in the Go heap has cascading negative effects on performance ([2, 1]). For one, increasing the amount of data in the Go heap will reduce the frequency of garbage collection cycles, which often leads to out-of-memory terminations ([4]. Additionally, increased memory in the Go heap means that the mark phase of garbage collection will take far longer, which has especially large effects if gigabytes of cache data persist in memory across multiple garbage collection cycles. As a result, developers who wish to extend a system or application written in Go with caching today typically allocate their caches in manually-managed memory ([2, 1]).

To achieve this, Go developers are required to statically partition the amount of memory available to the system between the manually managed cache and the Go heap. This static partitioning creates a difficult tradeoff for developers: if they overprovision memory available to the cache, they risk an out-of-memory termination if the Go heap grows too large. If they are too conservative and overprovision memory available to the Go heap by alotting very little memory to the cache, then a substantial amount of memory can end up unused. Further, applications must provision memory to the Go heap according to the maximum amount of memory that the Go heap uses over the application's duration, even if the average amount of memory used by the Go heap is far less than this amount. To prevent crashes, applications must overprovision memory to the Go heap, resulting in significant memory underutilization in these applications.

1.3 Proposed Solution

To prevent underutilization in Go applications, the cache would ideally grow and shrink dynamically depending on the total amount of available memory on the machine during a given point in time. Because soft memory is revocable, soft memory is a good fit for this cache data. If the cache memory can be reclaimed when needed elsewhere—specifically, when it's needed by the Go heap—Go applications can avoid statically partitioning memory between the cache and the Go heap and instead allow the cache to shrink when Go heap usage increases. Similarly, when Go heap usage goes down, previously reclaimed cache data should be restored in soft memory. To achieve this, we integrate Hypno (further discussion in §2.3), an existing soft memory system designed for C and C++ applications, with Golang. As a result, we show that soft memory reclamation is not only useful for providing other processes with more memory, but also for creating more free space for use by the same process.

2 Background

2.1 Golang Garbage Collection

Go developers are currently able to influence the rate at which garbage collection occurs by modifying the GOGC environment variable, which is set by default to 100. Per the GOGC documentation, "a collection is triggered when the ratio of freshly allocated data to live data remaining after the previous collection reaches this percentage." Represented as an equation, a GC cycle is thus triggered when the following condition is met ([4]):

$$\frac{\text{newly allocated data since last cycle}}{\text{live data remaining at end of previous cycle}} > GOGC$$

When large amounts of the Go heap memory are occupied by a cache that does not get garbage collected for a long duration, the denominator in this equation (live data remaining at the end of the previous garbage collection cycle) is extremely large because cache data (often gigabytes in size) will remain in memory after each garbage collection cycle concludes. A large denominator means that the amount of newly allocated data must grow very large before garbage collection is triggered again. For example, if the size of the cache is 10 GiB, then an additional 10 GiB of memory needs to be allocated before garbage collection is triggered again (assuming the default GOGC value of 100). Developers of Go applications have identified that the infrequency of garbage collection cycles in applications with large caches often results in out-of-memory-terminations ([2, 1]). As a result, they today usually opt to allocate and manage large caches by calling into C memory allocators; this removes large amounts of memory from the Go heap and thus increases GC frequency, dramatically reducing the risk of running out of memory. While they could also increase GC cycles by modifying the GOGC environment variable, this route requires significant developer effort because choosing the optimal value for GOGC is a non-trivial task ([1]). However, using a C memory allocator to manage a cache in a Go application also presents problems of its own. Manually managing cache memory leads to statically partitioning available memory between the Go heap and the cache. This leads to memory underutilization, as shown in the next section.

2.2 Static Partitioning in Golang Applications

Because the Go runtime will trigger garbage collection infrequently when applications allocate large caches on the Go heap, developers often manually manage their cache memory ([2, 1]). This requires statically partitioning available memory by allotting a certain amount for the manually managed cache; remaining memory is available for the Go heap to use.

The amount of memory provisioned for the Go heap to use must always equal or exceed the maximum amount of memory used by the Go heap over the duration of the entire application. This is especially problematic when the amount of space used by the Go heap fluctuates over time.

