Troubridge: Learning-Based Access-Frequency Prediction for Memory Allocation

Bokai Bi
Brown University
Providence, Rhode Island

Zi Qiao
Brown University
Providence, Rhode Island

Patrick Rui De Jing
Brown University
Providence, Rhode Island

Abstract

The continued scaling of cloud workloads and warehouse-scale systems increasingly depend on in-memory computation. Coupled with the recent emergence of LLM-serving workloads that benefit from model caching, DRAM demands have been soaring along with prices. At the same time, because of the end of Moore's law for memory, the cost of DRAM is no longer getting cheaper. These create challenges to the total cost of ownership to warehouse-scale computers.

Traditional memory allocators treat all allocations identically, regardless of how often those objects are accessed by the application. As a result, rarely accessed (*cold*) data and frequently accessed (*hot*) data are co-located, causing a variety of issues. First, the mixing of objects (data in memory) with different access frequencies leads to inefficient usage of the TLB cache, since some areas of an in-cache span will contain cold objects that do not benefit from being cached. This is especially problematic in systems employing hugepages, since the greater size allows for more heterogeneous objects to potentially be packed together. Secondly, this mixing also prevents systems from fully exploiting "far memory" (a memory tier slower but cheaper than DRAM), which depends on identifying pages that are infrequently accessed as a whole.

We propose Troubridge, a new memory allocation framework that predicts object access hotness at allocation time, using techniques from natural language processing (NLP) over calling contexts. We record the access frequencies of allocated memory at training time to enable dynamic predictions of memory hotness at runtime. By organizing the heap by "hotness class" rather than merely size class (as in traditional allocators like TCMalloc), Troubridge improves cache locality for hot objects, and lays the foundation for integration with far-memory systems and huge-page aware allocators.

Our work demonstrates:

(1) It is feasible to build a context-sensitive hotness predictor using a GRU over stack-trace data, generalizing to previously unseen calling contexts.
(2) A heap structure and allocator design that uses predicted hotness classes to assign allocations to segregated hugepage pools.
(3) The conceptual and implementation blueprint for far-memory cooperation in memory allocators.

Though we do not yet provide full end-to-end performance results, we carefully describe design tradeoffs, implementation challenges, and preliminary evaluation metrics. We hope this work will seed further research into learning-based memory allocators.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/20XX/XX
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

• **Computing methodologies** → **Supervised learning**; • **Software and its engineering** → **Allocation / deallocation strategies**.

Keywords

Memory Management, Machine Learning, Lifetime Prediction, Profile-Guided Optimization, LSTMs

ACM Reference Format:
Bokai Bi, Zi Qiao, and Patrick Rui De Jing. 20XX. Troubridge: Learning-Based Access-Frequency Prediction for Memory Allocation. In *Proceedings of Conference (Conference acronym 'XX)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern warehouse-scale computers (WSCs) increasingly operate under stringent memory pressure. Across large-scale fleet deployments, DRAM has emerged as a dominant determinant of both performance and cost: it is expensive to provision, difficult to oversubscribe, and fundamentally inelastic. Empirical studies from hyperscaler operators consistently report that memory utilization, rather than CPU utilization, is the limiting factor in job placement and availability. Studies from Google and Alibaba report an average memory utilization across servers of around 60%, with substantial variance across machines, compared to an average of 40% CPU utilization. [3] Once a server runs out of available DRAM, applications must be terminated. At Google, over 790,000 jobs experienced at least one instance killed due to memory pressure in a month [6].

At the same time, industry measurements show that a significant fraction of DRAM—often exceeding 30%—contains *cold* data that has not been accessed for minutes, yet remains unidentifiable to existing systems at sufficient granularity or timeliness to enable reclamation [2]. This mismatch between the *cost* and *utility* of allocated memory underscores a growing need for mechanisms that reason about memory access patterns at the level of individual allocations.

A promising approach to improving memory efficiency is the introduction of far memory, a slower but cheaper memory tier that sits between DRAM and flash storage. However, far-memory mechanisms rely on identifying infrequently accessed regions of

memory that can be safely migrated without imposing unacceptable latency on applications. Similarly, hugepage-aware memory allocators would benefit from identifying hot allocations from cold data, where hugepage backing provides little value and may instead increase fragmentation. In both settings, the key question is: how can a system determine whether a newly allocated object is likely to be hot or cold, early enough to influence the allocator's placement decisions?

