Learning-based Memory Allocation for C++ Server Workloads

Martin Maas, David G. Andersen*†, Michael Isard, Mohammad Mahdi Javanmard*‡, Kathryn S. McKinley, Colin Raffel

Google Research †Carnegie Mellon University ‡Stony Brook University

Abstract

Modern C++ servers have memory footprints that vary widely over time, causing persistent heap fragmentation of up to 2× from long-lived objects allocated during peak memory usage. This fragmentation is exacerbated by the use of huge (2MB) pages, a requirement for high performance on large heap sizes. Reducing fragmentation automatically is challenging because C++ memory managers cannot move objects.

This paper presents a new approach to huge page fragmentation. It combines modern machine learning techniques with a novel memory manager (Llama) that manages the heap based on object lifetimes and huge pages (divided into blocks and lines). A neural network-based language model predicts lifetime classes using symbolized calling contexts. The model learns context-sensitive per-allocation site lifetimes from previous runs, generalizes over different binary versions, and extrapolates from samples to unobserved calling contexts. Instead of size classes, Llama's heap is organized by *lifetime* classes that are dynamically adjusted based on observed behavior at a block granularity.

Llama reduces memory fragmentation by up to 78% while only using huge pages on several production servers. We address ML-specific questions such as tolerating mispredictions and amortizing expensive predictions across application execution. Although our results focus on memory allocation, the questions we identify apply to other system-level problems with strict latency and resource requirements where machine learning could be applied.

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

Keywords Memory management, Machine Learning, Lifetime Prediction, Profile-guided Optimization, LSTMs

* Work done while at Google.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03.
https://doi.org/10.1145/3373376.3378525

ACM Reference Format:
Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378525

1 Introduction

Optimizing interactive web services, many of which are written in C++, requires meeting strict latency requirements while minimizing resource usage. Users abandon services if response times are too slow and data center costs are directly proportional to resource usage. Multithreaded services require large heaps both to minimize the number of deployed instances and to handle multiple requests simultaneously. Hardware has not kept pace with these demands. While memory sizes have increased, Translation Lookaside Buffers (TLB) have not, because address translation is on the critical path. One solution is increasing TLB reach with huge (2 MB) pages, i.e., each entry covers more memory. Huge pages reduce TLB misses, improving performance by up to 53% [33, 37]. Looking forward, 1 GB pages are already available and variable-sized ranges can eliminate even more TLB misses [27, 33]. Future virtual memory systems may hence predominantly rely on huge pages and ranges.

These trends require workloads to efficiently use huge pages. While Operating Systems (OS) have explored transparent huge pages [37, 45], they either trade performance for space, increasing the physical memory footprint by up to 23% and 69% on server workloads [37], or break up huge pages, sacrificing performance (TLB hits) and depleting contiguous physical memory for all workloads on the machine [37, 45]. If the C++ memory allocator is not huge page aware, it may further defeat the OS. Only one C++ memory allocator in the literature uses huge pages, but its evaluation uses microbenchmarks [36]. To our knowledge, no current memory allocator efficiently manages memory entirely with huge pages without incurring significant fragmentation.

We identify a root cause of huge page fragmentation in long-running servers: allocations of long-lived objects at peak memory usage. Since C++ allocators cannot move objects, using huge pages increase the probability of one long-lived object preventing a page from being released to the

OS. For instance, if 99.99% of objects are short-lived and their average size is 64 B, then using 4 KB pages, the probability that any given page contains a long-lived object is less than 1% ($1-(0.9999)^{4096/64}$). Using 2 MB huge pages, the probability is 96%. Figure 1 shows that heap fragmentation with huge pages for a production image processing service on a synthetic workload grows over time as a function of peak memory consumption. Many web services exhibit such highly variable memory consumption [37, 40] and allocate critical long-lived session state.

Solving this problem fundamentally depends on reasoning about object lifetimes and grouping objects with similar lifetimes together [4, 11, 16, 17]. Prior lifetime region and pool memory management techniques [6, 34, 43] depend on programmer intervention and are limited because not all lifetimes are statically known, software can change over time, and libraries are used in multiple contexts. Previous object lifetime predictors for C++ and garbage collected languages use profiling to classify objects as short or long lived, but are used in settings (such as pretenuring) where mispredictions are tolerable [4, 11, 16, 30]. In contrast, because every wrong prediction may retain up to 2 MB and errors accumulate on long-running servers, we require an approach that does not induce fragmentation upon misprediction, and need to address the following challenges:

Lifetime accuracy and coverage. Full coverage and perfect accuracy are not achievable because exercising all possible application behavior ahead-of-time is challenging, especially for evolving servers configured in myriad ways with different libraries.

Overheads. Continuous profiling in deployment is not practical because it adds 6% overhead [13, 42], which can be more than memory allocation itself [31].

These challenges require accurate predictions in previously unobserved contexts and a memory manager that explicitly reasons about lifetimes to recover from mispredictions. Our contributions are as follows: (1) The design of a recurrent neural network predictor that trains on samples and generalizes to different application versions and build configurations with accurate, but not perfect prediction. (2) A novel *Learned Lifetime-Aware Memory Allocator* (Llama) with low fragmentation that only uses huge pages, but subdivides them into blocks and lines. It then manages huge pages and blocks using their predicted and observed *lifetime class*. (3) Some lessons for applying ML to other systems problems.

To increase coverage and accuracy, the predictor can be trained on different server versions and configurations. To reduce profiling overhead, we sample allocations and frees to produce training data with allocation calling context (i.e., stack traces) and object lifetimes. We classify objects into lifetime classes separated by orders of magnitude: ≤ 10 ms, 100 ms, 1 s, 10 s, etc. Based on the insight that program symbols in stack traces carry meaning similar to words in natural

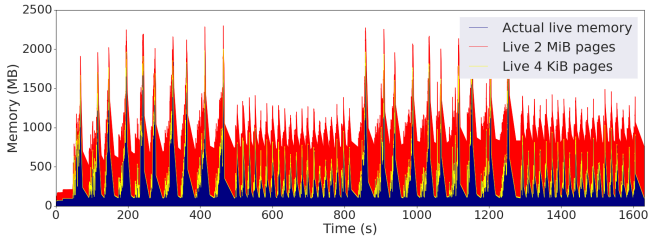


Figure 1. Image server memory usage resizing groups of large and small images either backed by huge (red) or small (yellow) pages in the OS, derived from analyzing an allocation trace in a simulator. Huge pages waste systemically more memory and increasingly more over time.

language, we train simple language models on symbolized calling contexts. We use a *Long Short-Term Memory* (LSTM) recurrent neural network model to learn common and rare contexts (Section 5). Whereas other lifetime predictors are simple binary classifiers for exactly matching contexts or single allocation sites [4, 11, 16], Llama's predictor learns multiple lifetime classes and accurately predicts unobserved contexts because it uses program symbols, rather than matching stack traces or hard-coded allocation sites. However, performing inference on every allocation is too expensive, so Llama caches inferences and periodically re-evaluates them.

In contrast to C/C++ free-list allocators that organize the heap based on object size classes [5, 9, 19, 21, 38], Llama organizes the heap based on lifetime classes. It manages huge pages by subdividing them into 8 KB blocks and 128 B lines. It assigns each huge page and block a *lifetime class* (LC). Llama maintains two invariants: 1) it fills blocks with one predicted lifetime class (LC) at a time and 2) this LC is the same or shorter than the huge page's LC. The huge page's LC thus matches or over-predicts its blocks to tolerate mispredictions. To limit fragmentation and handle mispredictions, Llama dynamically *reclassifies* a huge page's LC based on its observed block lifetimes.

Llama assigns each huge page a predicted LC and a deadline, by when all objects should be dead. It first fills blocks in huge pages with objects of the same LC, marking these same-LC blocks *residual*. When blocks are freed, Llama aggressively reuses them for predicted shorter-lived (*non-residual*) LC blocks. These shorter-lived blocks are likely to be freed before the huge page's deadline. This policy limits fragmentation without extending huge page lifetimes. If the deadline expires and any residual blocks are still live (i.e., lifetime was under-predicted), Llama promotes the huge page to the next-longer-lived LC. If all residual blocks have been freed (i.e., lifetime may be over-predicted since all live blocks have a lower LC than their huge page), Llama reduces the huge page's LC and its remaining blocks become residual. Llama tracks line liveness in a block and recycles partially free blocks. Llama's hierarchical heap organization (huge page, block, line) follows Immix's (block, line) mark-region design [10, 48]. Its

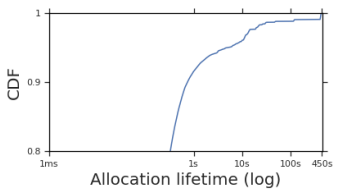


Figure 2. Long tail of object lifetimes from a single run; x-axis is log-scale. The vast majority of objects are short-lived, but rare long-lived objects impact fragmentation.

lifetime organization is similar to generational and lifetime-based copying garbage collectors [8, 10, 18, 49, 51]. However, unlike a managed runtime where GC can move objects between regions, Llama cannot move objects and instead reclassifies huge pages. Llama is the first C/C++ allocator to organize the heap based on lifetime versus object size.