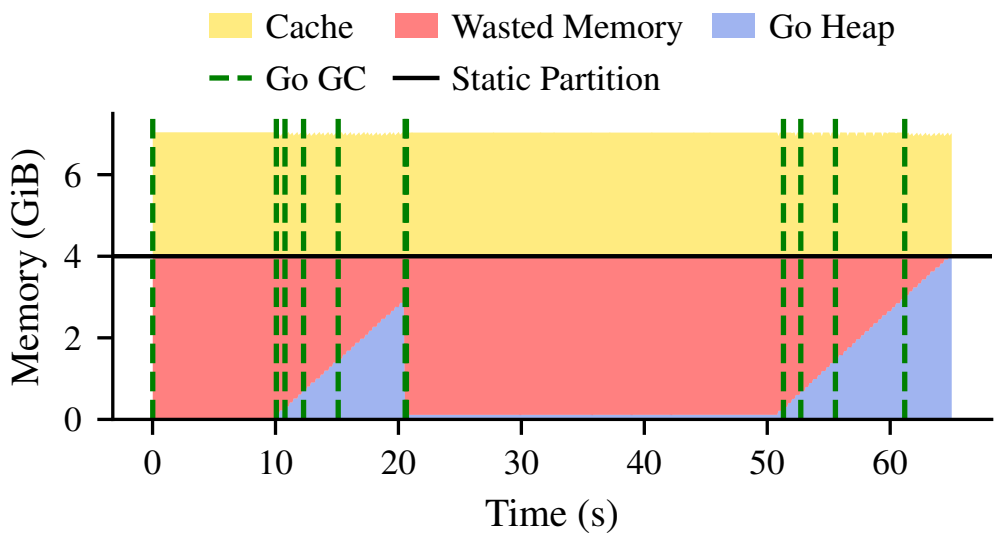


Figure 1: Example Go application using static partitioning. The red region indicates wasted memory, since it could otherwise be used for cache.

Figure 1 illustrates this problem. In this synthetic application, a 7 GiB memory limit is set for the entire application and 3 GiB are statically partitioned for the cache to use. The remaining 4 GiB are available for the Go heap to use. Given static partitioning, increasing the cache beyond 3 GiB would cause an out-of-memory termination because the Go heap at times requires a full 4 GiB of memory. However, there are many points in time where little to no space is occupied by the Go heap, particularly from 20 to 50 seconds. During this time, the red region—representing wasted memory—would be better utilized if it could be allocated to the cache. This motivates the need for a more flexible, dynamic approach to managing caches that allows for their memory footprint to grow and shrink according to the amount of available memory.

2.3 Soft Memory with Hypno

Hypno is a soft memory system that provides applications with transparent recomputation and reclamation. Additionally, Hypno leverages application semantics to inform which objects to reclaim ([7]. This allows for Hypno to prioritize objects that are less expensive to recompute for reclamation.

2.3.1 Transparent Recomputation

When applications access soft objects that have previously been reclaimed, Hypno transparently invokes recomputation of the missing data. It does so by invoking an application developer-defined recompute function that the application supplies to Hypno upon setup. Examples of recompute functions include reading from data from a form of persistant

storage—e.g., disk or a database—and restoring that data in soft memory for safe access by the application.

2.3.2 Semantic-Cooperative Reclamation

Hypno's primary contribution over existing soft memory systems is its use of application semantics in determining reclamation order. At allocation time, Hypno allows applications to specify a GroupID to indicate the priority level of a soft memory allocation. This priority level is designed to reflect how expensive—either in terms of time, compute resources, or monetary cost—it is to recompute a piece of data.

2.4 Other Related Work

2.4.1 Midas

Hypno builds on previous, single machine proposals for soft memory. Midas ([8]) uses cooperative garbage collection (GC) to coordinate soft memory across applications on a single machine, ranking objects by hotness and compacting them for reclamation. Unlike Hypno, Midas focuses on balancing memory usage across colocated applications in a single machine. Hypno focuses on improving memory utilization for a single application, locally or offloaded to remote memory on another machine.