Existing systems rely on reactive mechanisms such as kernel-level sampling, page-fault monitoring, or periodic background scans. In addition to being limited to retroactive measurements, these approaches are also coarse-grained, introduce runtime overhead, and lack allocation-site context. Reactive profiling captures where memory is accessed, but not why that data was allocated, nor the semantics of the calling context that produced it. Moreover, continuous profiling is impractical at scale: the overhead of frequent sampling often exceeds that of the allocations themselves. As a result, today's allocators make placement decisions without visibility into the object's future access behavior, severely limiting the effectiveness of both far-memory systems and hugepage optimizations. As moving memory post-allocation is generally expensive, it follows that a more globally efficient memory placement strategy must rely on information that's available at allocation time.

In this work, we investigate whether modern machine learning techniques can bridge the aforementioned gap. Program stack traces contain rich semantic information about the allocation context. Similar to natural language, stack frames encode structural and functional relationship such as control-flow patterns, library usage, and call hierarchy. Recent work has demonstrated that neural models trained on call-site information can predict dynamic properties of heap allocations, such as lifetime and reference locality. Inspired by these findings, we hypothesize that a model trained on calling-context sequences can predict access frequency at allocation time, thereby enabling proactive memory placement.

We present Troubridge, a prototype for a hotness-aware memory management system that integrates a learned prediction model with a redesigned allocator. Troubridge introduces two key components:

(1) **A GRU-based hotness prediction model** that consumes allocation-site stack traces and an object size to predicts an allocation's expected access frequency class. The model generalizes across workloads and calling contexts, enabling accurate predictions even for sites not encountered during training.
(2) **A hotness-class-aware allocator** that organizes memory not by size class alone (as in conventional allocators) but by predicted hotness class. Troubridge segregates pages and hugepages by hotness, co-locating similarly accessed objects, reducing fragmentation of hot regions, and enabling more aggressive use of far memory for cold regions.

At allocation time, Troubridge assigns each object to a hotness class and places it within a corresponding block of a hugepage, ensuring that hot objects remain tightly clustered. While our prototype focuses on inference-time integration and design feasibility, our methodology is general and is compatible with both future migration-based cold-memory systems and user-level hugepage managers.

Although our present work does not include an end-to-end evaluation on production workloads, it contributes a new design point in the allocator space. By enabling proactive hotness classification, Troubridge opens the door to more efficient use of DRAM, improved TLB coverage for hot objects, and higher overall memory-system efficiency in warehouse-scale deployments.

2 Motivation

The datacenter tax, which includes universal overheads from memory allocation, data serialization, compression, copying, and other such operations, represents a significant performance cost in warehouse-scale computing (WSC). Instead of optimizing on a per-application basis, it is often more effective to reduce specific aspects of the datacenter tax globally, yielding fleet-wide performance improvements. Among these, memory has been found to be a particularly critical part of datacenter tax [5].

To alleviate DRAM pressure, recent work has proposed the introduction of a far memory tier: a slower but more cost-effective memory pool such as compressed memory, remote memory, or locally attached non-volatile memory. Far memory systems aim to migrate cold pages into this tier while retaining hot data in DRAM. The potential savings are substantial: by offloading cold data, datacenters can reduce DRAM provisioning, increase job packing density, and improve resilience to memory spikes [2, 5].

However, existing far-memory designs suffer from two inherent limitations:

(1) **Reactive coldness detection.** Systems typically rely on periodic kernel-level sampling or page-fault tracking to infer whether a page is cold. This inference is lagging rather than predictive: coldness is only recognized after the data has remained inactive for a long period.
(2) **Page-level granularity.** Coldness decisions are made on coarse OS pages, even though the true semantic boundaries of objects exist at the allocation level.

These limitations result in delayed migration, unnecessary movement of mixed hot and cold pages, and a mismatch between allocator semantics and far-memory decisions. If instead memory allocators can predict whether an allocation will become cold at allocation time, they can cluster rarely accessed allocations together, improving the effectiveness of cold memory systems. Additionally, more overhead can be afforded to these decisions since they cannot occur very often.