We prototype Llama and compare to `TCMalloc`, a popular and highly tuned allocator, backed by OS huge pages. Llama never breaks up huge pages while simultaneously reducing fragmentation by up to 78% on several production code bases. We compare Llama to Mesh [46], which uses allocation randomization and page combining to combat fragmentation for small pages. Using Mesh's publicly available scripts on a worst case microbenchmark that emulates address randomization for long lived objects, Llama reduces fragmentation over Mesh on huge pages by an order of magnitude. We further show Llama accurately predicts new contexts, adds little overhead, and recovers from mispredictions. We also draw lessons for applying machine learning in other latency-critical systems settings.

2 Motivation and Background

2.1 Server Fragmentation

We demonstrate huge page fragmentation on a production C++ image server that applies filters and transforms images. We drive this server using a request generator that mimics workload shifts over time. One iteration running for 448 s with an average live set of 628 MB has ≈110 M allocations from ≈215 K allocation contexts. It allocates (with `malloc()` or `new`) objects of different sizes and frees (with `free()` or `delete`) allocated memory using TCMalloc [21]. Like all C/C++ allocators, once TCMalloc places objects in virtual memory, it never moves them. We extended TCMalloc to record every object allocation and free with the address, size, thread, dynamic stack trace, and timestamp.

We replay these traces in a simulator that determines which pages contain live objects at a given time by modeling the OS giving out 4 KB or 2 MB pages for unmapped virtual addresses. Figure 1 shows the average fragmentation (ratio of memory occupied by live pages to actual live memory) is 1.03x when the OS backs memory with 4 KB pages, but increases to 2.15x with huge pages and gets worse over time. Figure 2 shows object lifetimes. While 92% of the over 100 M allocations live for less than 1 s, 4% (millions) of allocations live for over 10 s and 1% (thousands) live for over 100 s.

These long-lived objects cause excessive fragmentation. Workloads with varying memory footprint are more susceptible to this problem because small numbers of long-lived objects on a huge page prevent reusing it for large allocations. In the image server, short-lived objects that cause the heap to grow temporarily include data structures to process each request and image data. At the same time, it allocates long-lived objects that are used for tracking the cluster environment, system statistics, log tracing, and long-lived session state. Long-lived state per request is application critical and is not the result of poor software engineering.

Highly varying memory footprints are typical of servers [37, 40]. Fragmentation remains an open problem, recently reported for many applications and allocators beyond TCMalloc [36, 37, 46]. However, strategies for addressing fragmentation in these allocators are designed for 4 KB pages [5, 46]. As our probabilistic argument in Section 1 points out, addressing fragmentation for huge pages is fundamentally more difficult, particularly without lifetime information.

2.2 Lifetime Prediction Challenges

Prior work predicts object lifetime as long or short based on allocation site and precisely matching calling context [11, 16] (although Cohn and Singh did use stack *data* for predictions instead [17]). Current approaches typically store a table of allocation sites, together with a summary of observed per-site lifetimes [13]. They either 1) collect lifetime information at runtime, i.e., dynamic pretenuring [16, 30] or 2) use profile-guided optimization (PGO), collecting lifetimes offline with special instrumentation, analyzing it offline, and then using it in deployment [11]. Lifetime prediction faces the following significant challenges:

Overheads. Collecting allocation lifetimes incurs a substantial overhead, e.g., stack tracing adds 14% end-to-end overhead and writing to disk further increases the cost, making continuous profiling infeasible in production. Looking up a predicted object lifetime also incurs overhead, including recording the calling context. Table 1 shows recording the calling stack for an allocation can take an order of magnitude longer than the allocation, which is problematic. Solutions include instrumenting the stack prologue and epilogue to keep track of the current stack through a series of bits stored in a register [12, 13, 29]. However, overheads of this approach are ≈6% and higher, exceeding all the time spent in memory allocation [31]. We solve these problems by using stack height and object size for per-site prediction and cache lookups.

Coverage and Accuracy. Encountering a sufficient fraction of allocation sites for accurate prediction is critical. When collecting lifetime data online, we cannot make a prediction unless we have seen its context at least once. However, in

TCMalloc Fast Path (new/delete)	8.3 ns
TCMalloc Slow Path (central list)	81.7 ns
Capture full stack trace	396 ns ± 364 ns
Look up stack hash (Section 7)	22.5 ns

Table 1. Timescale comparisons

Version Difference	Matching/Total # Traces
Revisions 1 week apart	20,606 / 35,336 (58.31%)
Revisions 5 months apart	127 / 33,613 (0.38%)
Opt. vs. non-opt. build	43 / 41,060 (0.10%)

Table 2. Fraction of individual stack traces that match between different binary versions (using exact match of symbolized function names).

our example workload, 64% of distinct allocation contexts are seen only once and 17% of all permanent allocations (i.e., allocations that never get freed) are from contexts that are only encountered once. PGO avoids this problem by using profiles from previous runs, but is more difficult to apply to lifetimes than in traditional scenarios, such as inlining [15]. First, these decisions do not depend on the dynamic calling context. As such, each call site only needs to be observed once (in *any* context). In contrast, lifetime prediction requires observing *every* context for full coverage. For instance, inlining data only needs to collect a single event per sample, while lifetime profiling requires observing both the allocation and free. As such, profiling data is more scarce in our setting than typical PGO scenarios.

Instability. Stack traces are brittle when used across executions. Even stack traces on the exact same binary may differ due to address layout randomization. Using symbol information, it is possible to reconstruct the original method name for each stack frame, but different builds of the same binary may still differ. For example, changing libraries can affect inlining decisions, different compiler settings lead to slightly different symbol names, and function names and interfaces change over time. This problem also occurs when collecting traces across a large number of instances of the same server with different build configurations and software versions. Table 2 shows that the fraction of matching stack traces between builds with even minor changes is low and decreases over time. This result explains why almost all practical lifetime predictors today use online profiling instead of PGO, or rely on site instead of the full dynamic stack.

We solve coverage and instability problems by enhancing PGO to work without observing all contexts. We design an ML-based predictor that learns on calling contexts of tokenized class and method names to produce accurate predictions for unobserved contexts. If a single binary is deployed sufficiently often to achieve full coverage, our approach reduces to conventional PGO. However, these situations are rare — most companies have different software versions in production at the same time [7, 47].

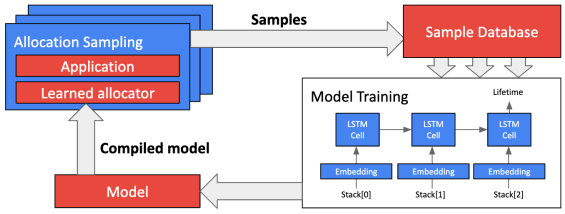


Figure 3. Overview of our ML-based Allocator

3 Overview of Lifetime Prediction

We address overhead and coverage challenges by sampling multiple executions. Sampling is suitable for both server applications in datacenters and multiple runs of a popular application (e.g., a web browser) on a client. We connect to a given application for a sample period and collect lifetimes for a small fraction of all allocations that occur during this period (Section 4).

Sampling may not observe all allocation calling contexts and we must combine samples from a heterogeneous set of different software versions, while the code bases are constantly updated. We therefore cannot simply use a lookup table, as shown in Table 2. Our solution is to use ML on the observed samples of tokenized calling contexts (i.e., symbolized/textual stack traces) to predict object lifetimes. We train a model that maps from calling context to lifetime, while generalizing to previously unseen contexts. The predictions drive our novel C++ memory allocator that organizes the heap based on lifetime to reduce fragmentation. While our prototype focuses on learning a mapping from contexts to lifetime, we could add other input features, such as performance counters or user-level statistics.

Another challenge is to perform prediction without significant overhead. The allocation fast path is 8.3 ns (Table 1), which is too short to obtain a prediction from an ML model. In fact, it is not even sufficient to gather all the required features since collecting a deep stack trace takes 400 ns. We address this problem by *not* invoking the model for every allocation. Instead, we use a hashing-based mechanism (Section 7) to identify previously seen contexts by using values that are already in registers (the return address and stack pointer) to index a hash table and execute the model only if the lookup fails. We thus amortize model executions over the lifetime of a long-running server. We discuss other strategies to reduce this cost even further (Section 10). We now explain each component in detail.

4 Sampling-based Data Collection

Our sampling approach periodically connects to servers (for a time period such as ≈5 minutes) and samples a subset of all memory allocations. Each sample includes stack trace, object size and address at allocation and deallocation time.

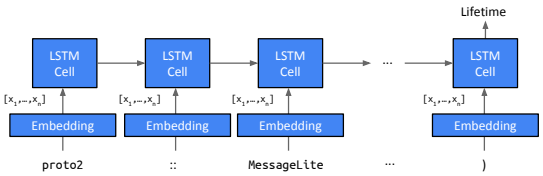


Figure 4. LSTM-based model architecture

This approach follows continuous profiling tools used in production settings [31].

