

Combining Machine Learning and Lifetime-Based Resource Management for Memory Allocation and Beyond

By Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel

Abstract

Memory management is fundamental to the performance of all applications. On modern server architectures, an application's memory allocator needs to balance memory utilization against the ability to use 2MB huge pages, which are crucial for achieving high performance. This paper shows that prior C++ memory allocators are fundamentally limited because optimizing this trade-off depends on the knowledge of object lifetimes, which is information allocators lack.

We introduce a two-step approach to attain high memory utilization in huge pages. We first introduce a novel machine-learning approach that predicts the lifetime of freshly allocated objects using the stack trace at the time of allocation and treats stack traces as natural language. We then present a fundamentally new type of memory allocator that exploits (potentially incorrect) object lifetime predictions to achieve high memory utilization at *full huge page usage*. In contrast to prior memory allocators that organize their heap around size classes and free lists, our allocator organizes the heap based on predicted *lifetime classes* and adjusts to mispredictions on the fly. We demonstrate experimentally that this learned lifetime-aware memory allocator (LLAMA) reduces fragmentation with huge pages by up to 78%.

Our approach gives rise to a new methodology for applying ML in computer systems. In addition, similar space-time bin packing problems abound in computer science and we discuss how this approach has applications beyond memory allocation to a wide range of problems.

1. INTRODUCTION

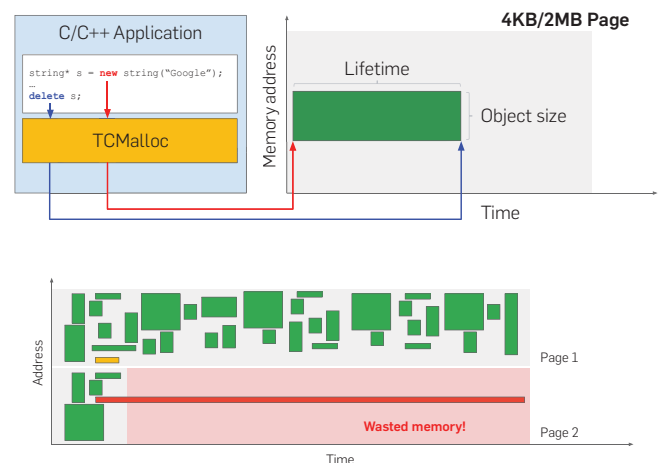
Memory management is a decades-old research area²⁴ that is fundamental to the performance of all applications. On modern architectures, memory managers determine a workload's ability to use 2MB (and 1GB) *huge pages* instead of traditional 4KB pages. The use of huge pages is crucial for performance on modern servers since they substantially reduce the cost of address translation by producing a wider reach in Translation Lookaside Buffers (TLB), reducing misses on the CPU's critical path.⁵

Current huge page-aware memory managers¹³ trade-off huge page usage with memory utilization, breaking up huge pages when they become inefficient. Figure 1 visualizes the source of this trade-off: When a C++ program allocates memory, it calls into a memory allocator library

(e.g., TCMalloc¹³), which places the object at a particular address in memory until the program deletes it. The object may not move. The memory allocator has two goals: use as little memory as possible while placing as many objects as possible into consecutive 2MB ranges of memory, which enables the utilization of huge pages. The latter is crucial for performance since these 2MB ranges can be represented by one (instead of 512) TLB entries in the CPU, reducing the number of TLB misses that slow down the application.

Challenges arise because memory allocators can only request memory from the operating system at the granularity of a page. As long as a page contains at least one object, it cannot be returned to the operating system. Poor object placement can therefore lead to mostly empty huge pages assigned to an application (Figure 1). These huge pages

Figure 1. Overview of C++ memory allocation and how long-lived objects lead to wasted memory. Had the red and yellow objects been swapped, page 2 could be freed and the memory footprint would have halved.



The original version of this paper is entitled “Learning-based Memory Allocation for C++ Server Workloads” and was published in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

represent wasted memory or need to be broken up into 4KB pages, reducing performance.

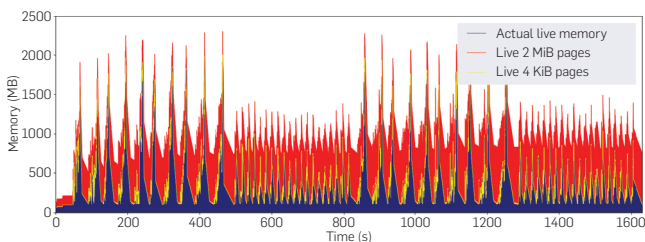
We demonstrate that wasted pages present challenges for long-running server workloads whose memory footprints shrink and grow over time depending on user demand. Many web services exhibit such highly variable memory consumption.¹⁶ Most objects allocated by a server are short-lived but a small fraction lives for a very long time, such as session state, logs, or in-memory data. This lifetime distribution is not a major problem with a 4KB page size: If, conservatively, 99.99% of objects are short-lived and their average size is 64B, then using 4KB pages, the probability that any given page contains a long-lived object is less than 1% ($1 - (0.9999)^{4096/64}$). With 2MB huge pages, the corresponding probability is 96%.