2.4.2 AIFM and Far Memory

Soft memory is closely related to prior abstractions such as far memory, which extend the effective memory capacity of a machine by transparently offloading data to remote or secondary storage. Systems such as AIFM allow applications to access data that exceeds local DRAM capacity by transparently paging data across the network ([9]).

Despite these similarities, soft memory differs fundamentally in its semantics and intended use cases. Far memory systems aim to preserve data by relocating it, ensuring that all memory remains accessible, albeit with higher latency. In contrast, soft memory explicitly allows data to be discarded under memory pressure, with the expectation that it can be recomputed if needed. Soft memory is preferable to far memory when recomputation is cheaper than copying data from another machine. Additionally, far memory doesn't prevent out-of-memory terminations on the remote server, while soft memory does ([7]).

Hypno integrates with the AIFM runtime, managing the remote tier as soft memory. Under local memory pressure, the runtime evacuates local objects to far memory; under remote memory pressure, the runtime reclaims capacity from the soft memory pool—preventing OOM on both ends.

3 Design

3.1 Goals

In order to solve the problem of wasted memory caused by static partitioining, we explore how Hypno can be integrated with Golang. The result is a system called Hypno-Go.

Several goals informed its design:

1. Allow the cache to grow and shrink arbitrarily in order to free memory when it would better serve other processes and take up memory when there's idle space available.
2. Provide safe, ergonomic abstractions for Go developers to interact with Hypno-Go without requiring deep changes to application logic.
3. Minimize modifications required to existing Go applications, particularly those already using manually managed caches.
4. Use a standard version of Go runtime for ease of use.
5. No out-of-memory terminations or other failures.

These goals are motivated by the static memory underutilization problem illustrated in Figure 1. Hypno-Go addresses this by introducing a dynamic soft memory cache.

3.2 Integration with Hypno's Allocator

Soft memory is manually managed by Hypno's allocator rather than allocated directly on the Go heap. This is because Hypno-Go seeks to solve the problem of static partitioning common in Go applications today. Large amounts of memory on the Go heap have downstream effects on Go runtime behavior, namely garbage collection, as described in §2.1. Manually managing soft memory allows applications using Hypno-Go to avoid these downstream effects and decrease developer effort for developers who port existing Golang applications with manually managed caches to use Hypno-Go. This separation also allows Hypno-Go to preserve compatibility with a standard Go runtime while delegating all memory pressure decisions to Hypno. It also ensures that garbage collection behavior in Go remains unaffected, since soft memory allocations are entirely external to the Go heap and invisible to the Go runtime.

3.3 Preventing Races

A key benefit that Hypno-Go provides is ensuring that soft memory is safe for developers to use. Soft memory introduces new classes of concurrency bugs that do not already exist in Golang applications. This subsection explores types of potential races and Hypno-Go's approach to preventing them.

3.3.1 Races Between Application Goroutines

The first type of race condition that arises involves applications with multiple goroutines. If a piece of data has previously been reclaimed by Hypno-Go and multiple goroutines later

concurrently read the same piece of data, Hypno will invoke recomputation of the same data multiple times. This redundancy can be costly in practice. For example, recomputation may involve invoking an LLM (introducing monetary cost), querying large datasets, or performing expensive aggregation over external storage. Beyond performance concerns, correctness issues may also arise if recomputation is not idempotent, such as duplicate writes to logs or external systems.

While synchronization between application threads is generally incumbent on the application developer and necessary in many contexts beyond soft memory, this issue is Hypno-specific because concurrent reads in other contexts usually do not cause races. Because applications integrated with Hypno-Go have the unique potential to invoke recomputation upon reading data, Hypno-Go also presents a solution to this problem by synchronizing before data access and reclamation. Implementation details surrounding this synchronization are further discussed in §4.2.

3.3.2 Races Between Mutator and Reclaimer

Races are not only possible between application goroutines but also with external reclamation. Figure 2 illustrates a possible example: when the mutator accesses a piece of data in soft memory, the Hypno runtime may reclaim it while the application still holds a reference to it. Subsequent accesses to the data via this reference will cause segfaults.