We integrate this approach into TCMalloc [21]. Its existing heap profiling mechanism identifies long-lived objects well by producing a list of sampled objects at the end of the application's execution, most of which are long-lived, including their allocation sites. It misses the more prolific allocations of short-lived objects that are not live at the end of the program. We therefore extend the heap profiling mechanism to record frees (deallocations) as well. We do so using *hooks* (i.e., functions) that are called periodically, based on the number of allocated bytes. These hooks incur virtually no overhead when they are disabled. When enabled, each sampled allocation triggers TCMalloc to store it at a special address in memory and then deallocation can identify those sampled objects and call the corresponding deallocation hook.

We install an HTTP handler accessible by pprof [25], an open-source profiling and analysis tool. When invoked, the handler registers two hooks, one for allocation and one for deallocation. It also allocates a new data structure (outside of the TCMalloc-managed heap) to store observed stack traces. The allocation hook stores the allocation's full stack trace, a timestamp of the allocation, object size, alignment, and the stack and processor ID of the allocation into a hash table, indexed by a pointer to where the object was allocated. The deallocation hook matches its pointer to the hash table and if it finds an entry, records its own stack trace, timestamp and thread/CPU where the deallocation occurred. This pair of entries is now stored in a different hash table, which is used to deduplicate all samples. For each entry, we keep a running tally of the distribution of lifetimes, by storing the maximum, minimum, count, sum and sum of squares (the latter two allow us to calculate mean and variance of the lifetime at a later point). We also store how many of these allocations were allocated and deallocated on the same CPU or thread (we do not currently use this information, but explain in Section 10 how it might be used). At the end of a sampling period, we store the result into a protocol buffer [24].

In deployment, we would periodically connect to servers in the fleet and collect samples. For this research, we run smaller-scale experiments to understand the trade-offs of our approach and mostly rely on full traces collected by instrumenting allocation and free calls. While too expensive for production, this approach is useful for understanding coverage of different sampling rates (Section 9), or to replay the entire trace in simulation (Section 2). The two approaches produce consistent results (Section 9.3).

5 Lifetime Prediction Model

Our goal is to predict object lifetimes based on our collection of past lifetime samples. As shown in Section 2, a simple lookup table is insufficient and brittle to changes in the application. We instead construct a dataset of samples from a range of scenarios and train a machine learning model on this dataset to generalize to previously unseen stack traces.

5.1 Data Processing

We pre-process our sample data using a distributed dataflow computation framework [2, 14]. We group inputs by allocation site and calculate the distribution of observed lifetimes for each site. We use the 95th percentile T^i_{95} of observed lifetimes of site i to assign a label $L_i \in \{1, \ldots, 7, \infty\}$ to the site such that $T^i_{95} < T(L_i) = (10)^{L_i}$ ms. Objects the program never frees get a special long-lived label ∞. This produces lifetime classes of 10 ms, 100 ms, 1 s, 10 s, 100 s, 1000 s, $\geq$1000 s, and ∞. Our model classifies stack traces according to these labels. To ensure that our model assigns greater importance to stack traces that occur more often, we weight each stack trace according to the number of times it was observed and sample multiple copies for frequently occurring traces. The resulting datasets for our applications contain on the order of tens of thousands of elements.

The use of wallclock time for lifetime prediction is a departure from prior work that expresses lifetime with respect to *allocated bytes* [4], which can be more stable across environments (e.g., server types) at short timescales. We experimented with logical time measured in bytes, but for our server systems, wallclock time works better. We believe time works better because 1) our lifetime classes are very coarse-grained (10×) and absorb variations, 2) if the speed difference between environments is uniform, nothing changes (lifetime classes are still a factor of 10× apart). Meanwhile, variations in application behavior make the bytes-based metric very brittle over long time ranges (e.g., in the image server, the sizes of submitted images, number of asynchronous external events, etc. dilate logical time).

5.2 Machine Learning Model

We use a model similar to text models. First, we treat each frame in the stack trace as a string and tokenize it by splitting based on special characters such as: , and ::. We separate stack frames with a special token: @. We take the most common tokens and create a table that maps them to a particular ID with one special ID reserved for unknown or rare tokens, denoted as UNK. The table size is a configuration parameter (e.g., 5,000 covers most common tokens).

```
1  __gnu_cxx::__g::__string_base char, std::__g::char_traits
   char,std::__g::allocator char::_M_reserve(unsigned long)
2  proto2::internal::InlineGreedyStringParser(std::__g::
   basic_string char, std::__g::char_traits char,std::__g::
   allocator char*,char const*,proto2::internal::ParseContext*)
3  proto2::FileDescriptorProto::_InternalParse(char const*,
   proto2::internal::ParseContext*)
4  proto2::MessageLite::ParseFromArray(void const*, int)
5  proto2::DescriptorPool::TryFindFileInFallbackDatabase(std::
   __g::basic_string char, std::__g::char_traits char , std::
   __g::allocator char const ) const
6  proto2::DescriptorPool::FindFileByName(std::__g::
   basic_string char, std::__g::char_traits char , std::__g::
   allocator char const) const proto2::internal::
   AssignDescriptors(proto2::internal::AssignDescriptorsTable*)
7  system2::Algorithm_descriptor()
8  system2::init_module_algorithm_parse()
9  Initializer::TypeData::RunIfNecessary(Initializer*)
10 Initializer::RunInitializers(char const*)
11 RealInit(char const*, int*, char***, bool, bool)
12 main
```

Figure 5. An example of an altered but representative stack trace used to predict object lifetimes.

We use a long short-term memory (LSTM) recurrent neural network model [28]. LSTMs are typically used for sequence prediction, e.g., for next-word prediction in natural language processing. They capture long-term sequential dependencies by applying a recursive computation to every element in a sequence and outputting a prediction based on the final step. In contrast, feed-forward neural networks like multi-layer perceptrons [23] or convolutional neural networks [20, 39] can recognize local patterns, but require some form of temporal integration in order to apply them to variable-length sequences.

Our choice of an LSTM is informed by stack trace structure. Figure 5 shows an example. Sequentially processing a trace from top to bottom conceptually captures the nesting of the program. In this case, the program is creating a string, which is part of a protocol buffer ("proto") parsing operation, which is part of another subsystem. Each part on its own is not meaningful: A string may be long-lived or short-lived, depending on whether it is part of a temporary data structure or part of a long-lived table. Similarly, some operations in the proto might indicate that a string constructed within it is temporary, but others make the newly constructed string part of the proto itself, which means they have the same lifetime. In this case, the enclosing context that generates the proto indicates whether the string is long or short-lived.

For our model to learn these types of patterns, it must step through the stack frames, carrying through information, and depending on the context, decide whether or not a particular token is important. This capability is a particular strength of LSTMs (Figure 4). We feed the stack trace into the LSTM as a sequence of tokens (ordered starting from the top of the trace) by first looking up an "embedding vector" for each token in a table represented as a matrix A. The embedding matrix A is trained as part of the model. Ideally, A will map tokens with a similar meaning close together in embedding space, similar to word2vec embeddings [41] in natural language processing. Here lies an opportunity for the model to generalize. If the model can learn that tokens such as `ParseFromArray` and `InternalParse` appear in similar contexts, it can generalize when it encounters stack traces that it has not seen before.

Note that our approach is not specific to LSTMs. We chose the LSTM architecture since it is one of the simplest sequence models, but future work could explore more sophisticated model architectures that could incorporate more details of the underlying program (e.g., Graph Neural Networks trained on program code [3]). Our specific model architecture is a standard single-layer LSTM with a hidden state size of 64 (we experiment with 16 as well), embedding size of 32, uses a softmax output, and is trained against a standard cross-entropy classification loss via gradient descent. The final state of the LSTM is passed through a fully connected layer. Training uses the Adam optimizer [35] with a learning rate of 0.001 and gradients clipped to 5.0.

5.3 Model Implementation

We implement and train our model using TensorFlow [1]. Calling into the full TensorFlow stack to obtain a lifetime prediction would be prohibitively expensive for a memory allocator, so after training, we use TensorFlow's XLA compiler to transform the trained model into C++ code that we compile and link into our allocator directly. The model runs within the allocating thread. To allow multiple threads to use the model concurrently, we instantiate the model's internal buffers multiple times and add concurrency control.

6 Lifetime Aware Allocator Design

This section introduces a fundamentally new design for C/C++ memory managers based on predicted object lifetimes. Instead of building an allocator around segmenting allocations into size classes [5, 9, 19, 21, 36, 38], we directly manage huge pages and segment object allocation into predicted lifetime classes. We further divide, manage, and track huge pages and their liveness at a block and line granularity to limit fragmentation. We implement our allocator from scratch. It is not yet highly tuned, but it demonstrates the potential of a lifetime-based approach. We address two challenges required to incorporate ML into low-level systems: 1) how to deal with mispredictions and 2) prediction latencies that are orders of magnitude longer than the typical allocation latency. We first describe the structure of the memory allocator, then how we make fast predictions, and follow with key implementation details.