When the footprint of a server workload shrinks, most of its huge pages cannot be released back to the operating system because they contain at least one long-lived object. Figure 2 shows this effect for a production image processing service on a synthetic workload.

Since C++ cannot move objects, solving this problem depends fundamentally on reasoning about object lifetimes and grouping objects with similar lifetimes together. In practice, the allocator thus needs to use the information it has at the time of allocation to make a placement decision that avoids long-lived objects being equally spread across pages. The most important information is the current stack trace at the time of allocation, which is called the *allocation context*. Figure 3 shows an example of such a context and the information it contains.

Prior work on compiler and language-runtime optimization also leverage allocation contexts. Commonly, these approaches collect *profiles* of allocation contexts with associated measurements and then leverage these profiles at run time. Profiles are collected during execution (online) or in a separate profiling run (offline). Allocation context-driven optimizations are often opportunistic⁹—they do not need to provide full *coverage* and make a prediction for every possible context, but yield improvements in cases where they can. In some settings, mispredictions may be corrected (e.g., garbage collectors can move objects). In contrast, we need to be able to make predictions for every possible context we may encounter, since a single long-lived allocation may “cost” up to 2MB and errors accumulate on

Figure 2. 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 (red) and increasingly more over time.



long-running servers. This requirement creates a set of new challenges:

- Online profiling is challenging because of overheads. Full-context profiling adds 6% overhead,^{7,20} which can be more than memory allocation itself.¹⁴ Sampling⁹ is sufficient for opportunistic optimizations but does not provide the full coverage we need.
- Offline profiling is challenging because exercising all possible application behavior ahead of time is infeasible and servers are configured in myriad ways with different libraries. Lifetimes are particularly difficult to profile since they require observing both an allocation and a deallocation event. We show that in practice, offline profiling does not provide full coverage either.

This paper addresses these problems by sampling a *subset* of allocation contexts and using machine learning (ML) to generalize from these contexts to previously unobserved contexts. In particular, our novel treatment of symbolized allocation contexts (stack traces) as natural language produces a model that extracts the meaning of function names and how they appear in stack traces. The model accurately predicts object lifetimes, even for previously unobserved contexts.

This new ability to predict object lifetime classes for every allocation inspires LLAMA (learned lifetime-aware memory allocator), a fundamentally new lifetime-predicting allocator design for huge pages that substantially reduces fragmentation on C++ servers. On allocation, LLAMA predicts N *lifetime classes*, where each class differs by an order of magnitude ($\leq 10\text{ms}$, 100ms , 1s , 10s , etc.). LLAMA organizes the heap by assigning each huge page to a lifetime class. It subdivides huge pages into blocks and lines, where each block's

Figure 3. An example of an altered but representative allocation context, with colored tokens. A string (1) is allocated within a protocol buffer function (2–6) as part of the initialization (9–12) of a larger system (7–8).

```

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

```

lifetime is predicted less than or equal to its huge page. LLAMA handles mispredictions by observing *actual* object lifetimes and using them to reclassify huge pages.

Figure 4 shows an overview of the approach. A subset of allocation lifetimes is sampled from prior runs and versions of a workload. A model is trained against these samples to provide predictions for *all* allocations. This model is compiled into the application for use by the novel LLAMA memory manager algorithm. We also introduce a new caching approach to make these predictions fast.

This paper makes contributions to memory management and the emerging area of ML for Systems.¹⁷ While many prior uses of ML in computer systems focused on tuning existing heuristics with ML, we instead use ML to reconsider the algorithmic context of memory management altogether. Our approach delivers the first allocator that substantially reduces fragmentation for modern C++ server workloads compared to a free-list allocator and *only* uses huge pages.

Since publication, this work has inspired a general methodology for leveraging ML in systems, with applications in operating systems and computer architecture.¹⁷ Instead of learning a systems problem end-to-end, cheap ML techniques are used to predict a previously unknown property (in this case, object lifetimes) and these predictions are then used to fundamentally redesign the algorithm around leveraging this property while tolerating mispredictions. Some insights from this work have also inspired separate (non-ML) TCMalloc optimizations that are deployed in production, leading to an estimated 1% throughput improvement across Google's fleet.¹⁸

While this paper focuses on C++ memory allocation, the algorithm we present applies to any *space-time bin packing problem*: Items (in our case, objects) are assigned to resources (in our case, pages) and a resource can only be released once all items within it have disappeared. Often, the items' lifetimes are predictable, which enables the use of the LLAMA algorithm. For example, we showed in a later paper that file lifetimes in storage systems are predictable.²⁵ The general LLAMA approach applies to areas such as storage systems, OS process management, and potentially operations research.

