Learning-based Memory Allocation for C++ Server Workloads [ASPLOS 2020]

Ashish Panwar

Slides inspired by (or borrowed from):

https://www.youtube.com/watch?v=gs8m5W-xdDM

https://abelay.github.io/6828seminar/notes/6828_llama_isaac.pdf

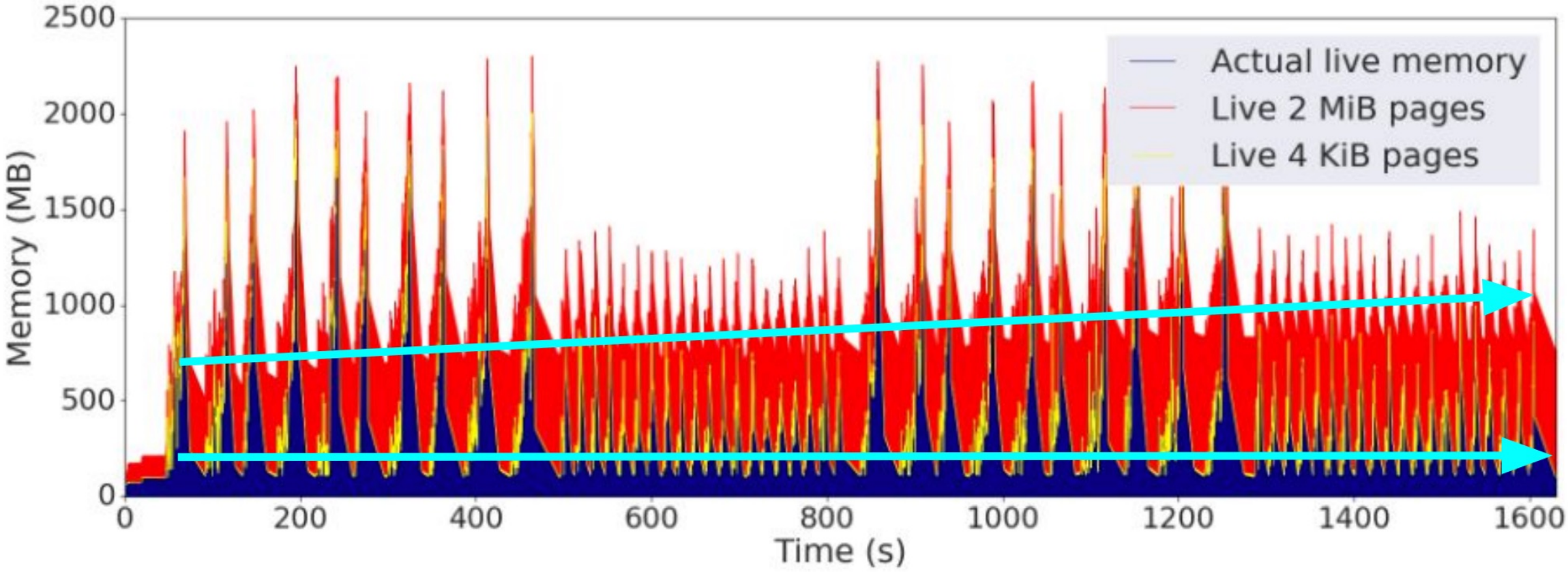
Problem statement

- Huge pages (e.g., 2MB pages) improve performance – up to 50%
- But cause (internal) fragmentation – memory waste up to 2x

Application	dTLB Load Walk (%)	
	Before	After
Tensorflow [1]	9.5 ± 0.6	9.0 ± 0.6
search1 [6, 18]†	10.3 ± 0.2	10.2 ± 0.1
search2†	8.9 ± 0.1	8.9 ± 0.3
search3 †	38.1 ± 1.3	15.9 ± 0.3
ads1	27.4 ± 0.4	10.3 ± 0.2
ads2	27.1 ± 0.5	11.6 ± 0.2
ads3	28.5 ± 0.9	11.1 ± 0.3
ads4	21.9 ± 1.2	16.7 ± 2.4
ads5	33.6 ± 2.4	17.8 ± 0.4
ads6	31.0 ± 4.3	15.7 ± 1.8
Spanner [17]	19.6 ± 1.2	9.5 ± 4.5
loadbalancer†		
Average (all WSC apps)	23.3	12.4

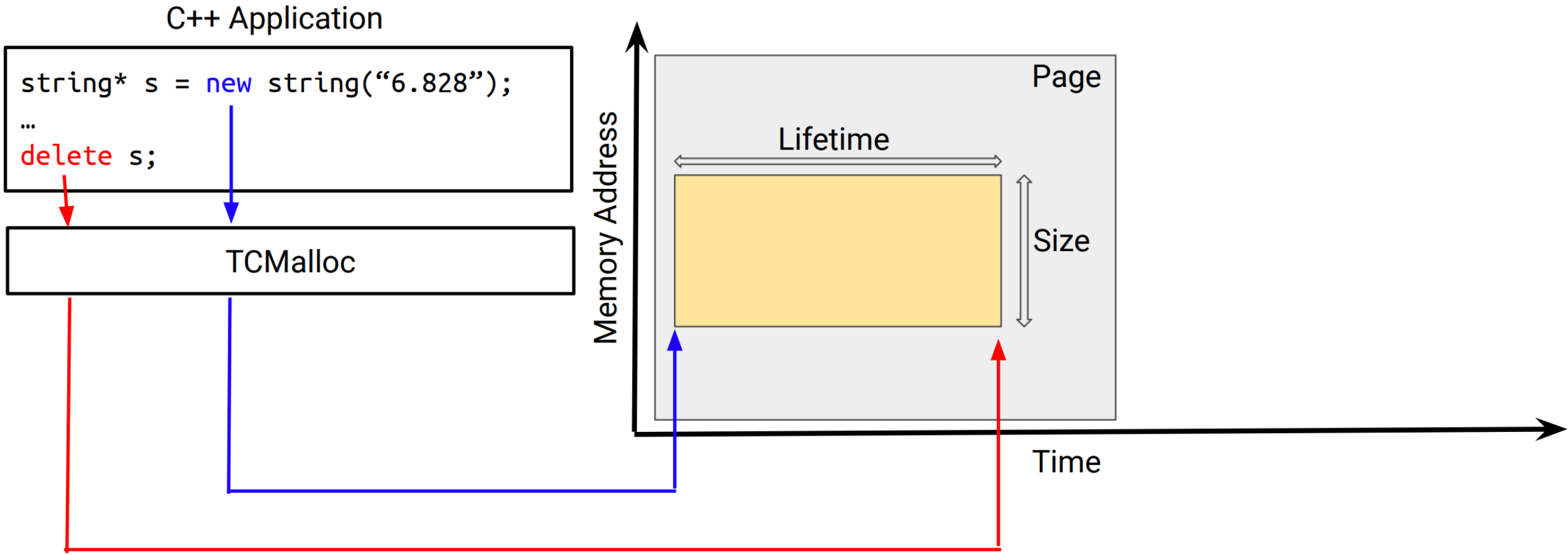
Table source: https://www.usenix.org/system/files/osdi21-hunter.pdf