6.1 Heap Structure and Concurrency

We design our memory manager for modern parallel software and hardware. LLAMA organizes the heap into huge pages to increase TLB reach. To limit physical fragmentation, we divide huge pages into 8 KB blocks and track their liveness. LLAMA assigns each active huge page one of N lifetime

classes (LC), separated by at least an order of magnitude (e.g., 10 ms, 100 ms, 1000 ms, ..., ∞). Our implementation uses a maximum of $N = 7$ lifetime classes. Llama exploits the large virtual memory of 64-bit architectures, as fragmentation of virtual memory is not a concern. Llama divides virtual memory into 16 GB *LC regions*, one per lifetime class. Section 8 describes enhancements when an LC region is exhausted.

The *global* allocator manages huge pages and their blocks. It performs bump pointer allocation of huge pages in their initial LC regions, acquiring them from the OS. It directly manages large objects (>= 8 KB), placing them into contiguous free blocks in partially free huge pages or in new huge pages. A huge page may contain large and small objects.

Llama achieves scalability on multicore hardware by using mostly unsynchronized *thread-local* allocation for small objects (<=8 KB). The global allocator gives *block spans* to local allocators upon request. When a thread-local allocator allocates the first object of a given LC or it exhausts its current LC block span, it requests one from the global allocator. Block spans consist of M blocks and reduce synchronization with the global allocator. Our implementation uses $M = 2$ (16 KB block spans) with 16 KB alignment. Llama further subdivides block spans into 128 B lines and *recycles* lines in partially free block spans for small objects (see Section 6.6). It tracks line and block liveness using *object counters*. Small objects never cross span boundaries, but may cross line and block boundaries. Each thread-local allocator maintains one or two block spans per LC for small objects.

Llama tracks predicted and actual block lifetimes and uses them to decrease or increase their huge page's LC. Llama maintains the following invariants. 1) It allocates only objects of one predicted LC into a block or span at a time. 2) A huge page contains blocks with the same or shorter predicted LC.

We next describe how we use LC predictions to manage huge pages and blocks. Sections 6.3 and 6.4 describe the policies that limit fragmentation and dynamically detect and control the impact of mispredicted lifetimes. Section 6.6 then describes how Llama uses lines to identify and recycle memory in partially free block spans.

6.2 Lifetime-Based Huge Page Management

Each huge page has three states: *open*, *active*, and *free*. Open and active blocks are *live*. The first allocation into a huge page makes it open and determines its LC. Only one huge page per LC is open at a time. While a huge page is open, Llama only assigns its blocks to the same LC. Llama transitions a huge page from open to active and assigns it a deadline after filling all its constituent blocks for the first time. The huge page remains active for the rest of its lifetime. The OS backs huge pages lazily, upon first touch. A huge page is free when all its blocks are free and is immediately returned to the OS.

All blocks in a huge page are *free* or *live*; *open* or *closed* for allocation; and *residual* or *non-residual*. All blocks are initially free. When the global allocator returns a block span to a local allocator, it marks the blocks *open* for allocation. If the blocks are on an *open huge page*, it also marks the blocks *residual*. Residual blocks are predicted to match the LC of their huge page. An *active huge page* may also contain other live (non-residual) blocks, but these blocks will contain objects of a shorter lifetime class, as explained below. Thread-local allocators bump-pointer allocate small objects in block spans. When they exhaust a span, they mark it *closed*.

Llama first fills a huge page with same LC blocks and then transitions it from open to active. At this point, the huge page contains residual blocks and maybe free blocks. Figure 6 shows an illustrative, but simplified, example of the logical LC Llama heap (huge pages and blocks) and its behavior over time. This heap has three lifetime classes, separated by orders of magnitude. A large amount of initial allocation in Figure 6a, including a large object allocation into huge page 11 and 12, is followed by a large number of frees in Figure 6b. Llama returns free huge pages 2 and 6 to the OS.

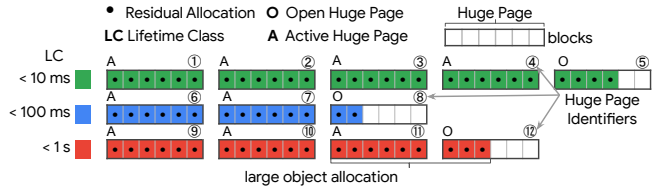
6.3 Limiting Fragmentation by Recycling Blocks

Notice in Figure 6b active huge pages contain free blocks and live residual blocks of the same LC. Llama limits fragmentation by aggressively *recycling* such free blocks for objects in shorter LCs (except for the shortest LC, since no LC is shorter). Section 6.5 explains the fast bit vector operations that find recyclable blocks of the correct size and alignment.

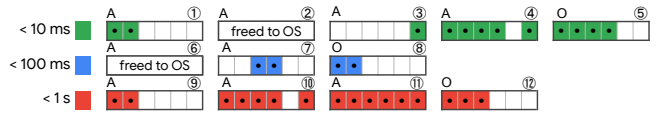
Given a request for LC lr, the global allocator prefers to use free blocks from a longer-lived active huge page (LC $> lr$). These recycled blocks are allocated non-residual, as illustrated in Figure 6c. If no such recyclable blocks exist, the global allocator uses block(s) from the open huge page of the same LC = lr. Intuitively, if the predictor is accurate or overestimates lifetime class, the program with high probability will free shorter-lived objects on recycled blocks before it frees residual blocks with the same LC as the huge page. Because lifetime classes are separated by at least an order of magnitude, the allocator may reuse these blocks many times while the longer-lived objects on the huge page are in use, reducing the maximum heap footprint. If the predictor underestimates lifetime, the objects will have more time to be freed. This design is thus tolerant of over and under estimates of lifetime.

For large objects, the global allocator assigns blocks directly. For example, given the heap state in Figure 6b and a request for a two block large object with $lr < 10$ ms, the global allocator allocates it into huge page 7 with LC < 100 ms and marks the blocks non-residual, as illustrated in Figure 6c.

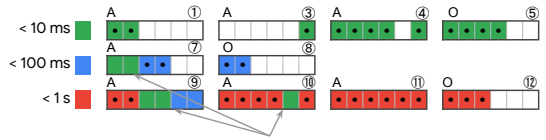
When Llama recycles a block span (assigning it to a local allocator), it marks the blocks open and non-residual. The local allocator assigns the span to the requested LC lr, *even if* the span resides on a huge page assigned to a longer lifetime class. The local allocator only allocates objects of this predicted lifetime lr into this span. After it fills the span with lr object allocations, it marks the blocks closed. This



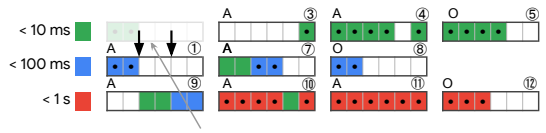
(a) Initial allocations. Huge pages are bump-pointer allocated into LC regions. Each huge page is first filled with same LC blocks, marked residual with a dot.



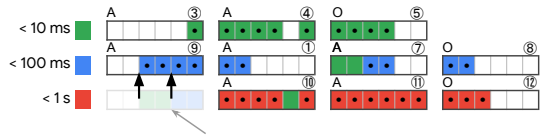
(b) After objects free, some blocks and huge pages are free (white). LLAMA immediately returns free huge pages to the OS to control maximum heap size.



(c) Subsequent allocations of shorter LC small objects first fill free blocks in the highest LC in A(ctive) huge pages 9 and 10, and then blocks in huge page 7. These blocks are not residual (no dot) and expected to be freed before the residual blocks. O(pen) pages 5, 8, and 12 are ineligible for such allocation.



(d) When huge page 1's deadline expires, residual blocks are still live (misprediction). LLAMA increases the huge page's LC by one, from 10 to 100 *ms*. Residual blocks remain residual; their expected lifetime is now at least 100 *ms*.



(e) Huge page 9 only contains non-residual blocks and consequently, LLAMA decreases its LC. It marks all live blocks residual since they match or are less than the huge page's LC.

Figure 6. LLAMA's logical heap organization with three lifetime classes (< 10 *ms*, < 100 *ms*, < 1 *s*). Each live huge page is A(ctive) or O(pen) and divided into blocks. Block color depicts predicted LC or free (white). Residual blocks are marked with a dot. Deadlines and lines are omitted.

policy guarantees that when a block is open for allocation, it receives only one LC.

LLAMA's recycling policy is configurable. In the current implementation, LLAMA prefers $lr + 1$ for large objects and the longest available LC for small objects.

6.4 Tolerating Prediction Errors

Lifetime prediction will never be perfect. LLAMA tolerates mispredictions by tracking block and huge page lifetimes using deadlines. It promotes huge pages with under-predicted object lifetimes to the next longer LC and huge pages with over-predicted objects to the next shorter lifetime class.