2. OBJECT LIFETIME PREDICTION

This section describes how we predict the lifetimes of C++ objects at the time of allocation, which has several challenges: (1) Lifetime depends on the entire calling context at allocation, not only the *allocation site* where the allocator was called. (2) The overhead of online profiling is

impractical because it costs 6% CPU performance,^{7,20} which would be more than allocation alone.¹⁴ (3) Full coverage of calling contexts and perfect accuracy are not achievable with offline profiling. Because servers evolve and are configured in myriad ways with different libraries, an offline profiler will thus only ever see a subset of contexts.

We address overhead and coverage challenges by sampling a subset of allocations across multiple executions (Section 2.1). 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. Sampling is suitable for both server applications in datacenters and multiple runs of a popular application (e.g., a web browser) on a client.

Sampling may not observe all allocation contexts and we must combine samples from a heterogeneous set of different software versions, while the code bases are constantly updated. Our solution uses ML on observed samples of *tokenized* (subdivided) contexts to predict object lifetimes. We train a supervised model (Section 2.2) that maps from calling context to lifetime and generalizes to unseen contexts.

Another challenge is to perform prediction without significant overhead. For example, TCMalloc's allocation fast path is 8.3ns (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 400ns. We address this problem with a hashing-based cache (Section 2.3) that identifies 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 next explain each component in more detail.

2.1. Sampling-based data collection

Always-on collection of allocation lifetimes incurs a substantial overhead. For example, stack tracing adds 14% end-to-end overhead, and writing to disk further increases the cost, making continuous profiling infeasible in production. We thus introduce a cheap sample-based continuous profiling mechanism and implement it in TCMalloc,¹³ similar to other production profiling tools.¹⁴ Our sampling approach periodically connects to servers (for a short duration such as ≈ 5 min) and samples a subset of all memory allocations within the process. Each sample includes a stack trace, object size, and address at allocation and deallocation time.

We implement this profiler with TCMalloc *hooks* that are called periodically, based on the number of allocated bytes. These hooks incur virtually no overhead when they are disabled. We also assign each sampled object an identifier at allocation time and match it at deallocation time to

Figure 4. Overview of our ML-based allocator.

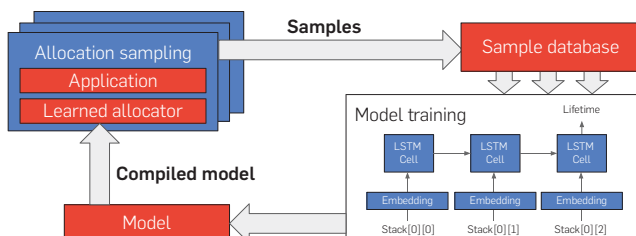


Table 1. Timescale comparisons.

TCMalloc fast path (new/delete)	8.3ns
TCMalloc slow path (central list)	81.7ns
Capture full stack trace	396ns \pm 364ns
Look up stack hash (Section 2.3)	22.5ns

compute lifetimes. For each sampled allocation, we keep a running tally of the distribution of lifetimes, by storing the maximum, minimum, count, sum, and sum of squares. We calculate the mean and variance of the lifetimes during post-processing. At the end of a sampling period, we store the result in a protocol buffer for later analysis using pprof.¹¹

2.2. Lifetime prediction model

Our sampling approach collects allocation contexts and their associated lifetimes. The simplest way to use these samples would be to store the values in a lookup table that maps allocation contexts to a lifetime class. Within a binary, an allocation context can be stored as a sequence of 64-bit pointers representing the locations of the instructions on the call stack. Building this table online is prohibitive due to the large overheads of always-on profiling. We, therefore, use data from prior executions of the application.

However, stack traces are brittle when used across executions. Even stack traces from the exact same binary may differ due to address layout randomization. Using symbol information, it is possible to compare stack traces based on 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 shows that a lookup table would not be suitable, particularly since our predictor needs to provide a high-quality prediction for *every* allocation context, which includes unseen contexts.

To address this problem, we design an ML-based predictor that learns calling contexts of tokenized class and method names to produce accurate predictions for unobserved contexts. We train this model using supervised learning. Training data is generated by grouping samples by allocation context and calculating the distribution of observed lifetimes for each context. We use the 95th percentile T_{95}^i of observed lifetimes of context i to assign a label $L_i \in \{1, \dots, 7, \infty\}$ such that $T_{95}^i < T(L_i) = (10)^{L_i}$ ms. Objects the program never frees get a special long-lived label ∞ . This produces lifetime classes of 10ms, 100ms, 1s, 10s, 100s, 1000s, ≥ 1000 s, and ∞ . Our model classifies stack traces according to these labels. To ensure our model assigns greater importance to stack traces that occur more often,

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

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%)

we weigh 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 at short timescales. We experimented with logical time measured in bytes, but believe wallclock time works better because (1) our lifetime classes are very coarse-grained (10 \times) and absorb variations, (2) if the speed difference between environments is uniform, nothing changes (lifetime classes are still a factor of 10 \times apart). Meanwhile, variations in application behavior make the bytes-based metric very brittle over long time ranges. For example, in our image server, the sizes of submitted images, number of asynchronous external events, etc. dilate logical time.