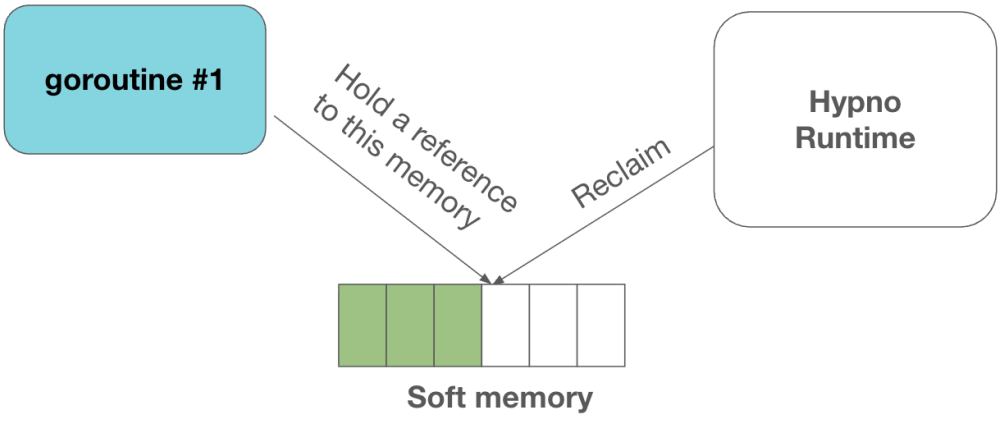


Figure 2: Visualization of a possible race condition between a mutator and a reclaimer. Soft memory is represented by green boxes. While the mutator holds a reference to a piece of data, the Hypno runtime reclaims it.

Hypno-Go eliminates this class of race by ensuring that dereferencing soft memory always produces a deep copy into Go-managed memory; the application never receives a direct pointer to soft memory. Since the application holds a reference to a traditionally-allocated copy, there is no longer a race with reclamation because reclamation will only

affect the underlying soft memory, which the application does not access. An example of this copying logic is shown in §3.4.

This design was chosen over alternative synchronization techniques like using dereference scopes, which are otherwise used by the canonical C++ Hypno implementation and its integration with far memory ([7]. The key difference from the canonical C++ Hypno implementation is that the Go abstraction avoids exposing raw pointers to the underlying soft memory; instead, it makes an explicit copy. This implementation intentionally trades performance for ease of integration, freeing developers from dereference scopes in the Go application logic.

3.4 Golang Ghost Pointers

Hypno-Go provides Go application developers with a ghost pointer library implemented entirely in Go, prioritizing both performance and usability. By avoiding a C++ interoperability layer and its associated C shim, the design eliminates foreign-function interface overhead and enables more efficient integration within Go applications.

Ghost pointers are 64-bit pointers into soft memory that allow developers to easily allocate and dereference soft memory. Ghost pointers abstract away recomputation and synchronization logic, allowing for applications to easily interact with soft memory via developer-friendly APIs. Figure 3 shows the layout of a ghost pointer.

Figure 3: Layout of a Golang ghost pointer. "L" represents the lock bit, "G" represents the ghost bit, and "V" represents the valid bit.

The `GroupID` stores information about reclamation prioritization. This information is provided by the application developer to indicate how expensive recomputation is; this is further described in §2.3.2.

The valid bit represents whether the ghost pointer points to a soft memory allocation. The lock bit is used to manage synchronization, as described in §4.2. Importantly, this design avoids introducing global synchronization overhead. Locks are fine-grained at the level of individual ghost pointers, which preserves concurrency across unrelated allocations while still guaranteeing safety for shared objects.

The ghost bit represents whether the data has been reclaimed and is no longer present in memory. Upon dereference, this determines whether data is recomputed. When the Hypno runtime finds data to reclaim, it will use the memory allocation's header to follow a reverse reference to its ghost pointer and set the ghost bit. When data is dereferenced and recomputed, the ghost pointer library implementation unsets the ghost bit to indicate that recomputation is no longer required.