Fragmentation with Huge Pages

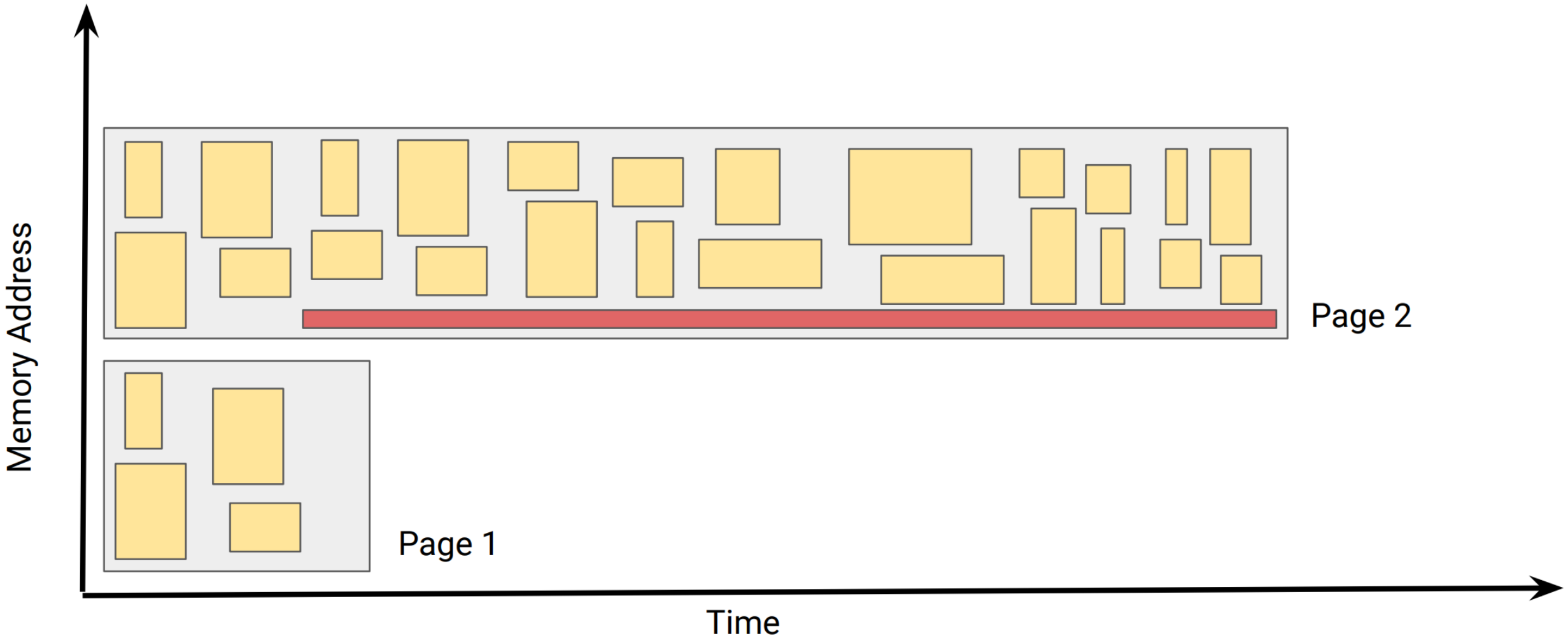


- Fragmentation in 4 KB pages is 1.03x
- Fragmentation in 2 MB pages is 2.15x

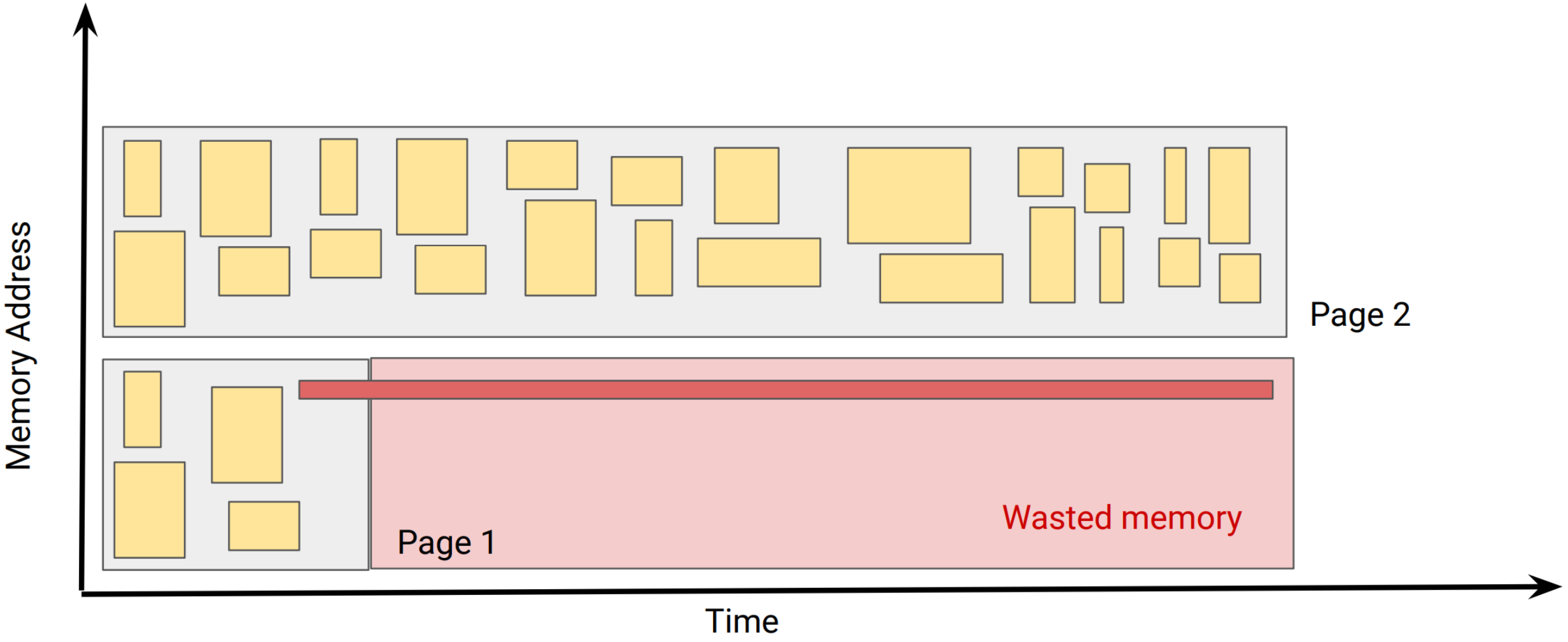
Memory allocation



Placement of long-lived objects matters



Placement of long-lived objects matters



Page size matters too!

- Assume 64B objects, 99.99% objects are short-lived
- # objects per 4KB page = 64
- # objects per 2MB page = 32,768