We use a model similar to text models. In recent years, there has been an explosion of work that learns text representations of code,² and our paper represents an example of this approach. In contrast to much of this work, we use code to reason about *dynamic* program properties rather than static code, an area that has seen less attention.⁸

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 IDs. One special ID is reserved for unknown or rare tokens, denoted as UNK. The table size is a configuration parameter (for example, 5000 covers the most common tokens).

We use a long short-term memory (LSTM) recurrent neural network model.¹² LSTMs are typically used for sequence prediction, for example, 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 perceptrons¹⁰ or convolutional neural networks¹⁵ 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 3 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) 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.

To learn these patterns, our model 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 5). 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 embeddings¹⁹ 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 incorporate more details of the underlying program, for example, Graph Neural Networks trained on program code.³

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.

2.3. Speeding up predictions

The allocator must predict object lifetimes quickly to meet latency requirements. TCMalloc allocation times are <100 cycles—recording the complete calling context and invoking even a simple neural network takes microseconds, and both are thus too costly. Table 1 shows recording the calling stack for an allocation alone can take an order of magnitude longer than the allocation. We solve these problems by cheaply caching predictions, using a hash of values that are already in registers. We cache predictions as shown in Figure 6 by computing a hash of the return address, stack height and object size, and indexing a thread-local hashmap. Prior work shows that stack height identifies C/C++ stack traces with 68% accuracy.²⁰ We add object size to increase the accuracy further. If the hash hits, we use the cached prediction. Otherwise, we run the compiled model, which takes hundreds of ms, and store the result in the cache.

When stack hashes with very different lifetimes alias or workloads change, prediction accuracy suffers. We found that 14% of predictions disagreed with the currently cached

value. To address this problem, we periodically discard cached entries. Every, for example, 1000 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 predictions. We use maximum because the allocator is more resilient to over-predicted lifetimes than under-predicted lifetimes.

3. LIFETIME-AWARE ALLOCATOR

This section introduces LLAMA, 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,^{13,24} 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. LLAMA is a fundamentally different approach to heap management, although its hierarchical heap organization has similarities to Immix.⁶ LLAMA is an untuned research prototype but demonstrates the potential of a lifetime-based memory allocation approach.

3.1. High-level structure

LLAMA organizes the heap into huge pages. To limit fragmentation, we divide huge pages into 8KB blocks and track their liveness. LLAMA assigns each active huge page one of N lifetime classes (LC), separated by an order of magnitude (e.g., 10ms, 100ms, 1000ms, ..., ∞).

LLAMA’s *global* allocator manages huge pages and their blocks. It acquires and releases huge pages from the OS as needed. It directly manages large objects (≥ 8 KB), placing them into contiguous free blocks in partially free huge pages or new huge pages. Small objects are handled by thread-local allocators that request ranges of blocks from the global allocator. A huge page may contain large and small objects. We will first describe the global allocator’s algorithm for large objects and then describe the handling of small objects in Section 3.5.

3.2. Lifetime-based huge page management

LLAMA stores a small amount of metadata for each huge page. Huge pages have three states: *open*, *active*, and *free*. Open and active huge pages are *live* and consume 2MB of memory each. Each huge page has an associated LC and only one huge page per LC is open at a time. While a huge

Figure 5. LSTM-based model architecture.

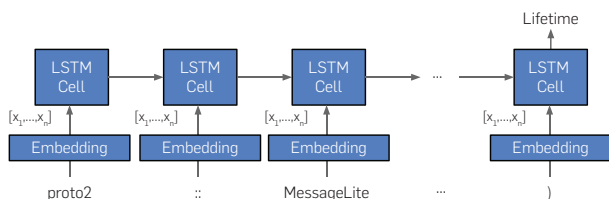
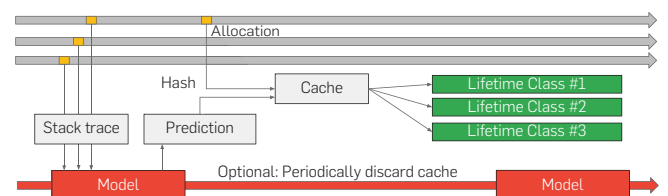


Figure 6. High-level overview of low-latency prediction. We use the model only when the hash of the current stack trace is not in the cache. Discarding cache entries periodically helps dynamically adapting to workload changes.



page is open, LLAMA only assigns its blocks to objects with the same predicted LC as the huge page. LLAMA transitions a huge page from open to active after filling all its constituent blocks for the first time. The huge page remains active for the rest of its lifetime. A huge page is free when all its blocks are free and is immediately returned to the OS.

All blocks on a huge page are *free* or *live*; and *residual* or *non-residual*. These properties are tracked via bitmaps. When blocks on an *open huge page* are assigned, these blocks are marked as *residual*, which means that they 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.

LLAMA initially places objects in the open pages corresponding to their predicted LC and transitions these pages from open to active once they are full. At this point, the huge page contains residual blocks and maybe free blocks. Figure 7 shows a simple example with three lifetime classes, separated by orders of magnitude. A large number of initial allocations are placed in open pages (Figure 7a), including a large object in huge pages 11 and 12. Figure 7b then shows many frees, which cause LLAMA to return free huge pages 2 and 6 to the OS.