Figure 6 shows pseudocode for `GhostPtr`'s `deref` function, which is a result of the synchronization and ghost pointer design decisions discussed throughout this section.

```
func (g *GhostPtr[T]) deref(size, args) T {
  lock object
  defer unlocking

  if ghost { // if data is no longer present
    soft_malloc(size)
    recompute(args)
    flip ghost bit // mark data as present
  }
  copy data // for synchronization
  return copy
}
```

Figure 4: Pseudocode for Hypno's `GhostPtr` dereference logic.

3.5 Hypno-Go Usage Example

Figure 5 shows an example of how an application developer can interact with the Hypno-Go library.

```
// create ghost pointer
// init func defines how to initialize memory
ghostPtr := NewGhostPtr[type](init func)
// NewGhostPtr allocates memory & synchronizes w/ reclaim

// dereference
data := ghostPtr.deref(recompute args)
```

Figure 5: Example of how an application developer can interact with soft memory using Hypno-Go. The developer allocates soft memory by constructing a new ghost pointer and accesses data by calling `GhostPtr`'s `deref` function.

The application allocates soft memory by constructing a new ghost pointer. `NewGhostPtr` abstracts away the logic of allocating memory and managing synchronization while data is being written to soft memory. When the application later needs to access the data, it calls the `deref` function on the ghost pointer and passes in arguments necessary for data

recomputation (to be used if the underlying soft memory has been reclaimed). This returns a copy of the data for the application to operate on.

In order to specify what data should be written to the soft memory allocation, the developer provides an initialization function that returns the data to be written to soft memory as an argument to `NewGhostPtr`. In turn, `NewGhostPtr` can call this function to retrieve the data and initialize the soft memory accordingly. This design is necessary to prevent race conditions. If the application developer could invoke `soft_malloc` directly before constructing the `GhostPtr` object, the Hypno runtime could potentially reclaim the soft memory application before the `GhostPtr` is constructed. Since `GhostPtr`s manage synchronization internally, the memory thus cannot exist until the `GhostPtr` exists. Since `deref` returns a copy of the underlying data, developers cannot mutate what it returns. Hence, application developers specify how data should be initialized by passing a function argument when creating a new `GhostPtr`.

4 Implementation

Hypno-Go is implemented as a Go library that bridges Go application code with Hypno's native allocator. The implementation consists of three main components: a CGO-based interface to Hypno's allocator, a synchronization layer to prevent races, and the ghost pointer abstraction itself.

4.1 Interaction with Hypno Allocator via CGO

Hypno-Go interacts with Hypno's underlying allocator through a minimal CGO interface that exposes only soft memory allocation and deallocation primitives. CGO allows Go packages to call C functions and use C types directly within Go source files by embedding C declarations in a preamble comment block. Hypno-Go uses this mechanism to expose `soft_malloc` and `soft_free` to Hypno-Go library code. This design ensures that the Go runtime remains unmodified while still enabling controlled interaction with Hypno-managed memory. The CGO boundary is intentionally narrow: Hypno-Go does not embed any allocator logic in Go, nor does it attempt to replicate memory management semantics on the Go heap.

4.2 Synchronization

Synchronization in Hypno-Go is designed around preventing unsafe concurrent access between mutator goroutines and the Hypno reclaimer. Because soft memory may be reclaimed asynchronously, all interactions with ghost pointers must be coordinated to ensure correctness.

At the core of this mechanism is a lightweight locking protocol embedded in the ghost pointer metadata. Each ghost pointer contains a lock bit (see §3.4) that is used to synchronize access during dereference and recomputation. Before any dereference or allocation, the runtime acquires this lock to ensure that reclamation cannot occur concurrently. The lock bit in each ghost pointer serves as the synchronization primitive. Before performing either dereference or recomputation, a goroutine attempts to acquire the lock using an atomic CAS operation that transitions the bit from "unlocked" to "locked". If the CAS fails, the goroutine retries in a tight loop, effectively spinning until the lock becomes available.