Page size	Probability that at-least 1 object is long-lived
4KB	0.6%
2MB	96.2%

This is a common problem

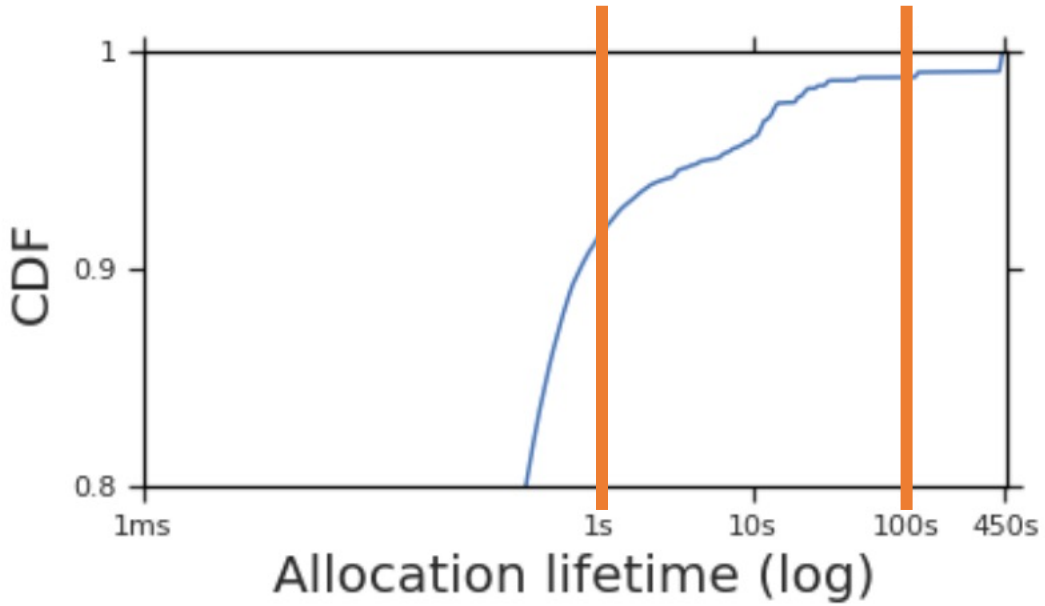
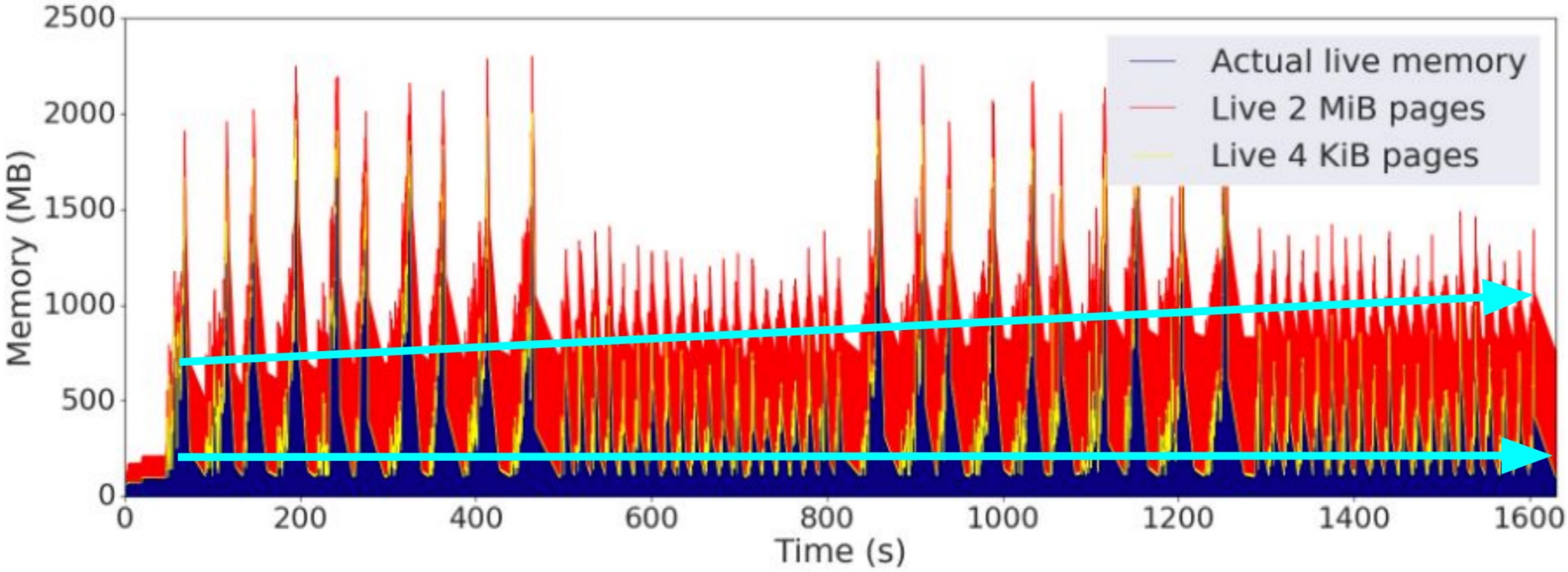


Figure 2. Long tail of object lifetimes from a single run; x-axis is log-scale. The vast majority of objects are short-lived, but rare long-lived objects impact fragmentation.

Fragmentation with Huge Pages



- Fragmentation in 4 KB pages is 1.03x
- Fragmentation in 2 MB pages is 2.15x

Mitigating internal fragmentation in huge pages requires information about object lifetimes

Paper contributions

- ML-based technique for predicting object lifetimes
- Lifetime-aware memory allocator
- Evaluation of the above two

Predicting object lifetimes

- Why conventional solutions do not work?
  - Such as profile-guided optimizations (PGO)

- PGO:
  - Calling context can be used to predict object lifetimes
  - Instrument code to profile object lifetimes – offline
  - Lifetime is classified as either short or long
  - Analyze and deploy

- PGO has been used for object pre-tenuring in managed runtimes
  - Place long-lived objects directly in the old generation
  - Can tolerate inaccurate predictions

Challenges

Insufficient Coverage

```
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> >*, cha
3. r const*, proto2::internal::ParseContext*)
4. proto2::FileDescriptorProto::_InternalPars
   e(char const*,
   proto2::internal::ParseContext*)
5. proto2::MessageLite::ParseFromArray(void
   const*, int)
6. proto2::DescriptorPool::TryFindFileInFallb
   ackDatabase(std::__g::basic_string<char,
   std::__g::char_traits<char>,
   std::__g::allocator<char> > const&) const
7. proto2::DescriptorPool::FindFileByName(std
   ::__g::basic_string<char,
   std::__g::char_traits<char>,
   std::__g::allocator<char> > const&) const
```

Stack Trace Instability

Performance Sensitivity

PGO Limitation: Coverage

- PGO needs to have seen the context before making a prediction
- 64% of distinct allocation contexts are seen once
- 17% of all permanent allocations are from contexts that are observed once

Combine profiling across multiple runs

Instability!

PGO Limitation: Instability

- Stack traces are brittle across executions
  - Compilation under different configuration settings, library updates
  - Function names and interfaces change over time
  - Address space randomization

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).

PGO Limitation: Performance

- Collecting a stack trace is expensive!

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

PGO is expensive and does not generalize well enough in production settings

Contribution-1: ML-based lifetime prediction

- Train an ML model over stack traces
- Pointer-bases stack traces are valid only within a run
- Train an ML model over symbolized stack traces