We detect under-prediction of lifetimes using deadlines. When a huge page becomes full for the first time, the global allocator transitions it from *open* to *active* and assigns it a deadline as follows:

$$deadline = current\_timestamp + \mathrm{K} \times LC_{HugePage}$$

When LLAMA changes the LC of a huge page, it assigns the huge page a new deadline using the same calculation and the new lifetime class. We experimented with $K = 2$ and $K = 4$.

When a huge page's deadline expires, then the predictor made a mistake. To recover, LLAMA increases the huge page's lifetime class and gives it a new deadline. Figure 6d depicts this case. The residual blocks in huge page 1 outlive their deadline and LLAMA increases its LC to 100 ms. A huge page may also contain non-residual blocks which it leaves unchanged. LLAMA essentially predicts that the residual blocks were just mispredicted by one LC and non-residual blocks are shorter lived than this LC. If either live longer that this LC, this process will repeat until the blocks are freed or reach the longest lived LC. This policy ensures that huge pages with under predicted objects eventually end up in the correct lifetime class, tolerating mispredictions.

LLAMA's recycling mechanism works well for both accurate and under-predicted lifetimes. If all lifetimes are accurate or under-predicted, a program will free all residual blocks before their huge page deadline since the deadline is generous. As blocks become free on active huge pages, the allocator may recycle them for shorter lifetime classes, as explained above. LLAMA may repeatedly recycle blocks on active huge pages, each time they are freed. Before the deadline expires, if all blocks in the huge page are free at once, LLAMA simply releases it to the OS. Otherwise given accurate or under prediction, the huge page will at some point contain only live non-residual (shorter LC) blocks when the deadline expires. LLAMA will then decrease the huge page's LC by one and compute a new deadline using the current time and new LC.

Figure 6e shows such an example. Because huge page 9 contains only non-residual blocks, LLAMA decreases its LC and marks all live blocks residual. With accurate and under-predicted lifetimes, this process repeats: either the huge page is freed or its LC continues to drop until it reaches the shortest LC. In the shortest LC since no blocks are recycled and when prediction is accurate, all blocks are freed before the deadline and the huge page is released.

6.5 Data Structures

LLAMA tracks liveness at the huge page, block, and line granularity. It stores metadata in small pages at the beginning of each 16 GB LC region. Each huge page in a region corresponds to one 256 B metadata region in the metadata page. Mapping between a huge page and its metadata therefore consists of quick bit operations.

The global allocator tracks active huge pages in a list for each LC and open blocks in each huge page in bit vectors. The metadata for huge pages stores bitmaps with 256 bits (< one cache line). One bitmap stores whether a block is live (i.e., contains live objects). Another bitmap identifies *residual* blocks that contain live objects of the same LC as the huge page. Non-residual live blocks thus contain shorter-lived objects. When the global allocator assigns a block from the same LC as the request, it marks the block residual.

When Llama frees a block, it clears the corresponding bits in both bitmaps. If all blocks in a huge page are free (the live bitmap is all zeros), it returns the huge page to the OS. Otherwise, it examines the residual bitmap. If it is all zeroes, any live blocks must contain objects with shorter predicted lifetimes. Llama therefore assign the page to the next-lower lifetime class (huge page 9 in Figure 6d), copies the current live bitmap into the residual bitmap and continues. The huge pages in the shortest LC contain no recycled blocks.

6.6 Recycling Lines in Block Spans

This section describes how Llama limits fragmentation by further subdividing block spans into lines, recycling lines in partially free spans, and using the overflow allocation optimization [10]. For spans with small objects, Llama keeps live object counts for each line, and a count of live lines per span. Small objects occupy one or more contiguous lines and only one span. Once a span is closed (filled at least once), subsequent frees may create a partially free span.

Multiple threads can free objects in a span, thus counting live objects requires synchronization. For each small object allocation, local allocators perform an atomic increment on its span's and line(s)'s object counts. For each free, an atomic decrement is performed on the counts. Section 8 describes this synchronization in more detail. If the span count becomes 0, the thread that drops the count to zero returns it to the global allocator. Free spans on active huge pages are immediately available for recycling. When a line count drops to zero, the freeing thread updates the span's count.

A span with free lines and live lines is partially free. The global allocator recycles partially free spans only after the deadline of their huge page expires. It scans the huge page and adds any closed partially free spans to a list. When it assigns spans to a thread-local allocator, it marks them as open. A local allocator may have one or two open spans per LC: one initially partially free and one initially fully free.

Each span is exclusive to the requesting local allocator which only allocates objects of the lifetime it requested, regardless of the huge page's LC. When a block is full, the local allocator marks the block closed and releases it. Each time a span is open for allocation, it only receives one LC. Each time a partially free span is opened, it may however receive a different LC, mixing lifetimes. The LC of these objects will always be shorter than the LC of the huge page.

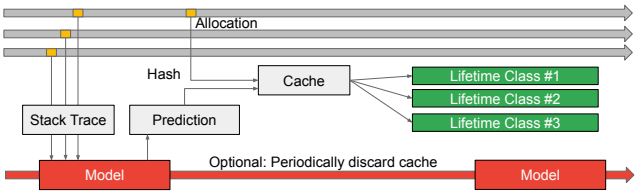


Figure 7. High-level overview of low-latency prediction. We use the model only when the hash of the current stack trace is not in the cache. Discarding cache entries periodically helps dynamically adapting to workload changes.

Llama bump-pointer allocates small objects into partially free spans until it encounters an occupied line or the end of the span. When it encounters an occupied line, it skips to the next free line(s). For *tiny* objects less than or equal to the line size (128 B), if the current line has insufficient free memory, it skips to the next free line which is guaranteed to be sufficient, wasting some memory. For other *small* objects (> 128 B), Llama limits line fragmentation using demand driven *overflow allocation* [10]. If a small (not tiny) object does not fit, the allocator instead obtains a second completely free *overflow* span from the global allocator for this object and any future such allocations. It thus avoids searching for $n > 1$ free contiguous lines or wasting free lines. A local allocator may thus have two spans per LC: one partially free and one overflow span. The local allocator prefers to allocate in the partially free span, but once exhausted, it will fill the overflow span before requesting a new span.

7 Low-Latency and Accurate Prediction

The allocator must predict object lifetimes quickly to meet latency requirements. TCMalloc allocation times are <100 cycles, but even a simple neural network takes microseconds. We therefore cache predictions. Figure 7 shows how at each allocation, we compute a hash of the return address, stack height and object size, and index a thread-local hashmap. Because stack traces have temporal locality, we expect the lookup will usually hit in the L1 cache. Prior work shows stack height identifies C/C++ stack traces with 68% accuracy [42]. We find adding object size increases accuracy. If the hash hits, we use the cached prediction. Otherwise, we run the compiled model which takes hundreds of μs (depending on the stack depth), and store the result in the cache.

When stack hashes with very different lifetimes alias or workloads change, prediction accuracy suffers. For example, if we store a hash for an allocation site that is predicted short-lived, but a second site, more common and long-lived, aliases, then Llama may allocate a large number of long-lived objects into short-lived block spans. We found that 14% of predictions disagreed with the currently cached value.

To address this problem, we periodically discard cached entries. Every, e.g., 1,000 cache hits, we run prediction again.

If the result agrees with the current entry, we do nothing. Otherwise, we set the cache entry to the maximum lifetime of the old and new prediction. We use maximum because the allocator is more resilient to under-predicted lifetimes than over-predicted lifetimes.

8 Implementation Details

Allocation size lookup. When freeing objects, we need to know their size. We use a 256-entry bitmap representing each block in a huge page. We set a bit to 1 if and only if the corresponding block is the last block occupied by an object. Given an address, we find the blocks it occupies by rounding it down to the closest block size and using the bitmap to find the next set bit. This approach does not work for the last object (which may span multiple huge pages). We therefore store a 64-bit value in the huge page metadata, which contains the size of the last object on the huge page. A similar approach tracks small objects that span lines, but since small objects cannot straddle spans, it needs only one byte to store the number of lines occupied by an object.

C-style malloc/free API. Our allocator is designed for C++, but supports legacy C code, which requires storing the precise allocation size to support `realloc` calls. If we encounter legacy malloc calls, we pad the object with a header that contains the object size.

Alignment. Our allocator handles alignment and aligns all objects to at least 8 B. The huge page allocator handles common alignments automatically, as blocks are 8 KB aligned. For larger alignments, we increase the allocation size as necessary and shift the start pointer to match the required alignment. When we search for a free gap in a page, we try gaps individually to find ones that fit the object with the correct alignment.

Lifetime region management. Above, we assume one 16 GB virtual memory region per lifetime class. Llama never reuses huge page virtual memory. Even after it frees a huge page, Llama still continues to use fresh virtual memory space if it needs to allocate another huge page in this region. This approach is practical because 64 bit architectures provide virtual address space that exceeds the physical address space per-process by orders of magnitude. If we run out of virtual memory in a region of a given lifetime, we allocate an additional 16 GB virtual memory region for this lifetime class. Llama manages these regions in an array. The OS only maps small and huge pages when the program accesses them and unmaps pages when the allocator releases them.