4.3 Golang Ghost Pointer Implementation

The ghost pointer abstraction is implemented entirely in Go, with no reliance on a C++ shim layer. As described in §3.4, each `GhostPtr` instance stores a 64-bit encoded address into soft memory along with metadata bits used for validity, reclamation state, and locking.

Allocation is performed through the `NewGhostPtr` constructor, which bridges Go and Hypno's allocator via CGO. The constructor is responsible for allocating soft memory and initializing it under synchronization to prevent concurrent reclamation from observing partially initialized state.

```
func NewGhostPtr[T](init func() T) *GhostPtr[T] {
    lock() // synchronize with reclaim
    defer unlock()

    size := unsafe.Sizeof(*new(T))
    raw := soft_malloc(C.size_t(size))
    ptr := (*T)(raw)
    *ptr = init()
    g := &GhostPtr[T]{}
    g.init(uint64(uintptr(addr)))
    return g
}
```

Figure 6: Implementation of `NewGhostPtr`. This function invokes `init`, which is provided by the application, to retrieve the object that should be allocated in soft memory.

5 Evaluation

We now evaluate Hypno-Go's impact on memory utilization using a synthetic Golang application with an integrated cache, similar to the application in Figure 1. However, rather than statically partitioning between the cache and the Go heap, we allocate the cache in soft memory using Hypno-Go.

5.1 Evaluation Questions

Our evaluation seeks to answer the following questions:

1. Can Hypno-Go help increase utilization of otherwise idle memory?
2. What is Hypno-Go's end-to-end performance under memory pressure?
3. What is the impact of Hypno-Go's design decisions on its performance?

5.2 Experiment Setup

The application has a total memory budget of 7 GiB, but uses a dynamically-sized soft memory cache rather than a static memory partition. The cache can grow up to the full memory budget and consists of an array of ghost pointers that each point to a 2 KiB-sized object. The application has two parallel goroutines: one that continuously accesses random elements in the cache according to a Zipf distribution (s=1.1) and a second goroutine that makes allocations on the Go heap. Hypno reclaims soft memory when the application's total memory footprint approaches the 7 GiB limit. The application registers a recompute function with Hypno that seeks to a random location in a 3 GiB file and reads 2 KiB from the file into a new soft memory allocation. We configure the Go runtime with its default settings. We measure the throughput achieved by cache accesses and amount of wasted memory. A good result for Hypno would show consistently high throughput and little wasted memory.

5.3 Evaluation Results

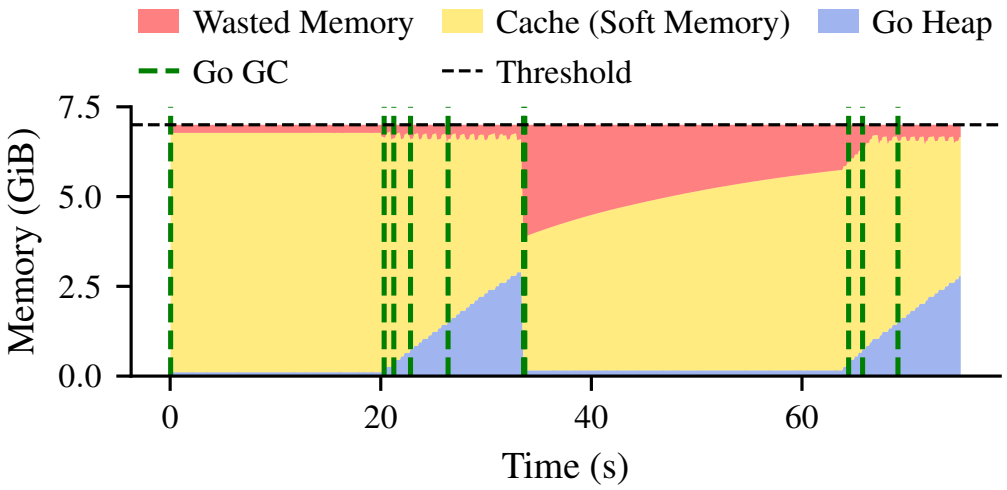


Figure 7: Golang synthetic application memory utilization. Hypno-Go reclaims from soft memory cache to keep total usage below the 7 GiB threshold. Shortly after 20 seconds, the Go heap is released back to the OS. Compared to Figure 1, Hypno-Go reduces the "wasted memory" region.