A parallel line of work investigates techniques for improving TLB coverage and memory translation performance by placing hot data on hugepages [1]. User-level hugepage-aware allocators such as `Temeraire` have the goal of improving TLB hit rates and reducing kernel involvement. However, they face a difficult trade-off. Allocating many hugepages increases performance, but also increases fragmentation. Splitting hugepages allows fine-grained reclamation, but undermines the core benefit of large translation units.

The effectiveness of hugepage-aware allocation is directly tied to whether the allocator can distinguish hot data from cold data. Existing allocators, however, lack access-frequency semantics and

thus rely on heuristics that often misclassify allocations or treat all objects uniformly.

Both far-memory systems and hugepage-aware allocators would benefit from understanding the *hotness* of memory objects. Unfortunately, modern allocators operate in near blindness: the calling context of an allocation, which contains rich semantic information about data structure use and program intent, is discarded as soon as the allocation is made. As a result:

(1) Memory managers cannot proactively segregate hot and cold objects.
(2) Reactive mechanisms introduce overhead and delay.
(3) Page-level decisions obscure object-level semantics.

Solving this problem fundamentally depends on reasoning about object hotness and grouping objects with similar hotness levels together. Our system needed to address the following challenges:

Hotness accuracy and coverage. Applications may exhibit large dynamic variability. Training data may not cover all possible calling contexts, and calling contexts encountered in deployment may have never appeared during profiling. Classical heuristics, such as object size or allocation-site frequency, offer poor predictive power across diverse workloads.

Overheads. While sampling-based profiling can reveal access frequency, continuous monitoring introduces overhead that often exceeds the cost of memory allocation itself.

Our key insight is that stack traces capture essential program semantics in a structured, sequential form that can be interpreted by modern machine-learning models. Just as natural-language models learn contextual associations between words, a neural model trained on calling contexts can learn correlations between program structure and allocation hotness. This observation motivates our learning-based design.

We build a GRU model that takes as input an allocation's calling context and predicts its hotness class. The model generalizes across unseen stack traces, enabling allocation-time prediction at high coverage without continuous profiling.

Finally, we integrate these predictions into an allocator, Troubridge, that clusters objects by hotness level rather than size alone. This design enables more efficient hugepage utilization, simplifies far-memory migration, and creates a clear semantic boundary between hot and cold data.

3 Related Work

3.1 Hugepage-Aware Memory Allocators

One approach to reducing the warehouse tax is to improve memory allocation decisions. The classical approach is to reduce the amount of cycles spent in allocator code; i.e. to make the memory allocator more efficient. But memory allocators can actually be *efficient* if we make up for it by reducing CPU time in application code.

One approach is hugepage-aware memory allocation. Pages are 4 KB, but hugepages are 2 MB. The increased size of the hugepages means that the same number of TLB entries can map a larger range of memory, reducing TLB misses. Hugepages also require less pagetables to traverse, reducing stall time even in the event of a TLB miss.

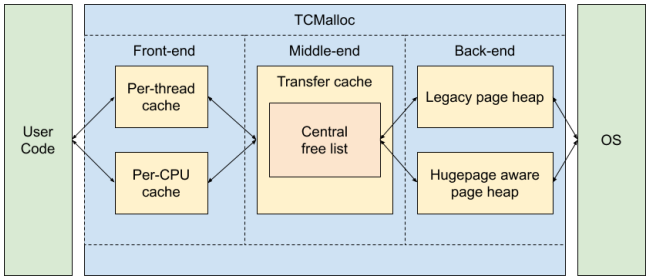


Figure 1: TCMalloc internal structure

3.2 Far Memory Systems

Another approach to reducing the warehouse tax is to leverate far memory, a slower but cheaper tier of memory. Far memory should have a lower cost per GB than DRAM and a higher performance than Flash. Examples of far memory include remote DRAM, NVRAM, or software-defined far memory.

Far-memory systems place infrequently accessed data in far memory, allowing the machine to perform the same jobs with less DRAM. Such systems can migrate around 20-30% of cold data, leading to 4-5% saves in memory.

One weakness of these systems is that they often rely on page-level coldness classification, often based on runtime sampling or usage statistics. These reactive approaches incur overhead, may misclassify pages, and lack per-allocation granularity. Memory allocators can improve far-memory cooperation by proactively placing cold objects together in memory.