Contribution-1: ML-based lifetime prediction

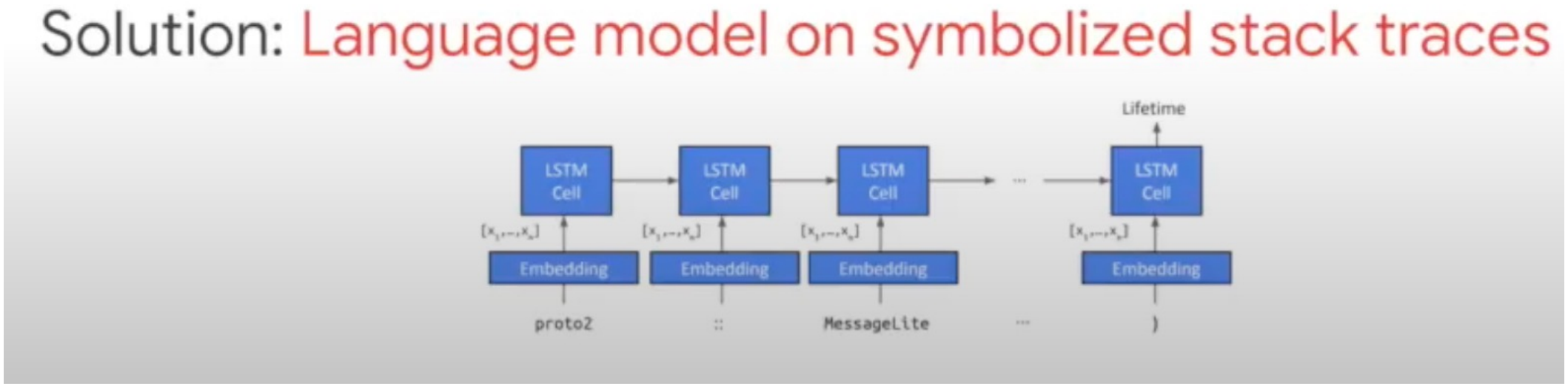
```
1   __gnu_cxx::__g::__string_base char, std::__g::char_traits
    char,std::__g::allocator char::_M_reserve(unsigned long)
2   proto2::internal::InlineGreedyStringParser(std::__g::
    basic_string char, std::__g::char_traits char,std::__g::
    allocator char*,char const*,proto2::internal::ParseContext*)
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.

Contribution-1: ML-based lifetime prediction

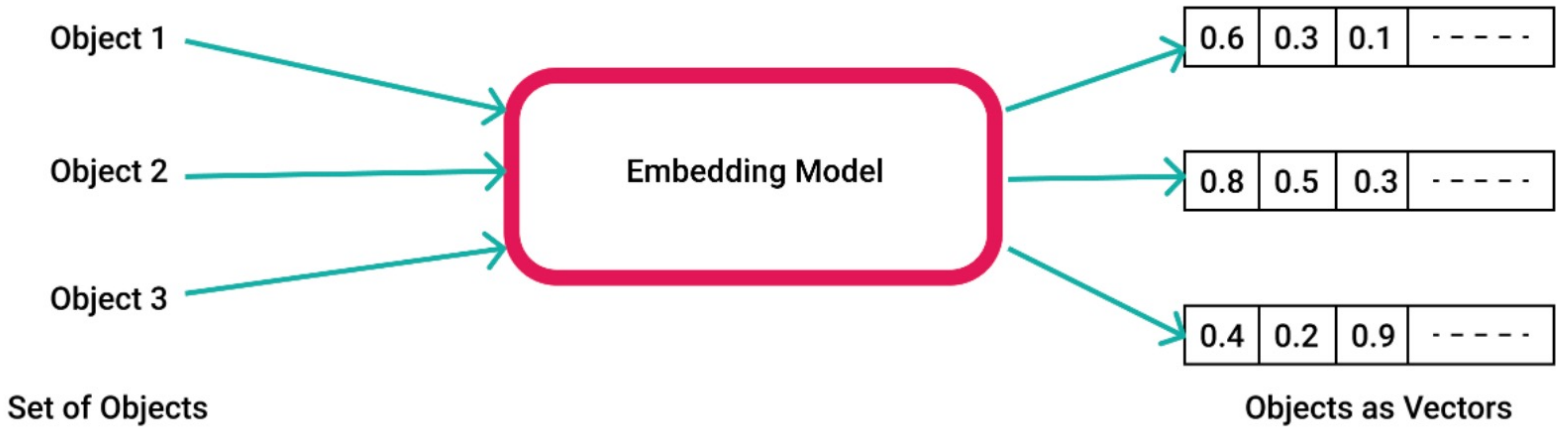
- Symbolized stack traces capture program nesting (as a string)
- Long-short term memory (LSTM) based recurrent neural networks
- LSTMs are used for sequence prediction in natural language processing
- A natural fit for mapping symbolized stack traces to object lifetimes
- The authors use LSTMs for simplicity (other more effective ML techniques can be explored)

Contribution-1: ML-based lifetime prediction



Contribution-1: ML-based lifetime prediction

- Input: symbolized stack traces
  - Each frame in the stack trace represents a string
  - Tokenize based on special characters (:: or ,)
