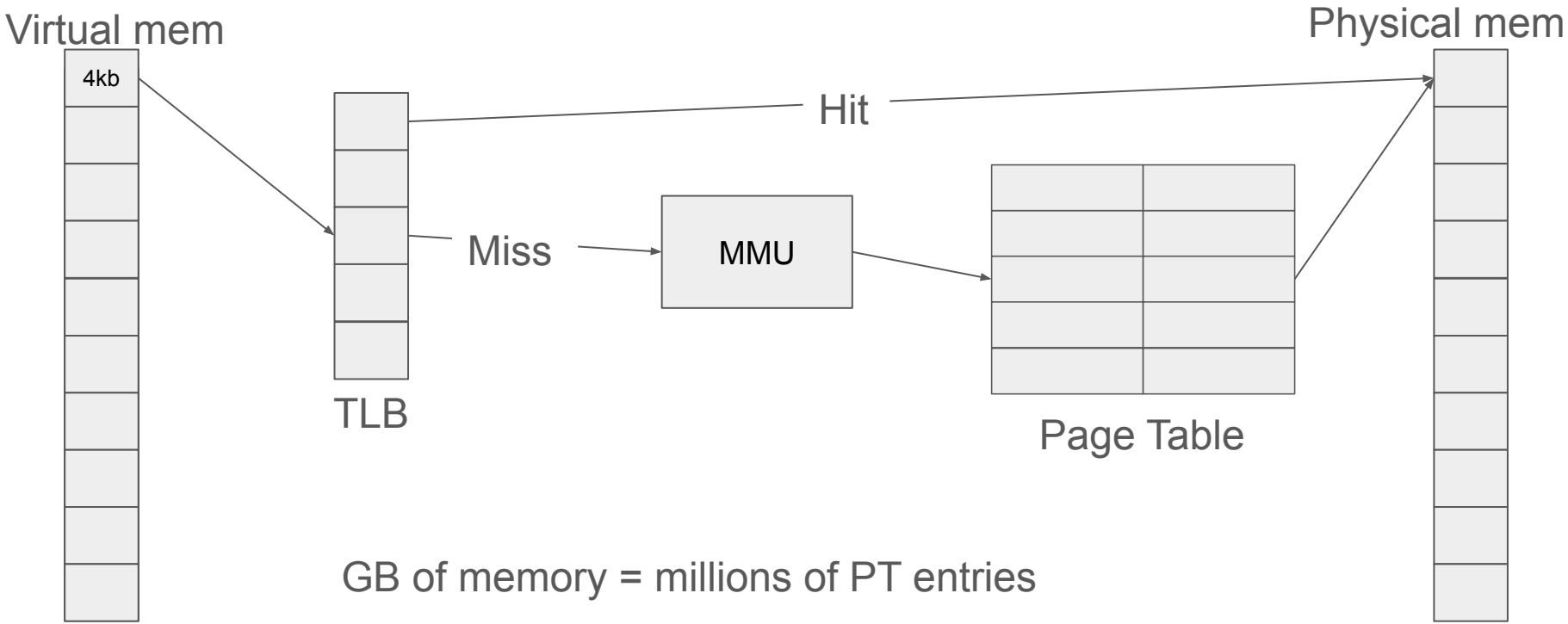
# Learning-based Memory Allocation for C++ Server Workloads