3.3 Learning-Based Memory Allocation

Our learning-based approach to memory allocation follows the spirit of other approaches. Llama uses a LSTM language model to predict object lifetimes in order to reduce hugepage fragmentation [4]. Our approach uses a GRU language model to predict access frequencies to improve far memory cooperation and hugepage cache hits. A commonality between the approaches is the idea that stack traces contain enough semantic information to make judgments about the access patterns of memory objects.

4 Overview of TCMalloc

TCMalloc is Google's implementation of C's `malloc()` and C++'s operator new used for memory allocation within C and C++ codebases. Our design builds directly on the structure of TCMalloc.

In TCMalloc, memory is partitioned into *spans* and aligned to page size. Some spans only contain one large object. Other spans contain multiple objects of the same size. Allocation requests are rounded to one of 100 sizeclasses.

TCMalloc uses a stacked cache design. TCMalloc will first attempt to serve allocations from a *local cache*, which are organized per-hyperthread. When the local cache has no more capacity for the sizeclass, requests are routed to a single *central cache* for the sizeclass. There are two components to the central cache: (1) the *transfer cache* contains a flat array of objects, and (2) the *central freelist* contains every span assigned to that sizeclass. When all objects from a span has been returned to the central freelist, that

span is returned to the pageheap. The pageheap tracks spans, splits and merges them, and calls `mmap` and `release` whenever it needs to return memory back to the operating system.

Temeraire is an enhancement on TCMalloc that makes the pageheap hugepage-aware. It consists of the following components:

(1) The `HugeAllocator` requests hugepages and keeps them unbacked.
(2) The `HugeCache` stores backed, fully-empty hugepages. It handles large requests (> 1 GiB) and allocations that would fit within exact multiples of hugepages.
(3) The `HugeFiller` stores partially-filled hugepages. It handles small to medium requests (< 2 MiB). The algorithm for allocating memory from `HugeFiller` will choose the hugepage with the lowest sufficient free range L and the highest number of allocations A.
(4) The `HugeRegion` handles medium-sized requests that do not fit well within hugepage boundaries. Unlike the `HugeFiller`, it performs a best-fit algorithm.

5 Access Frequency Aware Allocator Design

Troubridge primarily modifies the allocation policy of the `HugeFiller`. While Temeraire selects a hugepage by choosing, among all partially filled hugepages, the one with (1) the smallest sufficient free range L and (2) the highest number of extant allocations A, its policy is oblivious to object access patterns. As a result, hot and cold allocations may be interleaved on the same hugepage, diminishing the effectiveness of both far-memory systems and hugepage-locality optimizations.

To support access-frequency-aware placement, Troubridge associates every allocation with a predicted *hotness class* (HC) and extends this classification to hugepages. Let $\mathcal{H} = \{\text{Cold}, \text{Warm}, \text{Hot}\}$ denote a small, fixed lattice of hotness classes. Each allocation receives a predicted hotness class $h \in \mathcal{H}$ from the learned model, and each hugepage P is annotated with a page-level class $H_{page}(P)$.

Troubridge enforces the following invariant:

$$\forall P,\ \forall o \in P: \qquad H_{page}(P) = h(o),$$

In other words, every hugepage contains only allocations whose predicted hotness exactly matches the page's assigned hotness class. This invariant ensures that cold pages are not polluted by hot allocations and, critically, that cold allocations never occupy hot pages.

To maintain this invariant, Troubridge partitions the `HugeFiller` into independent per-class pools:

$$\texttt{Filler[h]} \quad \text{for each } h \in \mathcal{H},$$

where `Filler[h]` contains all partially filled hugepages whose page-level class H_{page} is exactly h.

Given an allocation request of size s and predicted hotness h, the modified `HugeFiller` performs the following selection:

(1) **Class-restricted search.** Search only within `Filler[`h`]` for hugepages that contain a free range sufficient for s.
(2) **Temeraire-style ranking.** Among candidate hugepages in `Filler[`h`]`, select the hugepage with the smallest sufficient free range L. Break ties by choosing the page with the highest allocation count A. This preserves Temeraire's low-fragmentation behavior within each hotness class.
(3) **Hugepage creation.** If no hugepage in `Filler[`h`]` has sufficient space, a new hugepage is requested from the underlying allocator. Its page-level hotness class is initialized to h, inserted into `Filler[`h`]`, and the allocation is served from this new page.