  - There is an embedding vector for each token



  - Set of vectors form an embedding matrix A
  - A is trained as part of the model

Contribution-1: ML-based lifetime prediction

- Sampling-based data collection – the training dataset
- Instrumenting allocation and free calls
- An allocation site is assigned 95th percentile of observed lifetimes

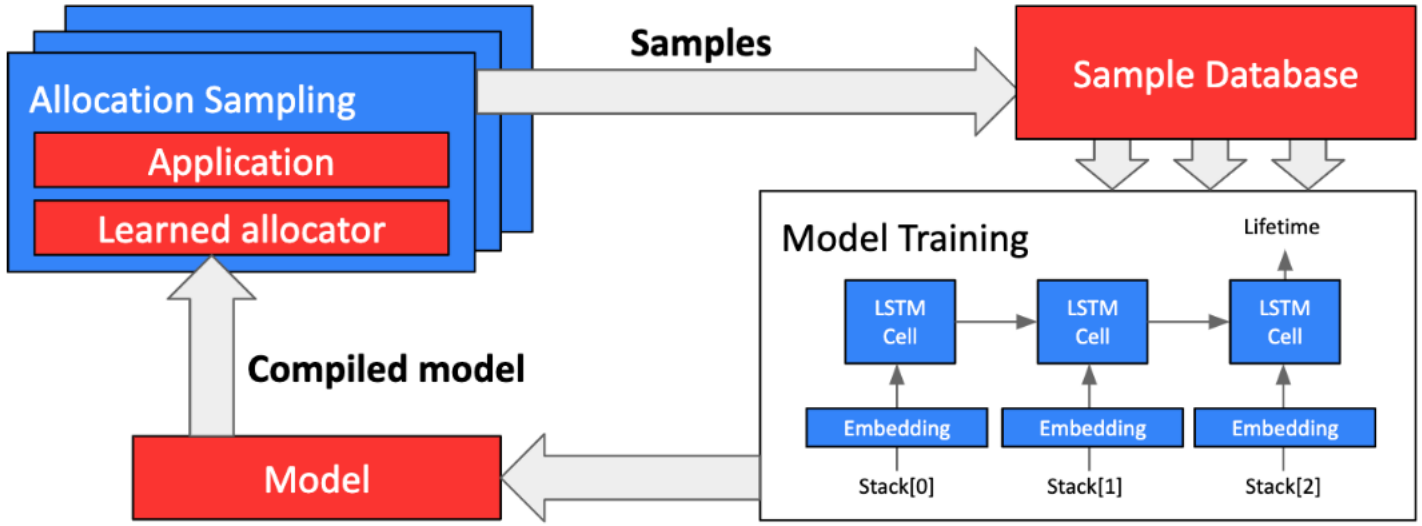
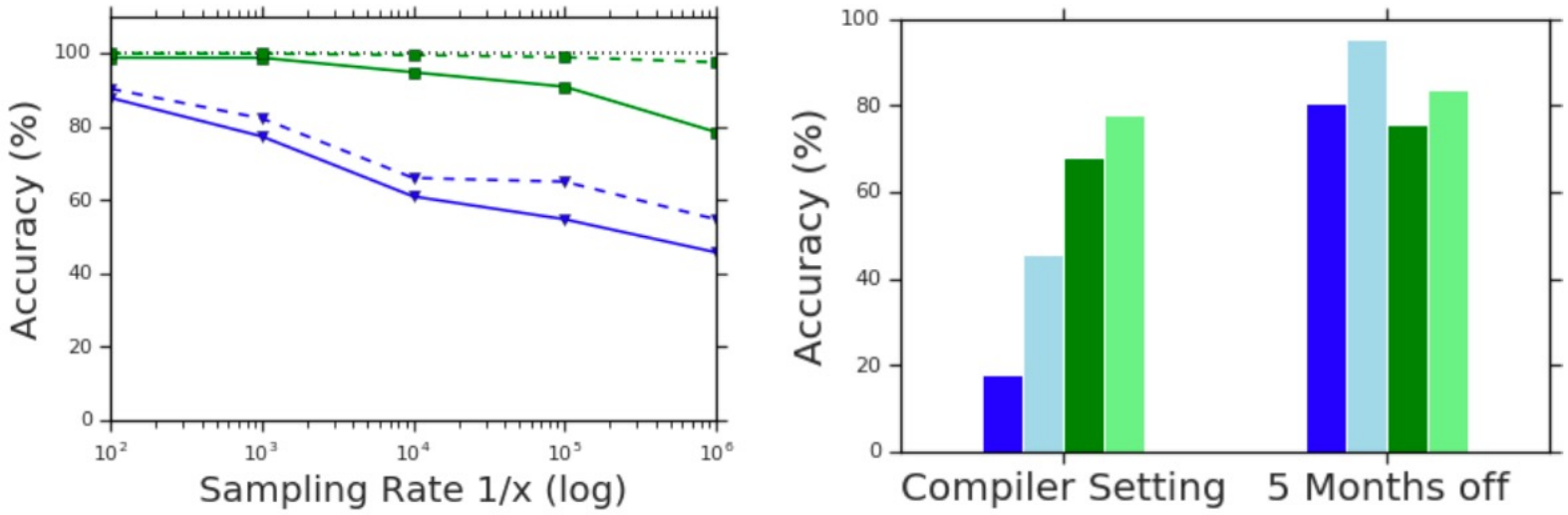


Figure 3. Overview of our ML-based Allocator

Lifetime prediction accuracy

Workload	Prediction Accuracy	
	Weighted	Unweighted
Image Processing Server	96%	73%
TensorFlow InceptionV3 Benchmark	98%	94%
Data Processing Pipeline	99%	78%
Redis Key-Value Store	100%	94%

Lifetime prediction generalizes well



(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.

Contribution-2: LLAMA

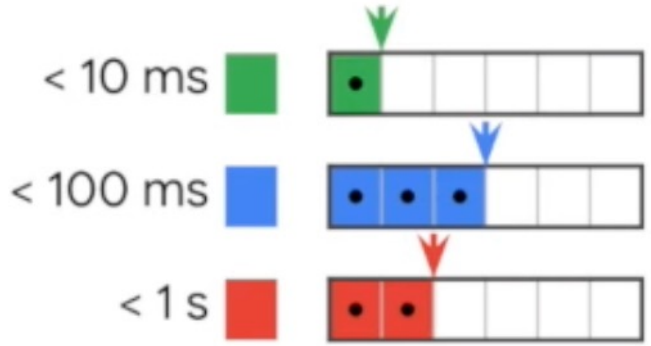
- LLAMA – Learned Lifetime-aware Memory Allocator

Allocator built around lifetime classes instead of size:

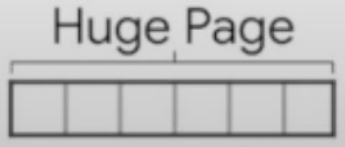
- Lifetime classes: {<10ms,<100ms,<1s,...,∞}
- Allocator manages memory in huge pages