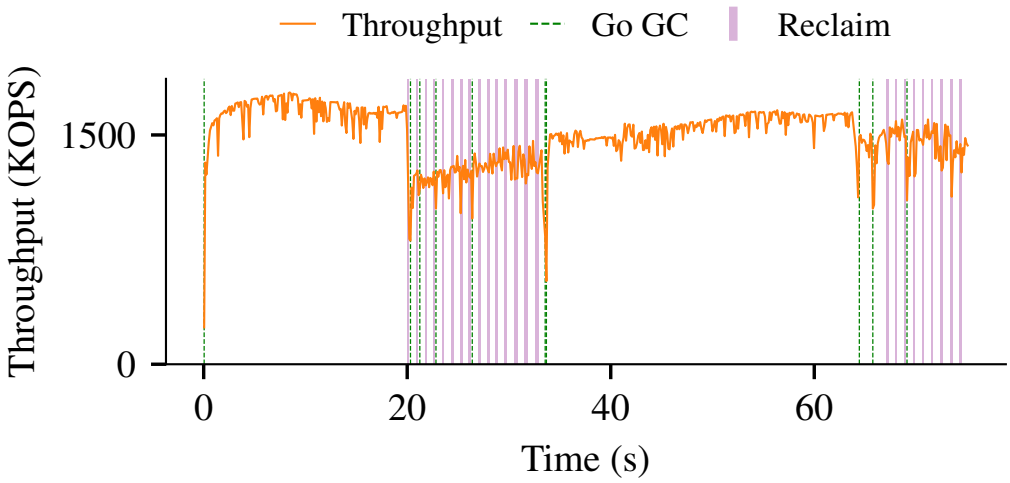


Figure 8: Golang synthetic application throughput. Throughput remains high throughout, with ≈20% reductions during reclamation phases as the reclaimer and application mutator threads compete for access to soft memory objects.

Figures 7 and 8 show the result. In the first 20 seconds, the Go heap doesn't grow and the application only accesses cache elements to establish a baseline for throughput. After

these 20 seconds, the application continuously allocates memory on the Go heap until the 35-second mark. During this phase, Hypno continuously reclaims memory from the cache to keep the application under the 7 GiB limit. At 35 seconds, 2.5 GiB of Go heap memory become garbage and the GC frees them. The cache now has room to regrow as cache accesses dereference ghost pointers, causing recomputations. At 65 seconds, growth on the Go heap resumes and Hypno reclaims cache memory again. The region of wasted memory—memory that is free for use but is not utilized by the application—is smaller in this experiment than in motivation figure. Throughput remains high over throughout the experiment, despite the cache growing and shrinking. Phases that involve reclamation see a $\approx$20% reduction in throughput because the reclaimer and the application's mutator threads compete for access to soft memory objects. Hence, Hypno allows Go applications to avoid the underutilization caused by the static partitioning common today ([2, 1]), as it allows the cache to dynamically grow and shrink and use spare memory.

6 Discussion & Future Work

Because Hypno-Go's design keeps the cache in manually-managed memory rather than on the Go heap, it introduces tradeoffs between soft memory reclamation and garbage collection. This separation results in two independent systems making memory pressure decisions without global visibility. As a result, Hypno might reclaim cache memory at times when a large amount of garbage is present on the Go heap. In these cases, it's always preferable to free memory by freeing garbage, which is semantically useless to the application, rather than soft cache memory, which can improve application throughput.

A key direction for future work is to more tightly integrate Hypno's reclamation decisions with runtime information exposed by the Go garbage collector. In particular, the Go runtime already provides visibility into garbage collection cycles, including when GC was last triggered, how long it ran, and how much memory was reclaimed ([4]).