Locking. The main scalability bottleneck in Llama is a single lock in the global allocator that performs huge page and block allocations. We leave adding per-huge-page locks and concurrency to the global allocator to future work. Llama also uses synchronized reference counting. Each block span has at most one owner thread, which performs unsynchronized allocation and atomic reference count increments. Since different threads can free objects, span and line reference count increments and decrements must be synchronized (or queued in a buffer for later processing [48]). The thread that drops a span reference count to zero is responsible for freeing it. The owner of an open span increments its reference count by 1 when it acquires a span and decreases it by 1 when it releases it since no thread can free the span while an owner is still allocating into it. We apply the same technique to lines – the allocator increments the reference count for a line when it partially allocates into it and then decrements it when it moves on to the next line.

Potential optimizations include eliding the increment and decrement pair when an object crosses lines and deferral, similar to reference counting Immix [48]. We note that highly tuned allocators perform a large number of additional optimizations (such as prefetching, restartable sequences, hand-tuned assembly sequences [32]) that are missing from this research allocator.

Bootstrap allocator. Llama needs some basic functionality during initialization, such as querying the binary's symbol table. For prototyping, Llama uses a bootstrap allocator that handles initial allocations before executing the program. Our prototype uses TCMalloc as this bootstrap allocator. The memory usage reported in this paper consists of memory allocated by both allocators, including fragmented memory. Bootstrap memory is a small fraction of the heap. A full implementation would likely use a simpler bootstrap allocator.

9 Evaluation

We evaluate Llama on four workloads. Except for Redis, they are large production code bases:

Image Processing Server. A Google-internal production image processing server that filters and transforms images. We use synthetic inputs, but the fragmentation in our experiments is consistent with production.

TensorFlow. The open-source TensorFlow Serving framework [44] running the InceptionV3 [50] image recognition model. This workload exercises libraries with complex memory allocation behavior, such as the *Eigen* linear algebra library. It runs 400 batches of requests in a harness.

Data Processing Pipeline. A Google-internal data processing workload running word count on a 1 GB file with 100 M words. We run the entire computation in a single process, which creates very high allocator pressure, resulting in 476 parallel threads and 5M allocations per second.

Redis. The open-source Redis key-value store (v. 4.0.1) running its standard `redis-benchmark`, configured with 5K concurrent connections and 100K operations of 1000 B. We rename its `zcalloc` function to avoid a name collision.

The goal of the evaluation is to 1) demonstrate this approach is promising and works on large production code bases; 2)

Workload	Prediction Accuracy		Final Steady-state Memory			Fragmentation
	Weighted	Unweighted	TCMalloc	LLAMA	Live	reduction
Image Processing Server	96%	73%	664 MB	446 MB	153 MB	43%
TensorFlow InceptionV3 Benchmark	98%	94%	282 MB	269 MB	214 MB	19%
Data Processing Pipeline	99%	78%	1964 MB	481 MB	50 MB	78%
Redis Key-Value Store	100%	94%	832 MB	312 MB	115 MB	73%

Table 3. Summary of Model Accuracy and End-to-end Fragmentation Results

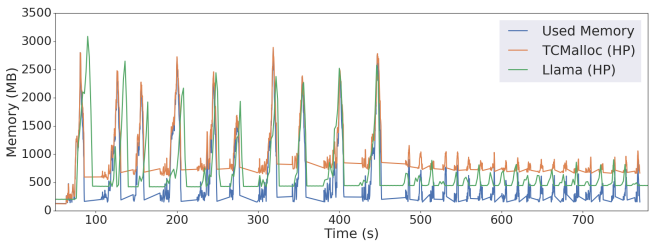
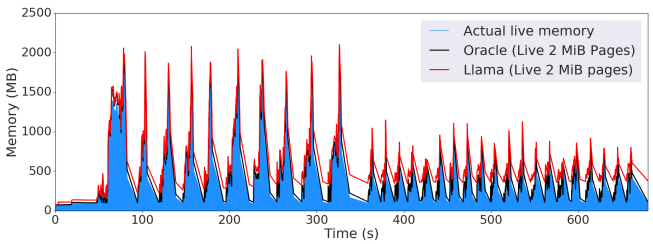
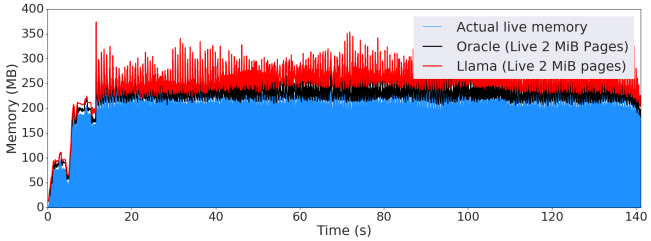


Figure 8. LLAMA reduces huge page (HP) fragmentation compared to TCMalloc on the Image Processing Server. TCMalloc numbers optimistically assume all free spans are immediately returned to the OS, which is not the case.



(a) Image Processing Server



(b) TensorFlow InceptionV3 Benchmark

Figure 9. LLAMA's memory consumption with perfect lifetime predictions (using traces) is close to an oracle and closely follows the live heap size.

understand trade-offs, such as the model's generalization abilities; and 3) characterize LLAMA. We use a workstation with a 6-core Intel Xeon E5-1650 CPU running at 3.60GHz with 64 GB of DRAM and Linux kernel version 4.19.37.

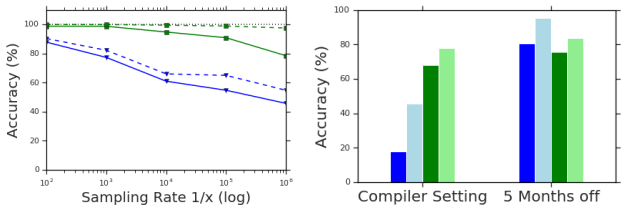
These workloads stress every part of our allocator. They use 10s to 100s of threads, a mix of C++ and C memory allocation, object alignment, have a large ratio of allocation to live objects, and a large amount of thread sharing. They frequently communicate objects between threads, causing the free lists to be "shuffled" and leading to fragmentation. We believe these workloads are representative of modern C/C++ server application. They stress the memory allocator significantly more than workloads used in some prior C/C++ memory manager evaluations, such as SPEC CPU. These patterns are similar to Java applications, illustrating the evolution of C/C++ applications and how they heavily rely on their memory managers.

9.1 End-to-end Evaluation

Table 3 shows end-to-end fragmentation improvements over TCMalloc for the four workloads (not from simulation), ranging from 19% to 78%. Figure 8 shows image processing server fragmentation as a function of time. Since vanilla TCMalloc does not support huge pages, we reconstruct the number of occupied and free huge pages from its bookkeeping information. This method is a lower bound because it does not take into account that TCMalloc does not immediately (or sometimes ever) release pages to the OS. TCMalloc's actual occupancy will be between this amount and the largest peak in the trace, depending on page release rate. Even when compared with the most optimistic variant, we eliminate 43% of the fragmentation introduced by TCMalloc for the image server (in steady state and at termination). Note these results include the memory overheads of our model.

The data processing pipeline represents a different kind of workload than the servers. While the heap size variation in servers results from changing request size patterns, the data processing pipeline's heap size varies based on its execution stages. Fragmentation occurs when long-lived outputs of a stage are allocated while the heap contains a large amount of temporary data from an active stage.

Redis illustrates the limitations of a PGO-based approach. Our model learns the difference between per-connection data (which is short-lived) and stored data (which is long-lived). However, Redis servers are often dominated by stored data and the lifetime of these objects is entirely determined by client requests and cannot be predicted. As such, Redis represents workloads where a PGO approach alone is limited. Redis implements a feature called *active defragmentation* that relocates its long-lived *stored-data*, giving the allocator an opportunity to compact memory and decrease fragmentation. Redis thus illustrates fragmentation is a large enough problem that the developers hand-coded a mitigation. However, this approach only supports Redis's stored-data data structure, and not other objects (e.g., session state). We hypothesize that a model can be effective when combined with



(a) Sampling Rate **(b)** Workload Variations

Figure 10. The lifetime model generalizes to unobserved allocation sites from different versions and compiler settings. Blue shows accuracy per stack trace, green weighted by allocations. Light/dotted data shows off-by-one accuracy.

this mechanism to only predict lifetimes of non-Redis objects. Further, if client requests have regularity (e.g., when Redis is used as a cache), the model might be able to learn this behavior as well.

To isolate the impact of the accuracy of lifetime predictions from Llama's memory management algorithm, we measure its effectiveness with perfect predictions. We link the allocator into our simulator and run it using pre-recorded traces with perfect lifetime information. Figure 9 shows that with a perfect lifetime oracle, the average fragmentation is less than 1.1× for both workloads. This result demonstrates that Llama succeeds at packing objects into huge pages.