# Virtual Memory: Default (4 KB) Pages

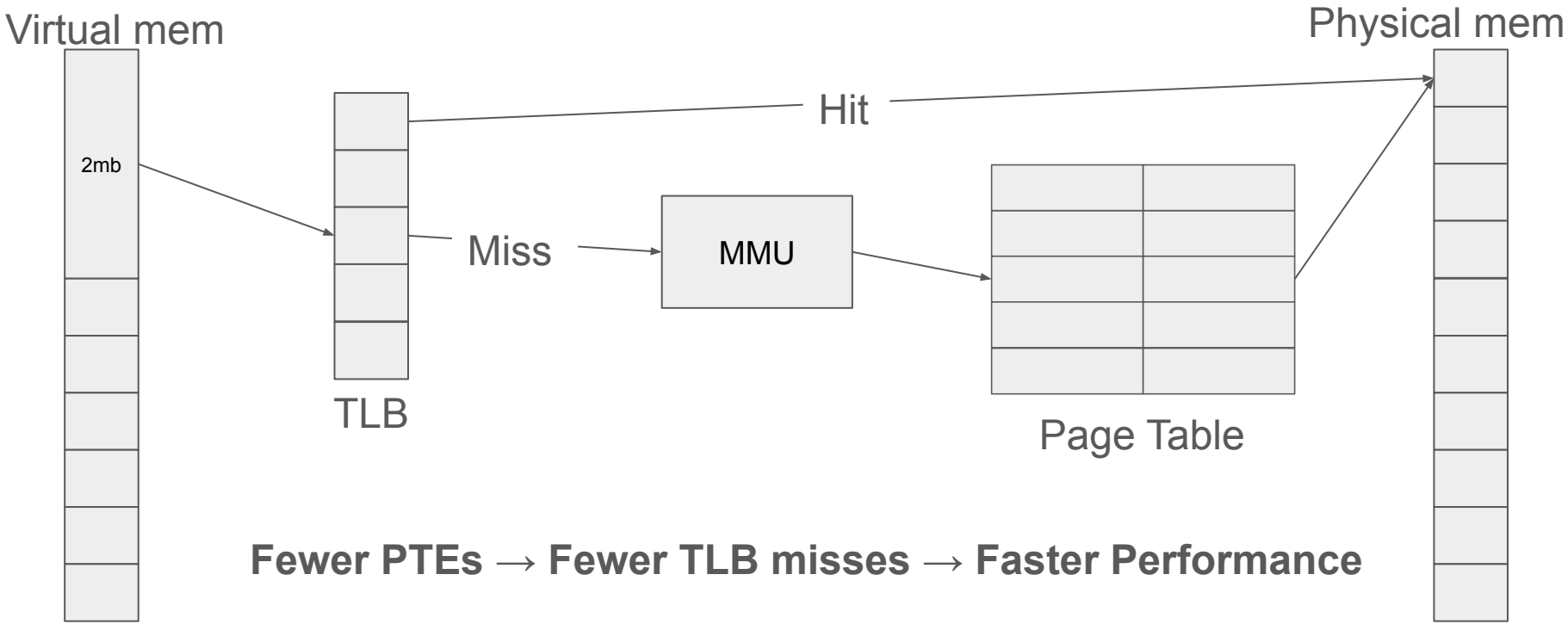
- Recap: How memory mapping works in Linux




GB of memory = millions of PT entries

# What are HugePages?

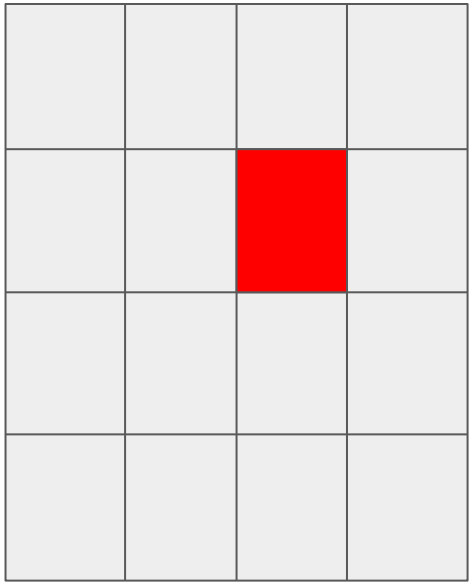
- HugePages allow larger pages sizes (typically 2MB)




**Fewer PTEs → Fewer TLB misses → Faster Performance**

# HugePage fragmentation

- If part of the page is allocated, the hugepage cannot be reclaimed, so the mapped physical memory can be larger than allocated memory
  - Worse, the subpages cannot be remapped to service other allocations
- The OS can break up huge pages