- Small objects (<8 KB) managed in 16 KB block spans through policy similar to Immix (details in the paper)

Each huge page has a lifetime class and only contains objects predicted to be at most this lifetime class

Contribution-2: LLAMA

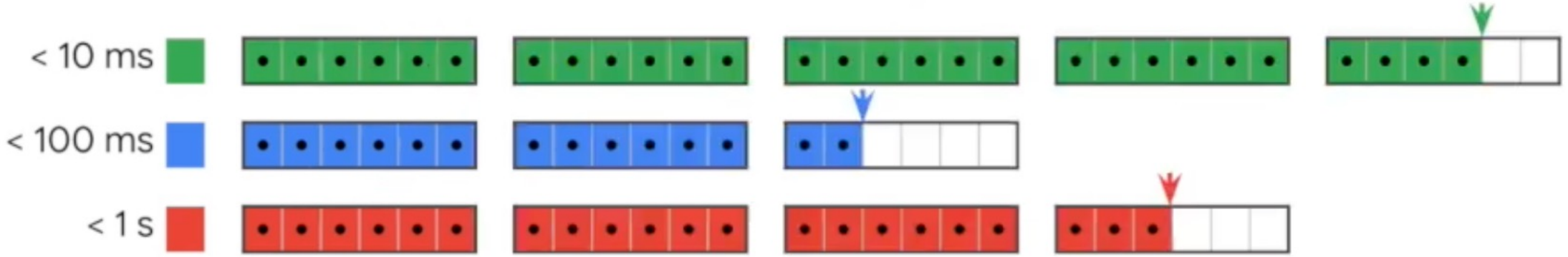


Perform bump-pointer allocation into new pages



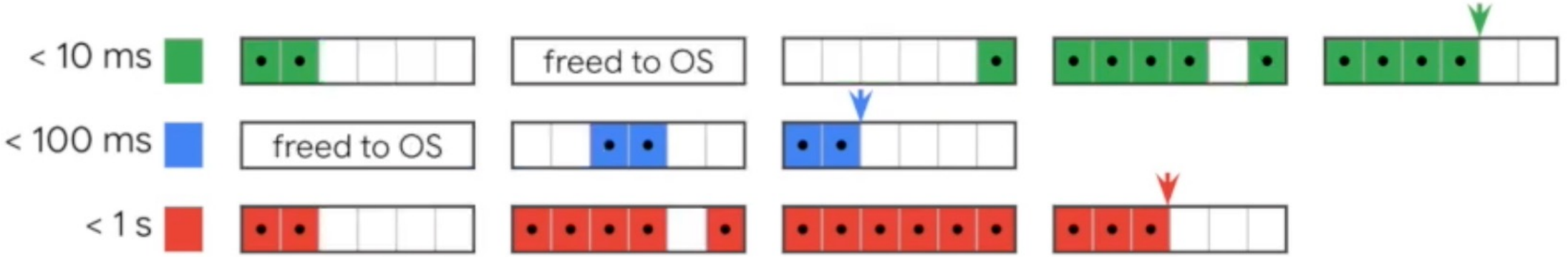
• Residual Allocation **LC** Lifetime Class

Contribution-2: LLAMA



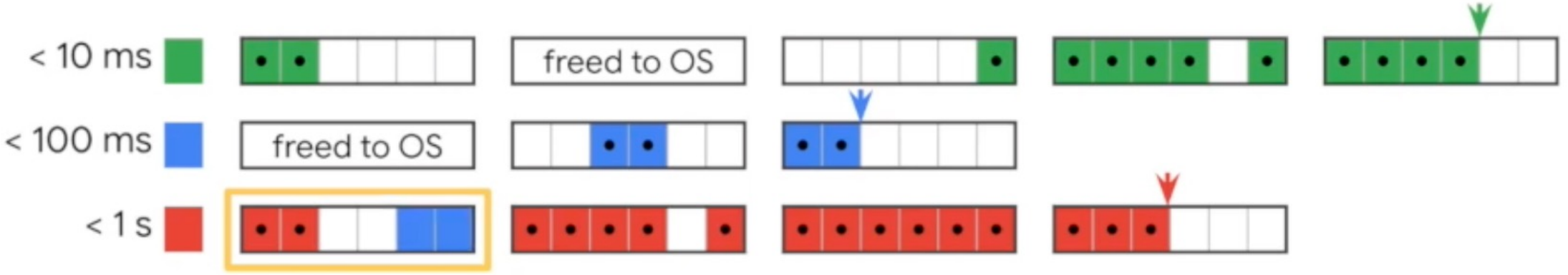
Perform bump-pointer allocation into new pages

Contribution-2: LLAMA



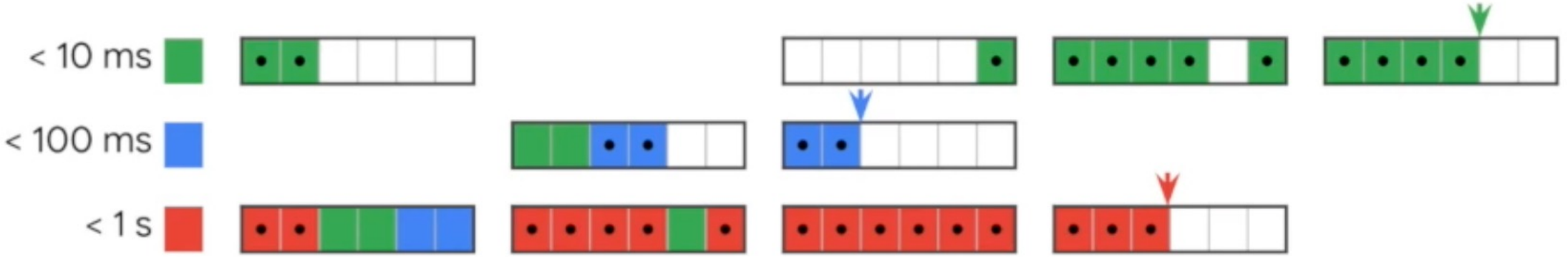
(Blocks of the original lifetime class are called residual blocks, shown with dots)

Contribution-2: LLAMA



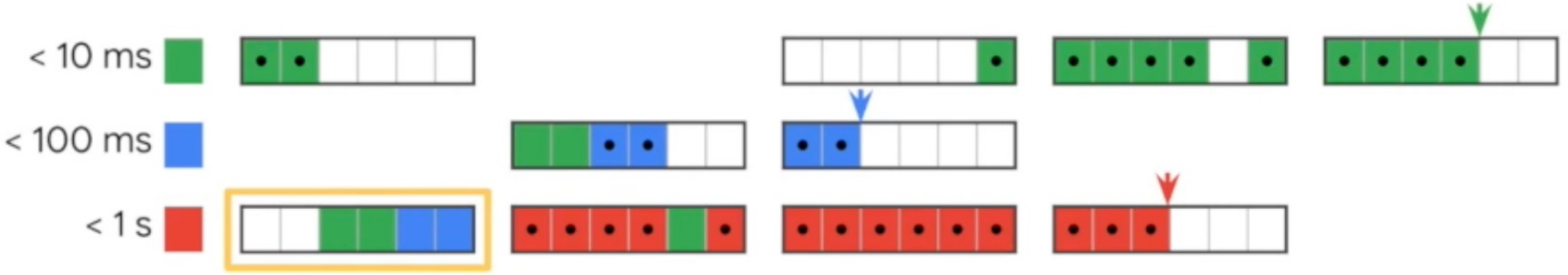
As blocks disappear, gaps are filled with shorter-lived blocks

Contribution-2: LLAMA



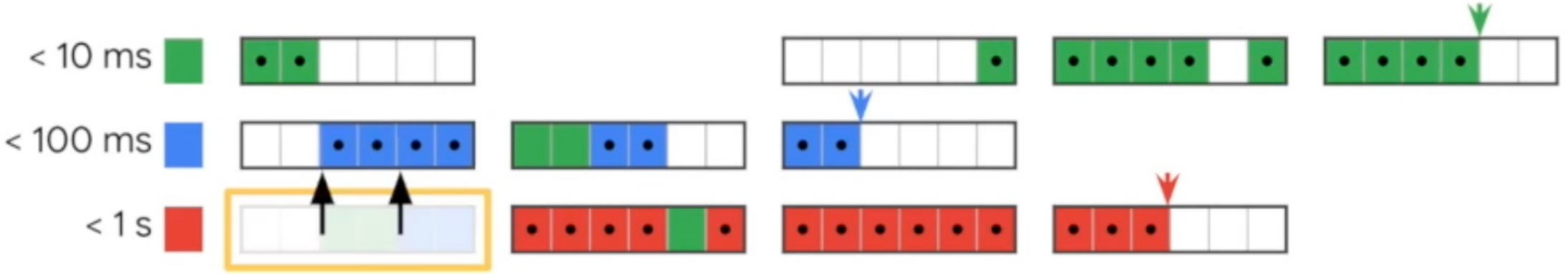
As blocks disappear, gaps are filled with shorter-lived blocks

Contribution-2: LLAMA



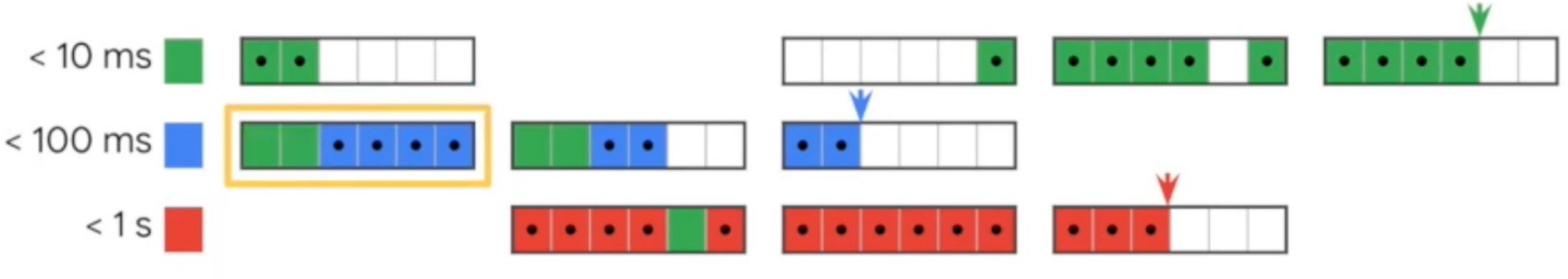
When all residual objects have disappeared, the page is moved into the next-lower lifetime class or freed

Contribution-2: LLAMA



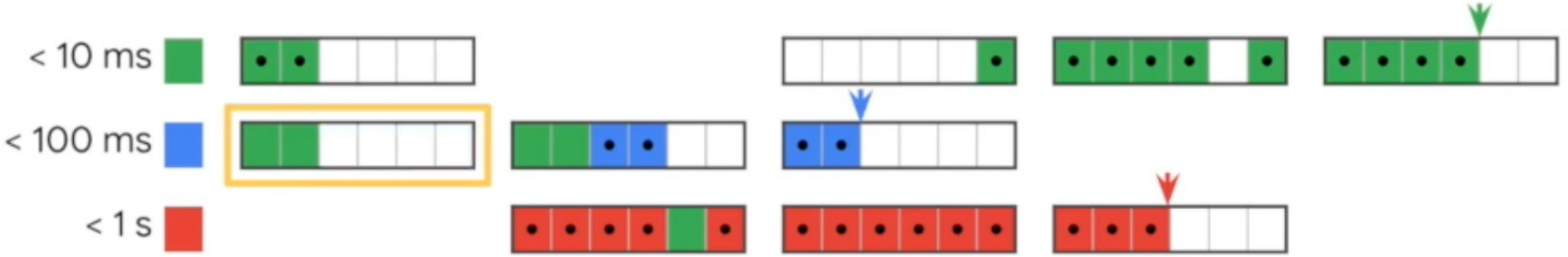