Although the present prototype does not implement full dynamic reclassification, Troubridge is designed to support it. Each hugepage maintains lightweight access-frequency statistics that can be sampled periodically. When observed behavior diverges substantially from its predicted class, the hugepage may be reclassified. In the static policy presented here, reclassification is conservative: once a page is assigned to class h, it remains in `Filler[`h`]`. A more flexible policy, such as monotonic upgrades or migration-based demotion, is a promising direction for future work.

6 Hotness Prediction Model

We connect to an application and collect lifetimes for a sample of all allocations that occur during this period. Because sampling may not observe all allocation calling contexts, we cannot simply use a lookup table. Our solution is to use a machine-learning model to map from calling context to hotness, while generalizing to previously unseen contexts. The predictions allow Troubridge to organize the heap based on hotness levels to increase cache locality.

6.1 Data Collection

We sample all memory allocations made during a given time period. Each sample includes stack trace, object size, and address at allocation time.

We integrated this approach into TCMalloc. Its existing heap profiling mechanisms identifies objects by producing a list of sampled objects at the end of the applications execution. However, it does not profile short-lived objects that are not alive at the end of the program. We therefore extended the profiling mechanism with custom hook called `AllocationSiteRecorder` which records unique allocation sites. For each site, the stack trace, total allocation count, total bytes allocated, and min/max allocation sizes are recorded. A hash map keyed by stack trace is used for deduplication. The hook incurs virtually no overhead when it is disabled. When the program first starts, a separate thread is spawned that periodically checks all currently active allocation sites for whether their latest allocation was accessed since the last period. An access percentage is produced accordingly. We hooked TCMalloc's free functionality to make sure freed memory is not counted in this process.

One challenge with creating the `AllocationSiteRecorder` is that it allocates a data structure to record the unique allocation sites *within* the TCMalloc heap. This causes a few subtle thread-safety bugs. After `RecordAllocation` locks the mutex, a modification to the data structure can trigger a resize that allocates more memory from the heap, which reenters the allocator and calls `RecordAllocation` again. We solved this creating reentry guards that skips recording when already inside `RecordAllocations`. A better solution would be to allocate the data structure outside of the TCMalloc heap entirely. At the end of the sampling period, we store the result into a protocol buffer.

6.2 Data Processing

The collected data is piped to disk for training purposes. In either case, we extract the following information from each allocation site: allocation size, total intervals checked, number of intervals with access, and the stack trace of the allocation site. Each site is delimited on a new line. The data is fed into a Huggingface Tokenizer for performant cross-platform compatibility between Python (training) and C++ (inference). We used a BertNormalizer and a BPE model to train the tokenizer. The resulting tokenizer has a vocabulary size of 5000 in addition to a [UNK] token for unknown sequences. We also made sure that common stack trace symbols with semantic meaning, including "::", "(", ")", and "," are represented as special symbols to allow for better extraction of meaning from the stack trace. Tokenized inputs are then converted to a Pytorch dataset for training and running purposes.

6.3 Model Architecture

We represent each allocation calling context as a sequence of symbol identifiers (the stack trace), similar to a sentence in natural-language processing. Since we aim to generalize to unseen calling contexts by exploiting semantic meanings in the stack trace, we first convert each token into an embedded space representation with 32 dimensions. This sequence is then fed into a Gated Recurrent Unit (GRU) network. The output of the GRU and the size of the allocation is then concatenated and fed to 2 linear layers with a hidden size of 64, connected by a ReLU activation function. The final output is processed by a Sigmoid layer to predict the category.

In order to prevent overfitting and ensure generalization, we reserve a held-out set of stack traces from a separate run or workload for test and validation as per usual for deep learning model training.

7 Implementation

We implemented a functional prototype of Troubridge by extending the hugepage-aware variant of TCMalloc, Temeraire. Our implementation consists of three major components: hotness prediction, stack-trace capture, and hotness-aware hugepage allocation. Together, these changes enable the allocator to maintain hotness-segregated hugepages and to route allocations based on predicted access frequency.