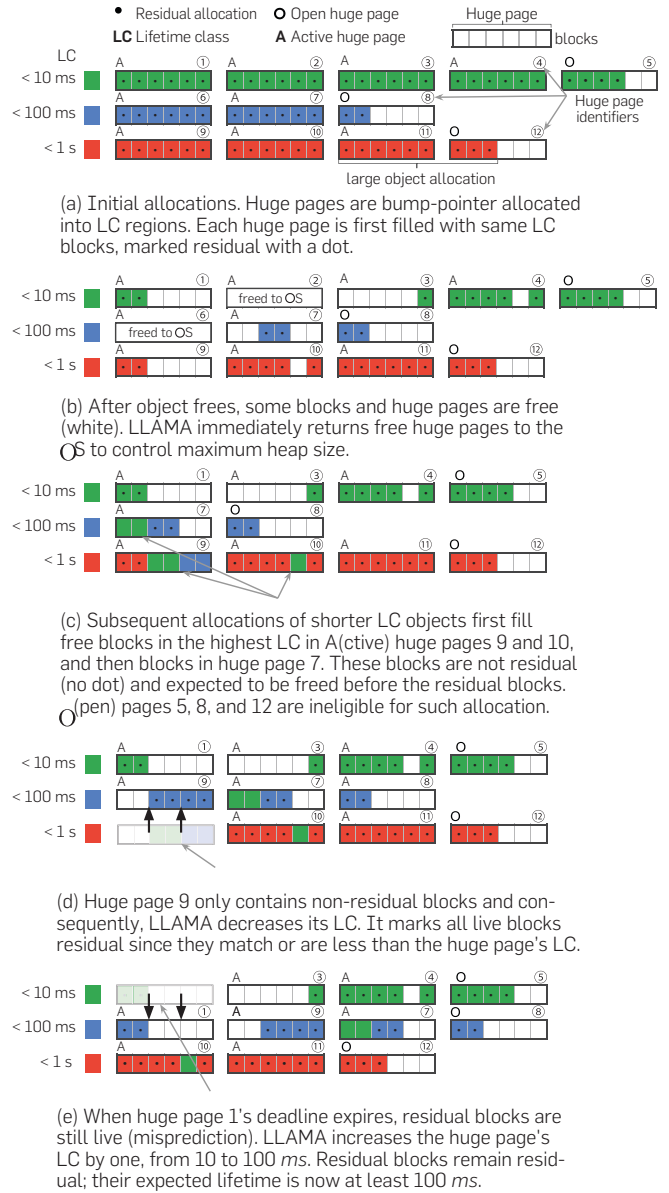
3.3. Recycling blocks to limit fragmentation

As shown in Figure 7b, active huge pages contain free blocks and live residual blocks of the same LC. If the free blocks were never reused, the allocator would waste significant amounts of memory to fragmentation. LLAMA limits fragmentation by aggressively *recycling* such free blocks for objects in shorter LCs. 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 marked as non-residual, as illustrated in Figure 7c. 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, all objects on a huge page with lifetime class LC will be freed within $1.1 \times LC$. All residual objects have a lifetime of at most LC and all non-residual objects are of at most the next-lower lifetime class, that is, $0.1 \times LC$. Because lifetime classes are separated by an order of magnitude, the allocator may reuse the non-residual blocks many times while the longer-lived objects on the huge page are in use, reducing the maximum heap footprint.

For example, given the heap state in Figure 7b and a request for a two-block large object with $lr < 10ms$, the global allocator allocates it into huge page 7 with $LC < 100ms$ and marks the blocks non-residual, as illustrated in Figure 7c. Once the program has freed all the objects on residual blocks within a huge page, all remaining (non-residual) blocks have a lifetime class at least one less than the huge page's current lifetime class. At this point, the huge page is reclassified as the next-lower lifetime class and all the current live blocks are set to residual (Figure 7d). Allocation then proceeds as before, filling the gaps between these blocks with even shorter-lived blocks and repeatedly reducing the lifetime class of the page until it is free.

Figure 7. LLAMA's logical heap organization with three lifetime classes ($< 10ms$, $< 100ms$, $< 1s$). Each live huge page is A(Active) or O(Open) and divided into blocks. Block color depicts predicted LC or free (white). Residual blocks are marked with a dot. Deadlines and lines are omitted.



3.4. Tolerating prediction errors

Since not all predictions made by the model are correct, LLAMA needs the ability to handle both over-predicted and under-predicted lifetimes.

Over-predicted lifetimes are handled using the mechanism that reclassifies huge pages once all residual objects are gone. For example, if residual blocks are freed before their lifetime has expired, the page will be reclassified earlier, which may result in it freeing up earlier.