When all residual objects have disappeared, the page is moved into the next-lower lifetime class or freed

Contribution-2: LLAMA



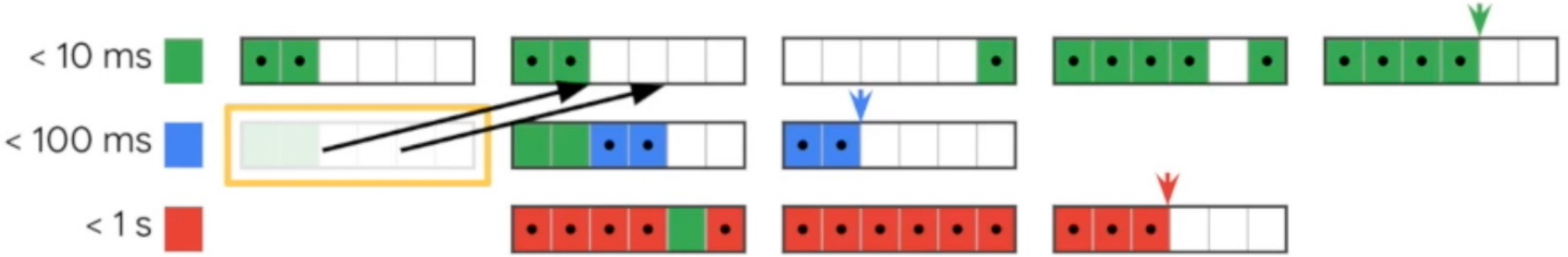
When all residual objects have disappeared, the page is moved into the next-lower lifetime class or freed

Contribution-2: LLAMA



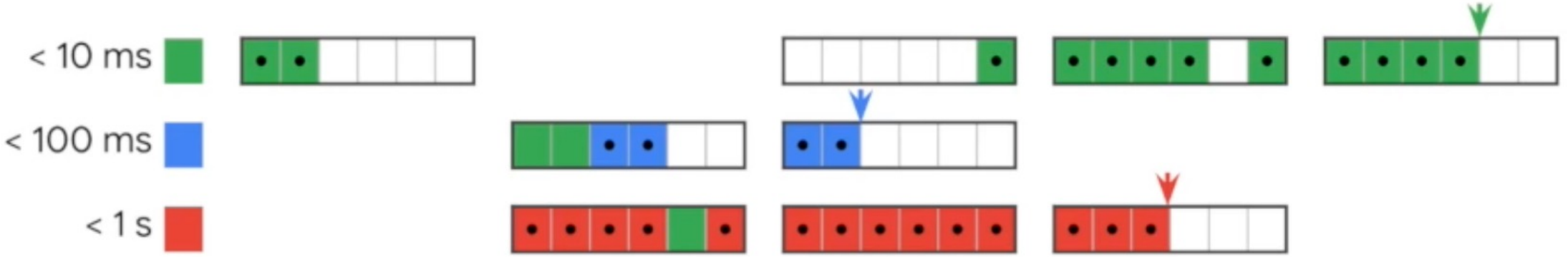
When all residual objects have disappeared, the page is moved into the next-lower lifetime class or freed

Contribution-2: LLAMA



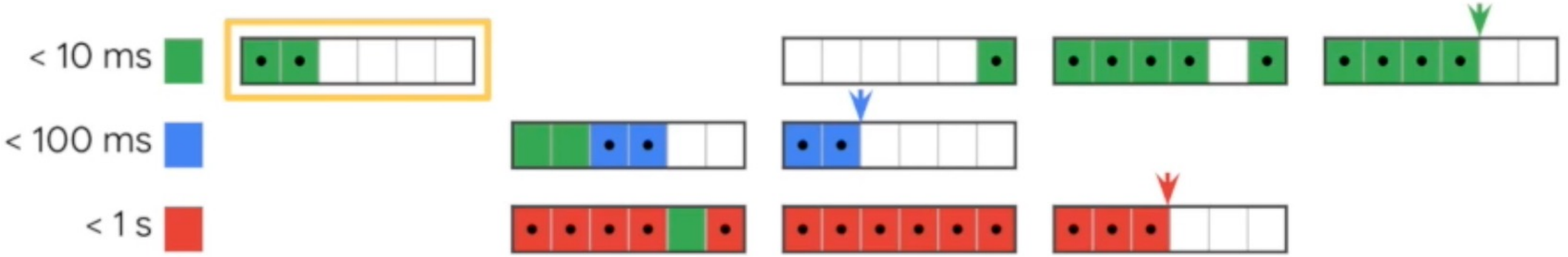
When all residual objects have disappeared, the page is moved into the next-lower lifetime class or freed

Contribution-2: LLAMA



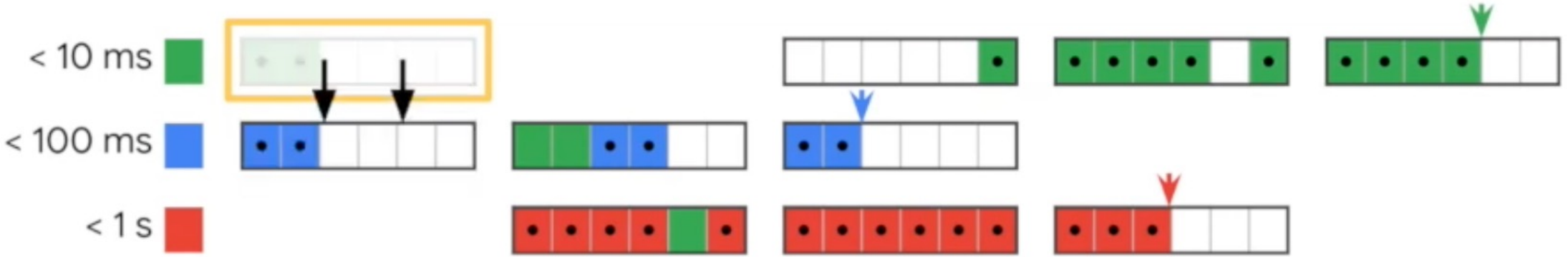
When all residual objects have disappeared, the page is moved into the next-lower lifetime class or freed

Contribution-2: LLAMA



If a page's deadline expires, we mispredicted and move it up one lifetime class

Contribution-2: LLAMA



If a page's deadline expires, we mispredicted and move it up one lifetime class

Performance optimization

- Avoid invoking the ML model by caching the results of prior requests
- Thread-local hashmap (key – return address, stack height, object size)