7.1 Collecting Calling Contexts and Access Frequency

Hotness prediction requires knowledge of the allocation-site context. Capturing stack traces for every object allocation would be prohibitively expensive. As a result, we capture stack traces *once per span*, relying on the span as the funding unit for small-object allocation. Although this approach may mix objects from different call sites within a span, it provides a reasonable tradeoff between granularity and overhead.

To support this, we extended TCMalloc internal data structure `SpanAllocInfo` with a field for the captured stack trace. We hooked `do_malloc_pages` in `tcmalloc.cc` to obtain the stack trace using `absl::GetStackTrace` and to pass it downstream as part of the span metadata. These changes provide the predictor with sufficient context to determine the appropriate hotness class for each allocation.

During data collection for training, in order to maintain a reasonable degree of performance to ensure time scales of the training data isn't skewed compared to production, we tuned the access frequency collection thread to trigger once every second, with 200 milliseconds between clearing access bits and checking them again.

7.2 Memory Allocation Algorithm

To enable access-frequency-aware placement, we first introduced an explicit notion of allocation hotness into TCMalloc.There are currently only three hotness classes for simplicity; this can be tuned to greater granularities if needed.

```
enum class HotnessClass { kCold, kWarm, kHot };
```

We also exposed a prediction interface:

```
HotnessClass HotnessPredictor(
    const void* stack_trace,
    size_t depth,
    size_t allocation_size
);
```

In Temeraire, each hugepage is represented by a `PageTracker` structure. We augmented `PageTracker` with a new field called `hotness_class_` which tracks the hotness class of the hugepage itself. When a hugepage is first created via `Contribute()`, we assign its hotness class based on the predicted hotness of the first allocation placed on that page. Subsequent allocations are admitted only if their predicted hotness matches the page's class, ensuring strict hotness segregation across the heap.

The most substantial modifications occurred within the HugePageFiller, which originally manages all partially filled hugepages in a single global set. To support hotness-aware allocation, we partitioned the filler's data structures into per-class pools:

```
regular_alloc_[HotnessClass][AccessDensityPrediction].
```

These changes create the abstraction of *three independent HugePageFillers* operating in parallel—one for each hotness class. This ensures that hot, warm, and cold objects are never co-located on the same hugepage.

7.3 Allocation Path

We modified the key allocation method, `TryGet`, to perform hotness-aware search. When an allocation of size s with predicted hotness h is requested, `TryGet` now (1) searches *only* within the pool corresponding to h; (2) applies Temeraire's original ranking *within that pool*; (3) if no suitable hugepage is found, requests a new hugepage and assigns it hotness class h; (4) updates indexing functions (`ListFor`, etc.) to incorporate the hotness class dimension. The machine learning model previously trained in Python is ran using PyTorch's C++ API (currently in beta). We decided against invoking a Python runtime for considerations of latency since this model will be invoked for every new memory allocation. Notably, even without an expensive Python runtime, running a neural network is still much more expensive than a typical memory allocation, even on the slow path. While not implemented, this can potentially be mitigated to acceptable levels by caching model results on an

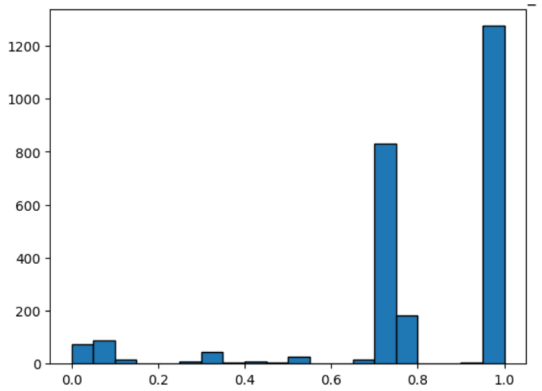
allocation site at runtime based on the stack pointer and allocation size.