This information opens the possibility of designing a hybrid decision-making system that dynamically chooses between relying on GC versus triggering Hypno reclamation. Intuitively, garbage collection is strictly better at identifying truly unreachable memory. However, GC is constrained by its scope: it cannot reclaim soft memory managed externally by Hypno.

Conversely, Hypno's reclamation operates at the level of application-defined soft memory, allowing it to reclaim large amounts of memory that the GC cannot observe. However, this reclamation is semantic rather than reachability-based, meaning that soft memory reclamation still comes with a cost if the data is later accessed.

A promising research direction is to construct a cost-aware heuristic that balances these two approaches to freeing memory. Such a heuristic could consider:

- The amount of memory freed in the last GC cycle
- The frequency and duration of recent GC cycles
- Current amount of system memory pressure
- Historical recomputation cost for soft memory objects (via `GroupID` or profiling)

Using these signals, Hypno-Go could estimate whether additional memory pressure should be handled by waiting for the next GC cycle or by proactively reclaiming soft memory. For example, if the heap size has grown substantially since the most recent garbage collection cycle and the system is under moderate pressure, aggressive Hypno reclamation may be unnecessary. Conversely, if garbage collection is frequent but yields low memory recovery, this may indicate that most reclaimable memory resides in soft memory regions, motivating more aggressive Hypno-driven reclamation.

More broadly, this direction points toward a unified memory management policy that treats garbage collection and soft-memory reclamation as cooperative rather than independent mechanisms. Designing such a policy raises several open questions, including how to model

recomputation cost accurately and how to ensure that the added overhead of these calculations does not offset its benefits.

Ultimately, we believe that integrating runtime garbage collection signals into soft memory management has the potential to improve memory utilization while preserving the abstraction benefits of Hypno-Go.

7 Conclusion

This paper presented Hypno-Go, a system that integrates soft memory into Go to address the problem of static memory partitioning between application heaps and manually managed caches while maintaining high throughput. In traditional Go applications, developers must statically partition memory allocation between the garbage-collected heap and external caches, often overprovisioning memory to avoid out-of-memory failures. This results in substantial underutilization, particularly in workloads with large but dynamically varying Go heap usage.

Hypno-Go addresses this limitation by introducing a dynamically managed soft memory layer that allows cache data to be transparently reclaimed under memory pressure and recomputed when needed. By keeping cache memory off of the Go heap in the same way that popular Go applications do, Hypno-Go avoids adverse interactions with garbage collection while also enabling applications to use otherwise idle system memory.

We further introduced a Golang implementation of ghost pointers as a safe and ergonomic abstraction for interacting with soft memory in Go applications. Ghost pointers encapsulate allocation, synchronization, and recomputation logic while ensuring that applications never directly access reclaimed memory. Instead, dereferencing always produces a safe copy, eliminating use-after-free errors and simplifying concurrency semantics at the application level.

Future work includes developing tighter integration between garbage collection telemetry and soft memory reclamation policies.

References

[1] Cockroach Labs, Inc. Pebble key-value store: Memory management.

[2] Dgraph Labs. Dgraph blog: Manual memory management in go using jemalloc.

[3] Megan Frisella, Shirley Loayza Sanchez, and Malte Schwarzkopf. Towards increased datacenter efficiency with soft memory. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS)*, page 127–134, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Go Authors. runtime package documentation. https://pkg.go.dev/runtime, 2026. Accessed: 2026-04-29.

[5] 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, pages 33–51, 2022.

[6] 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)*, pages 283–296, Santa Clara, California, USA, February 2020.

[7] Nathan Kim Megan Frisella Shirley Loayza Sanchez Nick DeMarinnis Po Hao Chen, Katie Li and Malte Schwarzkopf. Hypno: High-performance soft memory via semantic-cooperative reclamation. In *Under submission*, 2026.

[8] Yifan Qiao, Zhenyuan Ruan, Haoran Ma, Adam Belay, Miryung Kim, and Harry Xu. Harvesting idle memory for application-managed soft state with midas. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1247–1265, Santa Clara, California, USA, April 2024.

[9] 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)*, pages 315–332, November 2020.

[10] 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, April 2020.