Under-predicted lifetimes are more difficult. For example, if a huge page in the lowest lifetime class contains a long-lived object, it can result in a large amount of fragmentation since the page does not become free and also

cannot be used for allocating new objects. 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:

$$deadline = current_timestamp + K \times LC_{Huge\ Page}$$

When LLAMA changes the LC of a huge page, it assigns the huge page a new deadline using the same calculation. The intuition is that when all predictions are correct, the page would be free after $1.1 \times LC$. If we observe that a huge page has been active without reclassification for much longer than this (e.g., setting $K = 2$), we know that one or more of its constituent blocks were under-predicted.

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. The huge page remains in the *active* state. Figure 7e depicts this case. The residual blocks in huge page 1 outlive their deadline and LLAMA increases its LC to 100ms. A huge page may also contain non-residual blocks, which are left unchanged. If blocks live for even longer than 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.

3.5. Handling small objects

LLAMA achieves scalability on multicore hardware by using mostly unsynchronized *thread-local* allocation for small objects ($\leq 8KB$). The global allocator gives 16KB *block spans* to local allocators upon request. Local allocators hold one or two block spans for each LC. LLAMA further subdivides block spans into 128 B *lines* and *recycles* lines in partially free block spans for small objects. It tracks line and block liveness using *counters* that describe how many objects live within this span or line.

Small objects occupy one or more contiguous lines within the same span. Once a span is closed (filled at least once), subsequent frees may create a fully or partially free span. Fully free spans are returned to the global allocator. Partially free spans are recycled, but only after the deadline of their huge page expires. On huge page expiration, the global allocator scans the huge page and adds any closed partially free spans to a list, to be reassigned to thread-local allocators in the future. When a span is assigned to a thread-local allocator, it is marked as open.

A local allocator may have one or two open spans per LC: one initially partially free and one initially fully free. LLAMA sequentially 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). If an object still does not fit, the allocator uses the fully free span, similar to Immix.⁶

4. EVALUATION

We evaluate LLAMA on four workloads. Except for Redis, they are large production code bases. Experiments are run

in a research setup on a workstation with a 6-core Intel Xeon E5-1650 CPU running at 3.60GHz with 64GB of DRAM and Linux kernel version 4.19.37.

Image Processing Server. A Google-internal production image processing server that filters and transforms images, using synthetic inputs for measurement that produce fragmentation consistent with production.

TensorFlow. The open source TensorFlow Serving framework²¹ running the InceptionV3²³ 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. While running an old model, the benchmark is representative of modern workloads as well.

Data Processing Pipeline. A Google-internal data processing workload running word count on a 1GB file with 100M 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 (4.0.1) running its standard redis-benchmark, configured with 5K concurrent connections and 100K operations of 1000B.

These workloads stress every part of our allocator. They use tens to hundreds of threads, a mix of C++ and C memory allocation, object alignment, 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 applications. 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 now heavily rely on their memory managers.

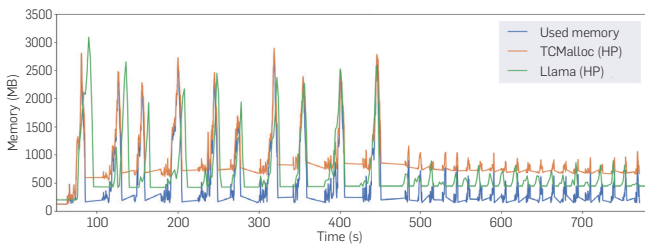
The goal of the evaluation is to (1) demonstrate that the LLAMA approach is promising and works on large production code bases; (2) understand trade-offs, such as the model's generalization abilities; and (3) characterize LLAMA. The original paper contains more details and a comparison to Mesh.²²

4.1. End-to-end evaluation

Table 3 shows end-to-end fragmentation improvements over TCMalloc for the four workloads, ranging from 19% to 78%. Figure 8 shows the image processing server's fragmentation as a function of time. Since vanilla TCMalloc did not support huge pages at the time the paper was written (it does now¹³), we reconstructed 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 the page release rate. Even when compared with the most

Table 3. Summary of model accuracy and end-to-end fragmentation results.

Workload	Prediction accuracy (%)		Final steady-state memory (MB)			Fragmentation reduction (%)
	Weighted	Unweighted	TCMalloc	LLAMA	Live	
Image processing server	96	73	664	446	153	43
TensorFlow InceptionV3 benchmark	98	94	282	269	214	19
Data processing pipeline	99	78	1964	481	50	78
Redis key-value store	100	94	832	312	115	73

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.

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.

4.2. Model evaluation

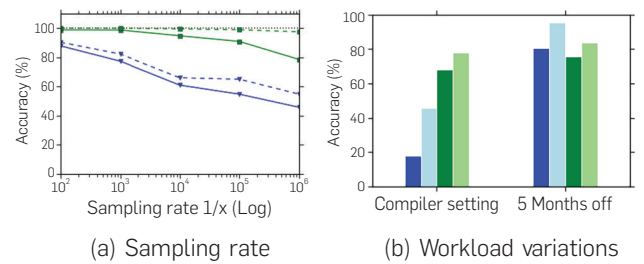
This section evaluates the accuracy of our model. Figure 9 shows that classification accuracy remains high when training our model on one version of the image server and applying it to another, and at relatively low sampling rates. The same configuration in Table 2 shows almost no matching stack traces with a lookup table. 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.