8 Evaluation

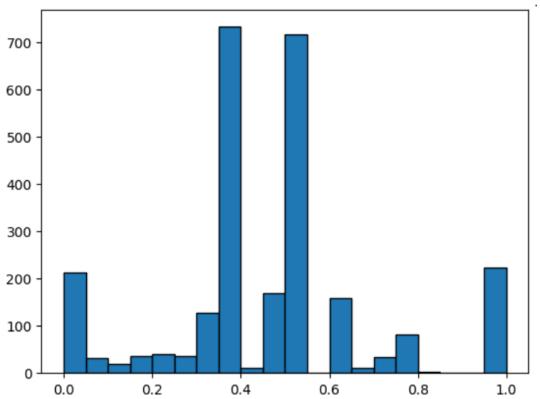
Prior works from Google have trained their memory lifetime prediction model on a large and inherently representative variety of industry software in real-world scenarios in their WSC data centers. While the same opportunities are not available to us, we attempted to replicate the diversity of workload artificially in our setup. We gathered our data from a custom benchmark system on a version of RocksDB, a common kvstore database used in cloud applications, that we compiled against Troubridge. We measured the generalizability of our approach by varying the benchmark to use very different memory access patterns in RocksDB.

We included 11 sets of benchmark data that varies across the dimensions of read/write percentages, value size, and request batching tendencies. Across all these dimensions, we found that varying the size of the data changed memory hotness distribution the most. Figure 2 is included to showcase the differing hotness distributions (measured in proportions of 200ms windows that see access) across data sizes 100, 512, and 1024 bytes. We trained on a mixture of data from 100, 1024, and 1536 byte benchmarks and validated the generalization of our model's prediction ability on 512 bytes. We achieved over 70% accuracy on data sizes seen in training suite and over 60% on data sizes that have not been seen before and which exhibit significantly different access patterns (see figure 3). This provides preliminary evidence that despite our limited access to varied training data, the approach can extract meaningful information and have generalizable predictive power that is more than just memorizing the training suite. Our model's training exhibited significant sensitivity to hyperparameters - up to 10% accuracy on validation data. The hyperparameters we finalized on can be seen in our GitHub repository. Training loss seems to converge after around 60 epochs, after which little gain in generalizable accuracy is seen (figure 4).

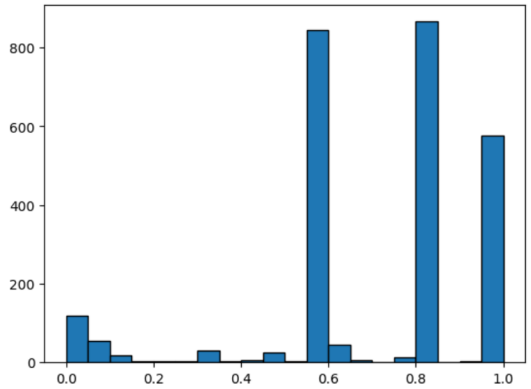
In addition to verifying the correctness of the model a-postori, we also planned to include a performance evaluation. While we have finished the full infrastructure that allows us to incorporate model decisions into TCMalloc allocation decisions, the current performance of running the model on the critical path for all memory allocations introduces excessive overhead and makes it impossible to obtain meaningful performance data. With future work, we hope to incorporate some of the critical optimizations implemented in Google's Learning-Based Memory Allocation, such as stack pointer based caching, in order to make it more applicable to industrial workloads.



(a) Value size 100 bytes



(b) Value size 512 bytes



(c) Value size 1024 bytes

Figure 2: Hotness distribution in benchmarks with different value sizes. x-axis is the proportion of 200ms sampling windows in which access is seen, y axis is the number of allocation sites with the given average hotness

9 Discussion

As expected, performing a full model inference on every allocation is prohibitively expensive in practice. Preliminary microbenchmarking shows that Troubridge takes around 8.8 ms to allocate 1024 bytes 1000 times, which is around 6.7x longer than base TCMalloc (1.3 ms). This leaves a lot of room for further optimizations. A natural extension would be to introduce a per-site inference cache, combined with periodic re-evaluation to correct stale predictions. Implementing such a cache, however, requires careful engineering of fast-path data structures and efficient invalidation mechanisms, and was therefore beyond the scope of this project.