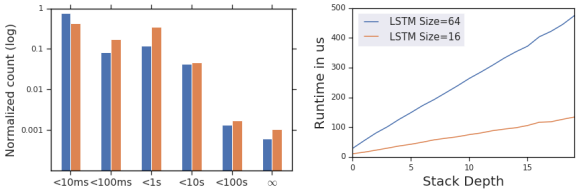
9.2 Model Evaluation

LSTM Model Generalization. Figure 10 shows accuracy remains high when training our model on one version of the image server and applying it to another. The same configuration in Table 2 shows almost no matching stack traces (i.e., a lookup table would not work). In contrast, the model achieves upwards of 80% accuracy when applied to the other revision, and increases to 95% when ignoring errors where the prediction is off by at most one lifetime class. We see an interesting effect for the non-optimized build. This example achieves few exact matches but higher accuracy for off-by-one errors. We hypothesize that because the non-optimized version of the code runs slower, lifetimes are consistently in a higher class than optimized code.

9.3 Sampling Effectiveness

We measure the overhead of sampling RPC latencies in the image processing server at an average of ≈5 %, but with large variations (1-8%). To evaluate if the sampled data and the full trace data we use elsewhere in the paper are consistent, Figure 11a shows the distribution of lifetime classes of full traces sub-sampled at 1:100 K, compared to the lifetime profiler's data. Note that this log-scale figure does not imply that the fractions of the different traces are the same, but that they are in the *same order of magnitude* for each of the classes, the accuracy the system needs.

To evaluate how many samples we need to construct an accurate model, we run our image processing workload 20



(a) Sampling Validation **(b)** LSTM Execution Time

Figure 11. Validation of sampling and compiled model execution latency for the image processing server.

times for a total of 2.3 B allocations, and sample each allocation with a particular probability ranging from 1:100 to 1:1,000,000. We then compare the resulting predictions to the original training data (Figure 10). Even when only sampling every millionth allocation, the model still produces the same output as the training data 80% of the time and almost 100% are off at most by one lifetime class. This demonstrates our model's ability to generalize.

9.4 Predictor Overheads

Latency. We next evaluate the computational performance of our model. Figure 11b shows the prediction latency with increasing stack sizes. We compare two different models to understand the trade-off space. The model we use throughout uses a 64-dimensional vector as the internal state of the LSTM. We compare to a smaller model with a 16-dimensional vector that can potentially store less information but executes more quickly. In practice, we would tune this parameter when we train an application-specific memory allocator.

Memory Consumption. We measure the memory consumption introduced by our predictor. First, our allocator loads the symbol map associated with the binary, which is 17 MB for the image processing server. Next, every instance of the model's internal buffers uses 58 KB (the number of instances limits the number of parallel threads performing prediction simultaneously). We use 64 of them (less than 4 MB of memory). Finally, the allocator maintains a map from symbols to tokens. We could fold this memory into the symbol table to eliminate most of this overhead. The allocator thus adds 56 MB for prediction for this workload, less than 2% of the maximum heap size. As we show in Section 9.1, Llama recoups this memory easily.

Stack hashing accuracy. For the image server, 95% of predictions hit in the cache, which shows stack hashing reduces model evaluations. To evaluate accuracy, we sample predictions and measure how often they disagreed with the cached value. They disagree 14% of the time, but only require updates to longer lifetime classes for 1.6% of allocation sites.

9.5 Lifetime Aware Memory Allocator Performance

We now characterize Llama's performance. While our research prototype is not highly tuned, we ensure its performance is sufficient to run the full benchmarks at reasonable

TCMalloc Fast path	8.3 ± 0.1 ns
TCMalloc Global allocator	81.7 ± 1.0 ns
Fast path (w/o prediction)	29.1 ± 0.9 ns
Without lines/recycling block spans	17.1 ± 0.8 ns
With 2 threads	28.6 ± 0.1 ns
With 4 threads	28.7 ± 0.1 ns
Fast path (prediction cached)	48.8 ± 0.1 ns
Fast path (run ML model, size=64)	144.6 ± 1.5 us
Global allocator (w/o prediction)	52.7 ± 2.9 ns
With 2 threads	274.5 ± 38.0 ns
With 4 threads	802.2 ± 75.0 ns
Global allocator (prediction cached)	88.0 ± 7.8 ns
Global allocator (run ML model, size=64)	143.8 ± 1.2 us

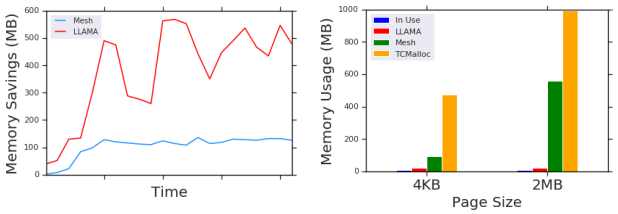
Table 4. Memory Allocator alloc+free Performance

speed. We now discuss the prototype's bottlenecks and how a production implementation could address them. We believe that none of these bottlenecks are fundamental.

Production memory allocators are highly tuned and applications are often co-optimized with a particular memory allocator. Allocator optimizations include rigorous tuning of every instruction on the fast path, software prefetch instructions, use of restartable sequences to reduce synchronization overheads, size class tuning, and fine-grained locking. In contrast, our allocator contains very few optimizations and the global allocator is protected by a central lock which is currently the main performance bottleneck. We also do not take advantage of sized deallocation in C++11. Compared to TCMalloc, which handles objects of up to 256 KB using the fast path, Llama's cut-off is 8 KB, causing a larger fraction of allocations to use the slow path. Kanev et al. describe many fast path optimizations in TCMalloc [32].

We use microbenchmarks to quantify the slowdown of allocation relative to TCMalloc for a number of common allocation paths. On average, allocation is currently 2 – 3× slower than TCMalloc. In practice, the memory allocator sees much less contention than in this stress test, and end-to-end slowdowns are less dramatic (Section 9.1). For example, the image server slows down ≈ 12.5% per query compared to TCMalloc. On the other end of the spectrum, the global lock is a problem under very high allocator pressure. For the data processing pipeline with 476 threads mapped to 6 physical cores and 5M allocations per second, Llama's performance degrades by 2.84× compared to a recent version of TCMalloc [26]. Note that TCMalloc is highly tuned and that this benchmark is limited by global synchronization in Llama and thus is particularly advantageous for TCMalloc.

The overheads in our allocator could be addressed. In the fast path, the main bottleneck is the atomic operations required to update object counts – these operations could be elided by operating entirely on thread-local counters and only writing them back when an open block span is released. In the slow path, the main bottleneck is the global lock. This is particularly pronounced when the number of threads



(a) Image Server Simulation **(b)** Microbenchmark

Figure 12. Llama reduces fragmentation compared to Mesh.

exceeds the number of physical cores. This lock could be replaced with a readers-writer lock for the list of active pages (which is mostly read-only) and a per huge-page lock that is only acquired when a page is updated. The list could also be implemented as a lock-free data structure. While these overheads mean that our research prototype is not production-ready, the focus of this work has been fragmentation and our prototype is suitable for evaluating it.

We also gather statistics to confirm that Llama's different behaviors and execution paths are actually exercised by our workloads. For the image processing server (spanning 130M allocations and 207 GB allocated), the allocator allocates 640 K block spans, observes expiring deadlines 1,011 times, and demotes huge pages 8,492 times, confirming the benchmarks exercise Llama's key features.

9.6 Comparison to Mesh [46]

Fragmentation induced by long-lived objects allocated at peak memory usage is fundamental to most memory allocators, since avoiding this fragmentation requires the allocator to know at allocation time which objects are long-lived. As such, strategies such as size class tuning or best fit allocation do not address this source of fragmentation.

A recent proposal, Mesh [46], takes a different approach and reduces fragmentation by combining (*meshing*) virtual pages with non-overlapping objects into the same physical page using copying and page remapping. As such, Mesh has the potential to address fragmentation caused by long-lived objects. For example, Mesh reduces fragmentation in Firefox by 16%. We compare Llama to Mesh. A challenge is that Mesh's probabilistic guarantees rely both on random allocation and on small 4 KB pages. The paper states that Mesh is not designed to work with huge pages. We thus compare with Mesh first on the Image Server using huge pages and then — using a microbenchmark that simulates varying heap sizes — on both small and huge pages.

Image Server (Simulation). For the image server, we use our simulator to compute occupancy bitmaps throughout the execution and then give them as input to Mesh's analysis scripts to compute meshing opportunities, using the "greedy" mesher. Figure 12a shows Llama saves memory between a factor of 2 to 5 compared to meshing throughout the execution of the image server.

Microbenchmark. We compare Llama to Mesh and TCMalloc on small and huge pages using a microbenchmark that mimics varying heap size. The microbenchmark allocates a sequence of short-lived 64 B objects and fluctuates between a 1 MB and a 1 GB heap size. Every 10,000 allocations, it allocates a long-lived object, for a total of 5 MB of long-lived data spread out evenly across the virtual address space. It represents a stress-test for the type of fragmentation that Llama and Mesh address. At the end of the execution, all but the long-lived objects are freed and we report live pages in Figure 12b for small and huge pages.