4.3. Performance overheads

This section explores overheads compared to TCMalloc. The original paper contains more details. The model takes 100–500 μ s per prediction for common stack sizes, which would be too expensive to run on every allocation but is acceptable in the context of LLAMA since the model only runs when missing in the cache. The allocator consumes 56MB for our first workload, less than 2% of the maximum heap size. As we show in Section 4.1, LLAMA recoups this memory easily.

We next evaluate our stack hashing approach. For the image server, 95% of predictions hit in the cache, which

Figure 9. 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.

shows that stack hashing reduces model evaluations. To evaluate the 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.

Finally, we characterize LLAMA's overall performance using a microbenchmark that stress tests the allocator. The average latency for global allocations hitting in the cache is 88.0ns (vs. 81.7ns for TCMalloc) while fast path allocations take 48.8ns (vs. 8.3ns for TCMalloc), much of it because of predictions. In practice, the memory allocator receives much less pressure than in this stress test and the difference is thus less pronounced. For example, the image server slows down $\approx 12.5\%$ per query compared to TCMalloc.

Our allocator is largely unoptimized. The global allocator is protected by a central lock that is currently the main performance bottleneck. We believe the prototype's bottlenecks can be addressed in a production implementation. Production allocator optimizations could 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.

5. DISCUSSION

This section discusses how we believe this work has broader implications for other space-time bin packing problems and the emerging field of ML for Systems.

5.1. Generalization to other problems

While this paper focuses on 2MB huge pages, 1GB huge pages are already available on current hardware. However,

to our knowledge, they are not widely relied upon for C++ workloads. LLAMA could feasibly be extended to handle this use case as well but may need to adjust some of its policies.

While this paper focuses on memory management, the LLAMA algorithm applies to any space-time bin packing problem where items are assigned to resources and a resource can only be released once all items within it are gone. These kinds of resource scheduling problems abound in computer science. For example, a component of a system can only be switched off if nothing is running on it, a storage system may require that a block can only be freed once it is fully empty, and load balancing in distributed systems often cannot move workloads and can only free resources once all running jobs on them have finished. Another example is NAND blocks in flash-based SSDs: When an SSD runs out of empty blocks, it needs to reclaim existing blocks and relocate all remaining data within them, incurring overhead for any block that is not completely empty.

In many of these cases, the lifetimes of items are predictable, for example, how long it will take to process a particular request or how long a particular file will persist. In this case, the LLAMA algorithm may provide an effective solution to these problems. The prerequisite is that there is some additional information that enables predictions, such as stack traces, memory addresses, or request metadata.²⁵

5.2. A general pattern of ML for systems

Generalizing some of the insights from this paper led to a new methodology for applying Machine Learning in computer systems.¹⁷ Prior work on ML for systems focused on learning a problem end-to-end, which can be inefficient and requires complex models. This paper instead shows an approach to using ML to learn only a particular piece of information that was previously unavailable—in this case, the lifetime of an allocated object. By applying ML to a more limited problem, simpler models can be used and learning can be integrated into a traditional system in a more practical way. At the same time, exploiting the new information may require redesigning the rest of the system, both a cost and an opportunity, and it certainly requires adding the ability to tolerate mispredictions.

Isolating the portion of the problem that needs to be learned also means that the community now can establish best practices, benchmarks, and tools for these specific sub-problems. We found that most of these problems fall into a small number of categories.¹⁷

5.3. Follow-up work since the original paper


While LLAMA focuses on memory allocation, we also worked on storage systems and showed that similar patterns repeat in this area.²⁵ Since the publication of the LLAMA paper, the TCMalloc team has published a paper about Temeraire, a new huge page-aware allocator.¹³ We applied insights from this work to Temeraire, in order to make better decisions about when to break up huge pages in this allocator, which led to an estimated 1% throughput improvement across Google's fleet.¹⁸

6. 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 contexts*, 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 lifetimes into the same huge pages, tracks actual lifetimes, and uses them to correct for mispredictions. It limits fragmentation by filling gaps created by frees with shorter-lived objects. In this context, this work solves challenges related to applying ML to systems problems with strict resource and latency constraints. We believe that the LLAMA approach applies to a wide range of other space-time bin packing problems.

Acknowledgments

We would like to thank Harry Xu who was the shepherd of the original paper. We would also like to thank Ana Klimovic, Chris Kennelly, Christos Kozyrakis, Darryl Gove, Deniz Altinbukan, Jae W. Lee, Jeff Dean, Khanh Nguyen, Mark Hill, Martin Abadi, Mike Burrows, Milad Hashemi, Paul Barham, Paul Turner, Sanjay Ghemawat, Steve Blackburn, 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, and Snehasish Kumar for open sourcing the lifetime profiler. 