Similarly, while Troubridge is designed to enable far-memory systems to operate on cleanly segregated hot and cold pages, we do not integrate a cold-memory tier or evaluate end-to-end memory

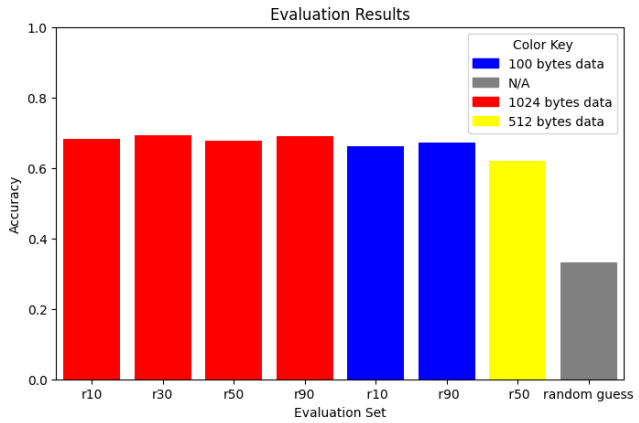


Figure 3: Evaluation Results, color represents the value size of the test dataset, which largely determines memory access behavior. The x-axis labels show the read/write percentage of operations (i.e. r10 = 10% read, 90% write)

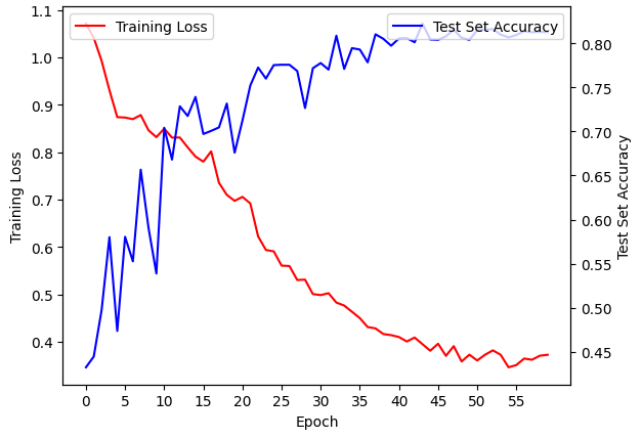


Figure 4: An example training run, where loss and accuracy stablizes at around the 60 epoch mark.

offloading. Our prototype focuses on allocator-level mechanisms rather than full-system behavior.

While our evaluation focuses on the feasibility of memory hotness prediction, only additional benchmarking on real workloads after optimizations could clarify whether hotness-based classification can approximate the performance benefits or drawbacks of lifetime-based placement decisions employed by Temeraire.

Finally, the current prototype assumes fixed page-level hotness classes and does not attempt to correct prediction errors at runtime. A production system would require lightweight access monitoring at page granularity and a policy for dynamically reclassifying pages whose observed access patterns diverge from their predicted class.

In terms of reflections, the project turned out to be much greater in scope than we realized, in parts thanks to the cross-domain expertise required for this project to work. In terms of things we would've done differently, we spent a lot of time trying out different ideas, many of which did not work, and it would've been better if we were able to decide on this from the start. We do plan on improving this project in the future, mainly to further increase the sampling range across different applications, introduce systems optimizations to achieve usability in the kernel, and further improve the model's performance while balancing maintaining a small size for performance. We're also interested in offloaded model inference to further reduce latency. We can have an external machine perform hotness prediction independently on a streamed history of memory allocations, sending finished results over to cache so repeated allocations can use the prediction results without the machine itself performing significant work.

10 Logistics

Bokai worked on data gathering (access frequency), data processing, and model design & training. Chloe worked on data gathering (stack traces & internal representation), TCMalloc model integration, and paper outline. Patrick worked on running the model and the memory allocator on various applications. All group members contributed to common project tasks, including logistics, brainstorming, prior works research, and paper writing.

References

[1] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 257–273. https://www.usenix.org/conference/osdi21/presentation/hunter

[2] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 317–330. doi:10.1145/3297858.3304053

[3] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*. 2884–2892. doi:10.1109/BigData.2017.8258257

[4] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2024. Combining Machine Learning and Lifetime-Based Resource Management for Memory Allocation and Beyond. *Commun. ACM* 67, 4 (March 2024), 87–96. doi:10.1145/3611018

[5] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: high-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 18, 18 pages.

[6] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. doi:10.1145/3342195.3387517

Received 20 February 20XX; revised 12 March 20XX; accepted 5 June 20XX