- 95% predictions are cache hits, 14% disagree with the cached value

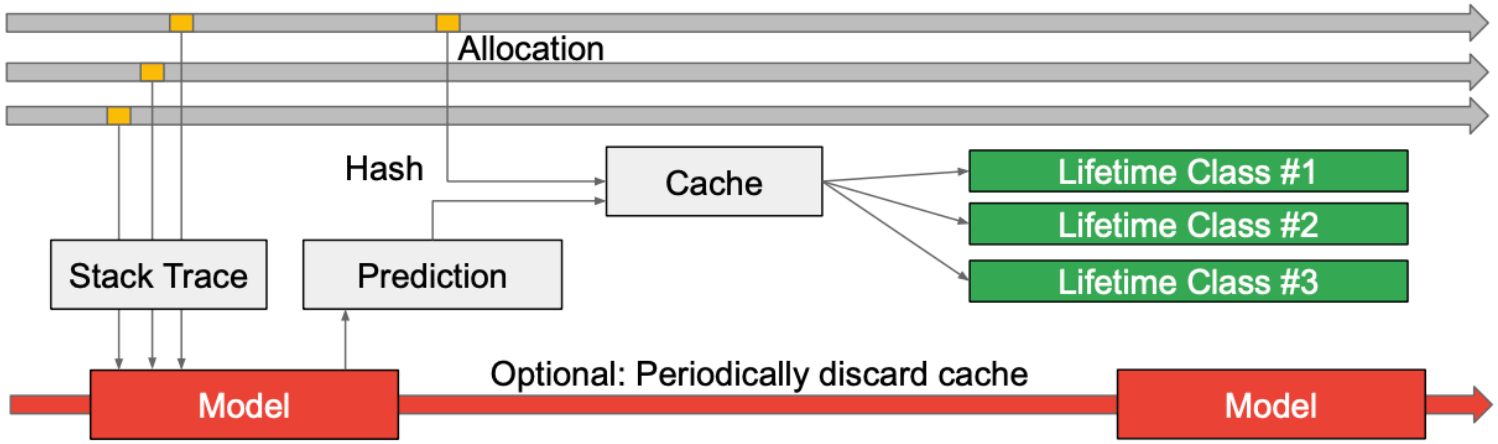


Figure 7. High-level overview of low-latency prediction. We use the model only when the hash of the current stack trace is not in the cache. Discarding cache entries periodically helps dynamically adapting to workload changes.

Results

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%

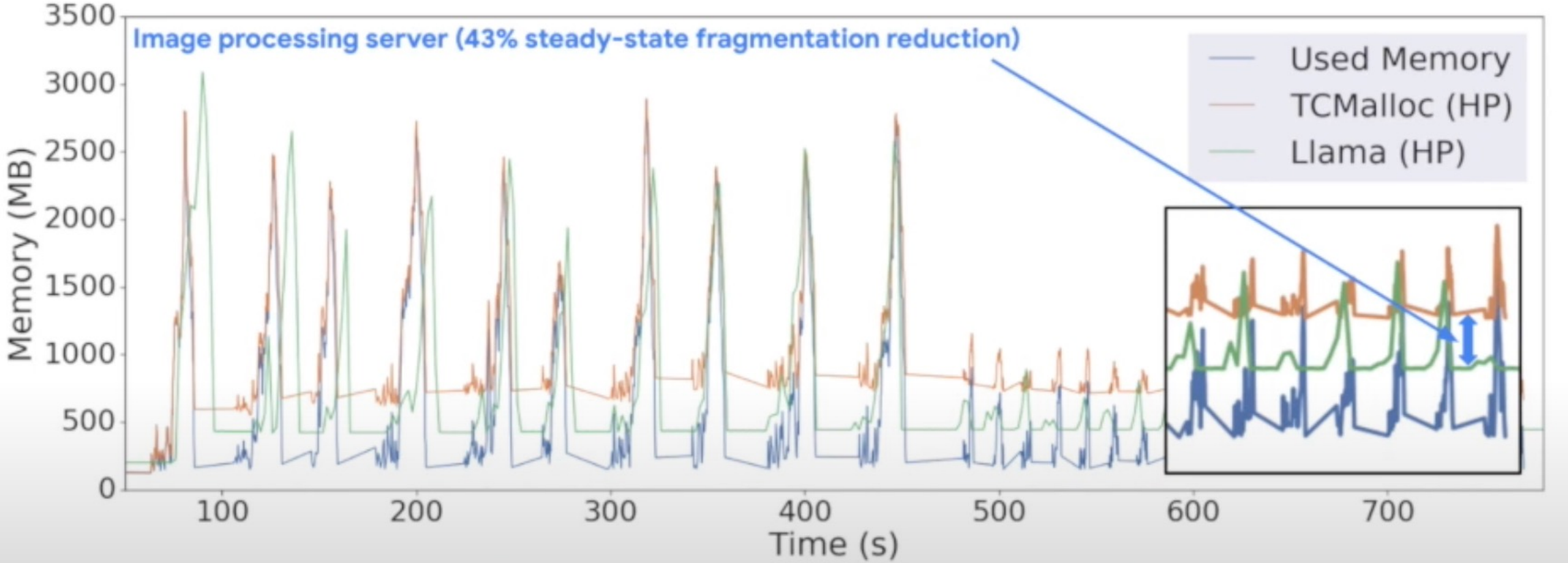
Table 3. Summary of Model Accuracy and End-to-end Fragmentation Results

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

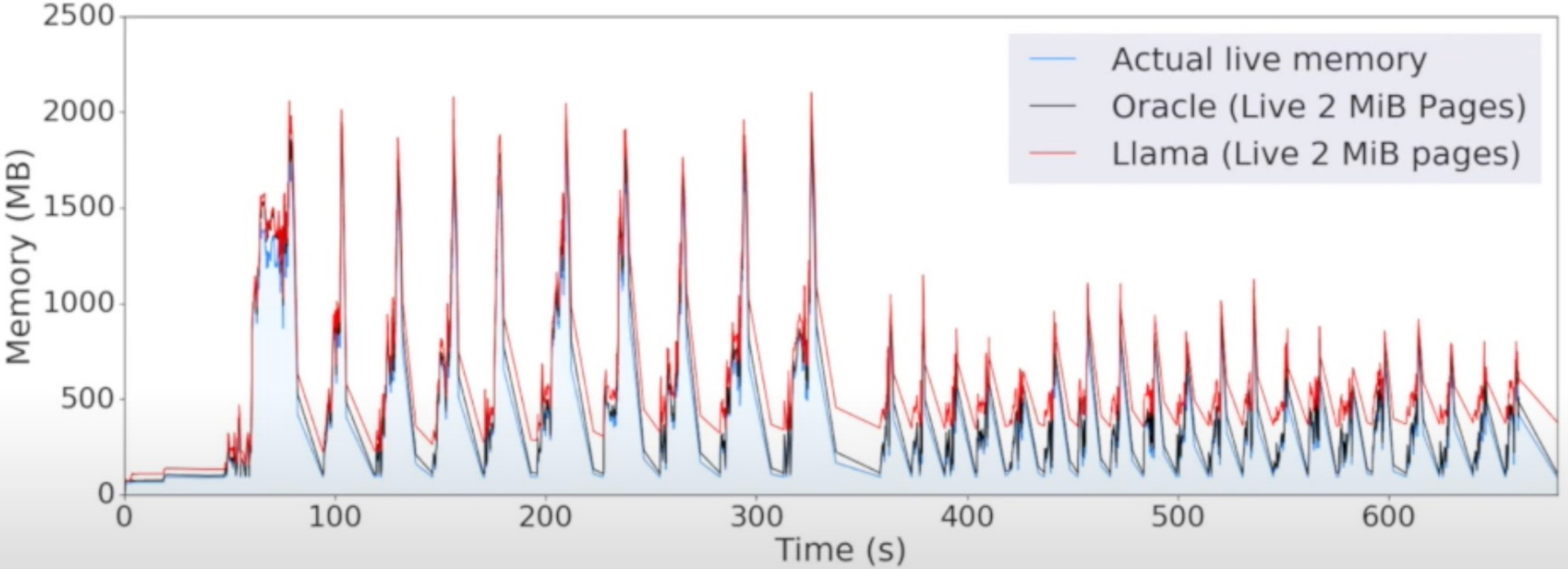
- Up to 12.5% application slowdown (image server)

End-to-end Results



Steady-state fragmentation reduced by 19-78%

Oracle Results



1.07x optimal memory footprint with perfect lifetimes

Take aways

- Predicting object lifetimes with good accuracy is feasible – thanks to ML!
- Lifetime based memory allocator reduces internal fragmentation in huge pages
- Performance optimizations (e.g., caching) can hide the overhead of ML models
- A great example of using ML to solve systems problems!