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation OSDI'16* (2016). USENIX Association, Berkeley, CA, 265–283.
2. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. (CSUR)* 51, 4 (2018), 81.
3. Allamanis, M., Brockschmidt, M., Khademi, M. 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* (2018). OpenReview.net.
4. Barrett, D.A., Zorn, B.G. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93*, (1993). ACM, NY, 187–196.
5. Basu, A., Gandhi, J., Chang, J., Hill, M.D., Swift, M.M. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13* (2013). ACM, NY, 237–248.
6. Blackburn, S.M., McKinley, K.S. 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* (2008), 22–32.
7. Bruno, R., Patricio, D., Simão, J., Veiga, L., Ferreira, P. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19* (2019). ACM, NY, 1–28.
8. Chen, B., Tarlow, D., Swersky, K., Maas, M., Heiber, P., Naik, A., et al. *Learning to Improve Code Efficiency* (2022). <https://arxiv.org/abs/2208.05297>.
9. Clifford, D., Payer, H., Stanton, M., Titzer, B.L. Memento mori: Dynamic allocation-site-based optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management* (2015), 105–117.
10. Goodfellow, I., Bengio, Y., Courville, A. *Deep Learning*. MIT Press (2016). <http://www.deeplearningbook.org>.
11. Google. pprof. 2020. <https://github.com/google/pprof>.
12. Hochreiter, S., Schmidhuber, J. Long short-term memory. *Neural Comput.* 9, 8 (1997), 1735–1780.
13. Hunter, A., Kennelly, C., Turner, P., Gove, D., Moseley, T., Ranganathan, P. Beyond malloc efficiency to fleet efficiency: A hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (2021). USENIX Association, Berkeley, CA, 257–273.
14. Kanev, S., Darago, J.P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y. et al. Profiling a warehouse-scale

- computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15* (2015), ACM, NY, 158–169.
15. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. et al. Backpropagation applied to handwritten zip code recognition. *Neural Comput.* 1, 4 (1989), 541–551.
 16. Lo, D., Cheng, L., Govindaraju, R., Barroso, L.A., Kozyrakis, C. Towards energy proportionality for large-scale latency-critical workloads. In *International Conference on Computer Architecture (ISCA)* (2014), IEEE, NY, 301–312.
 17. Maas, M. A taxonomy of ml for systems problems. *IEEE Micro* 40, 5 (2020), 8–16.
 18. Maas, M., Kennelly, C., Nguyen, K., Gove, D., McKinley, K.S., Turner, P. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management, ISMM 2021* (2021), ACM, NY, 28–38.
 19. Mikolov, T., Chen, K., Corrado, G., Dean, J. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations (ICLR 2013)* (Scottsdale, Arizona, USA, May 2–4, 2013), Workshop Track Proceedings.
 20. Mytkowicz, T., Coughlin, D., Diwan, A. Inferred call path profiling. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09* (2009), ACM, NY, 175–190.
 21. Olston, C., Li, F., Harmsen, J., Soyke, J., Gorovoy, K., Lao, L. et al. TensorFlow-Serving: Flexible, high-performance ML serving. In *Workshop on ML Systems at NIPS 2017*, 2017.
 22. Powers, B., Tench, D., Berger, E.D., McGregor, A. Mesh: Compacting memory management for C/C++ applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2019), ACM, NY, 333–346.
 23. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z. Rethinking the inception architecture for computer vision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), IEEE, NY.
 24. Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D. Dynamic storage allocation: A survey and critical review. In *Memory Management*. H.G. Baler, ed. Berlin, Heidelberg, Springer Berlin Heidelberg, 1995, 1–116.
 25. Zhou, G., Maas, M. Learning on distributed traces for data center storage systems. In *Proceedings of Machine Learning and Systems*. A. Smola, A. Dimakis and I. Stoica, eds, Volume 3, 2021, 350–364.

Martin Maas, mmaas@google.com, Google Research, Mountain View, CA, USA.

Michael Isard, misard@google.com, Google Research, San Francisco, CA, USA.

Kathryn McKinley, ksmckinley@google.com, Google, Seattle, WA, USA.

David G. Andersen, dga@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, PA, USA.

Mohammad Mahdi Javanmard, mjavanmard@meta.com, Meta, New York, NY, USA.

Colin Raffel, craffel@cs.unc.edu, University of North Carolina, Chapel Hill, NC, USA.

This work was done while Andersen, Javanmard, and Raffel were at Google.

© 2024 Copyright held by the owner/author(s).

Linking the World's Information

*Essays on
Tim Berners-Lee's
Invention of the
World Wide Web*

**Oshani Seneviratne,
James Hendler, Editors**

ISBN: 979-8-4007-0792-6

DOI: 10.1145/3591366

<http://books.acm.org>



ACM BOOKS
Collection III

Linking the World's Information

*Essays on
Tim Berners-Lee's
Invention of the
World Wide Web*

Oshani Seneviratne, James Hendler (Editors)



ASSOCIATION FOR COMPUTING MACHINERY