Figure 12b shows vanilla TCMalloc incurs high fragmentation. With 2 MB pages, it frees almost no pages. With 4 KB pages, it frees about half of the memory. Note that not all this fragmentation is caused by live objects, as TCMalloc has cells held in caches and free lists. In contrast, Mesh (only counting memory in MiniHeaps) reclaims most of the fragmentation in the 4 KB pages case (91.7 MB), as intended. However, when looking at 2 MB pages, this memory becomes 558 MB, confirming that Mesh works well with 4 KB pages but not 2 MB pages. Meanwhile, our allocator only uses 22 MB in both cases when supplied with correct lifetime predictions, not accounting for the bootstrap allocator or any models.

These experiments show Mesh is highly effective for addressing fragmentation with 4 KB pages. While Mesh alone does not solve the fragmentation problem for huge pages, we believe that our approach can be combined with Mesh to further reduce fragmentation. When Llama encounters long-lived blocks on mostly empty pages, the global allocator could avoid the corresponding locations on other pages, making it more likely that these pages can be meshed in the future. This approach could likely use the same bitmap-based mechanism already used by Llama.

10 Discussion

Extension to other properties. Our model predicts lifetimes, but the allocator can benefit from other properties, e.g., whether or not an object will be freed by the same thread that allocated it. This information is useful because it allows us to allocate objects that stay within the same thread in the same block span, which reduces synchronization and improves performance. As with the page allocator, we need to consider mispredictions. As we are using atomic operations to update the reference count, correction is simple. If the prediction was correct, performance improves from reduced synchronization. For incorrect predictions, we incur a minor performance loss by having to synchronize on the cache line, but these are rare if predictions are mostly correct. A more generalized predictor could inform various other memory allocation strategies (e.g., based on object sizes, alignment, freeing thread, etc.) and learn which strategy to pick for each allocation. The strategies themselves could be determined by simulating different scenarios and using techniques such as Bayesian optimization to choose among them [22].

Improving accuracy and reducing prediction costs. The cost of our model could be significantly reduced. Currently, we need to look up each stack pointer within a trace in the binary's symbol table, tokenize it, multiply the results with the embedding matrix, and feed it into the model. While we cache tokenizations of symbols, these lookups incur additional delays at runtime. Instead, we could precompute all of these steps at compile-time when we build the symbol table, including the execution of the part of the model that multiplies tokens with the embedding matrix. This approach is a form of partial evaluation.

We may also be able to reduce the latency of our model by not feeding sequences of tokens into the model but by learning an embedding for entire stack frames. This approach may reduce the LSTM length by an order of magnitude, and would be particularly effective when combined with partial evaluation. A final optimization is to memorize our hash tables across runs to avoid startup overheads.

General implications for ML for Systems. We believe that many issues this paper addresses for using ML apply to other systems problems, such as sizing queues and data structures (e.g., vectors and maps). These predictions are also latency-sensitive, can benefit from calling context, and need to tolerate mispredictions. We think a general approach to system resource management problems is to decompose the problem into a supervised learning problem that can be solved by learning from profiling data and a conventional algorithmic solution for handling mispredictions.

11 Conclusion

We show that modern ML techniques can be effectively used to address fragmentation in C++ server workloads that is induced by long-lived objects allocated at peak heap size. We use language models to predict lifetimes *for unobserved allocations sites*, a problem unexplored in prior lifetime prediction work. We introduce Llama, a novel memory manager that organizes the heap using huge pages and lifetime classes, instead of size classes. Llama packs objects with similar lifetime together in the blocks of a huge page, tracks actual lifetimes and uses them to correct for mispredictions. It limits its fragmentation by filling gaps created by frees in blocks and their lines with shorter-lived objects. In this context, this work solves challenges related to applying ML to systems problems with strict resource and latency constraints.

Acknowledgements. We would like to thank our shepherd Harry Xu for his help improving the paper. We would also like to thank Ana Klimovic, Chris Kennelly, Christos Kozyrakis, Darryl Gove, Jeff Dean, Khanh Nguyen, Mark Hill, Martin Abadi, Mike Burrows, Milad Hashemi, Paul Barham, Paul Turner, Sanjay Ghemawat, Steve Hand and Vijay Reddi, as well as the anonymous reviewers, for their feedback. Finally, we would like to give credit to Rebecca Isaacs and Amer Diwan for the initial implementation of the stack hashing mechanism.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

[4] David A. Barrett and Benjamin G. Zorn. 1993. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/155090.155108

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[6] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/582419.582421

[7] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc.

[8] Stephen Blackburn, Richard E. Jones, Kathryn S. McKinley, and J. Eliot B. Moss. 2002. Beltway: Getting Around Garbage Collection Gridlock. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 153–164.

[9] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and realities: the performance impact of garbage collection. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004*. 25–36. https://doi.org/10.1145/1005686.1005693

[10] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. 22–32. https://doi.org/10.1145/1375581.1375586

[11] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 342–352. https://doi.org/10.1145/504282.504307

[12] Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. 2010. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada*. 13–24. https://doi.org/10.1145/1806596.1806599

[13] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 28, 16 pages. https://doi.org/10.1145/3302424.3303988

[14] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. 2010. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 363–375.

[15] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. 1992. Profile-guided automatic inline expansion for C programs. *Software: Practice and Experience* 22, 5 (1992), 349–369. https://doi.org/10.1002/spe.4380220502

[16] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*. 105–117.

[17] David A. Cohn and Satinder P. Singh. 1997. Predicting Lifetimes in Dynamically Allocated Memory. In *Advances in Neural Information Processing Systems 9*, M. C. Mozer, M. I. Jordan, and T. Petsche (Eds.). MIT Press, 939–945. http://papers.nips.cc/paper/1240-predicting-lifetimes-in-dynamically-allocated-memory.pdf

[18] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *ACM International Symposium on Memory Management (ISMM)*. 37–48. https://doi.org/10.1145/1029873.1029879

[19] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[20] Kunihiko Fukushima. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 36, 4 (1980), 193–202.

[21] Sanjay Ghemawat and Paul Menage. 2009. Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[22] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1487–1495.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[24] Google. 2020. C++ Arena Allocation Guide. https://developers.google.com/protocol-buffers/docs/reference/arenas

[25] Google. 2020. pprof. https://github.com/google/pprof

[26] Google. 2020. TCMalloc. https://github.com/google/tcmalloc

[27] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/3173162.3173194

[28] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[29] Jipeng Huang and Michael D. Bond. 2013. Efficient context sensitivity for dynamic analyses via calling context uptrees and customized memory management. In *ACM Conference on Object-Oriented Programming Languages and Systems (OOPSLA)*. 53–72.

[30] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. 2004. Dynamic object sampling for pretenuring. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*. 152–162. https://doi.org/10.1145/1029873.1029892

[31] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[32] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 33–45. https://doi.org/10.1145/3037697.3037736

[33] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2015. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. 66–78. https://doi.org/10.1145/2749469.2749471

[34] Sang-Hoon Kim, Sejun Kwon, Jin-Soo Kim, and Jinkyu Jeong. 2015. Controlling physical memory fragmentation in mobile systems. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA, June 13-14, 2015*. 1–14. https://doi.org/10.1145/2754169.2754179

[35] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6980

[36] Bradley C. Kuszmaul. 2015. SuperMalloc: a super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA*. 41–55. https://doi.org/10.1145/2754169.2754178

[37] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[38] Doug Lea and Wolfram Gloger. 1996. A memory allocator.

[39] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural Computation* 1, 4 (1989), 541–551.

[40] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *ACM International Conference on Computer Architecture (ISCA)*. 301–312. https://doi.org/10.1109/ISCA.2014.6853237

[41] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[42] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. 2009. Inferred Call Path Profiling. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 175–190. https://doi.org/10.1145/1640089.1640102

[43] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 675–690. https://doi.org/10.1145/2694344.2694345

[44] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).

[45] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[46] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: Compacting memory management for C/C++ applications. In *ACM Conference on Programming Language Design and Implementation(PLDI)*. 333–346. https://doi.org/10.1145/3314221.3314582

[47] Chuck Rossi. 2017. Rapid release at massive scale. https://engineering.fb.com/web/rapid-release-at-massive-scale/.

[48] Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking off the gloves with reference counting Immix. In *ACM Conference on Object-Oriented Programming Languages and Systems (OOPSLA)*. 93–110. https://doi.org/10.1145/2509136.2509527

[49] Darko Stefanovic, Kathryn S. McKinley, and J. Eliot B. Moss. 1999. Age-Based Garbage Collection. In *ACM SIGPLAN Conference on Object-Oriented Programming Languages and Systems (OOPSLA)*. 370–381. https://doi.org/10.1145/320385.320425

[50] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[51] David M. Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*. 157–167. https://doi.org/10.1145/800020.808261