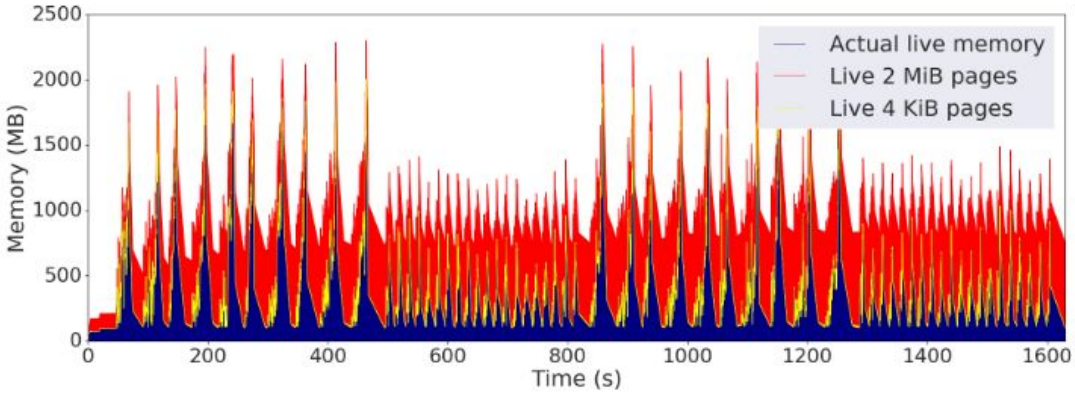



**Figure 1.** Image server memory usage resizing groups of large and small images either backed by huge (red) or small (yellow) pages in the OS, derived from analyzing an allocation trace in a simulator. Huge pages waste systemically more memory and increasingly more over time.

# Lifetime and fragmentation

- Fragmentation is caused by differing lifetimes on the same page.
  - *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%. [...] Solving this problem fundamentally depends on reasoning about object lifetimes and grouping objects with similar lifetimes together*

# TCMalloc

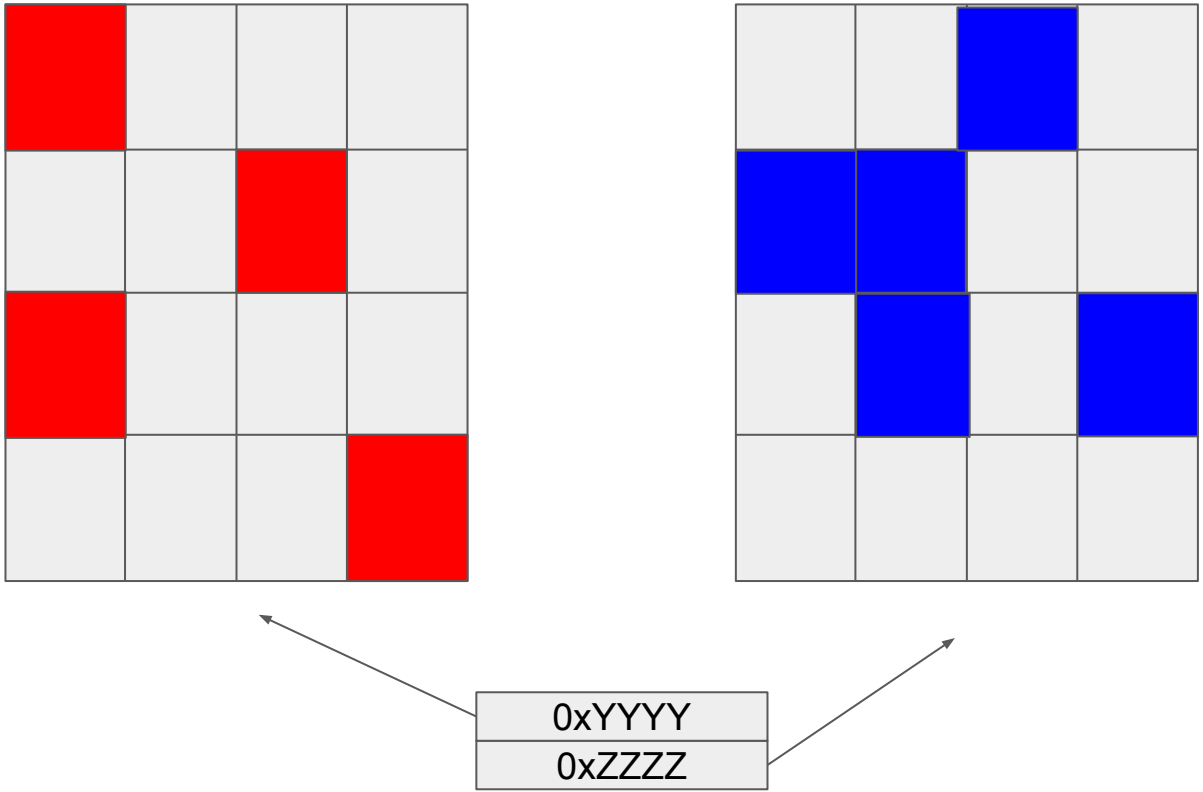
- Huge page aware allocator
- Uses madvise hints
- Provides good profiling APIs
- Everyone says it’s well tuned

# Mesh

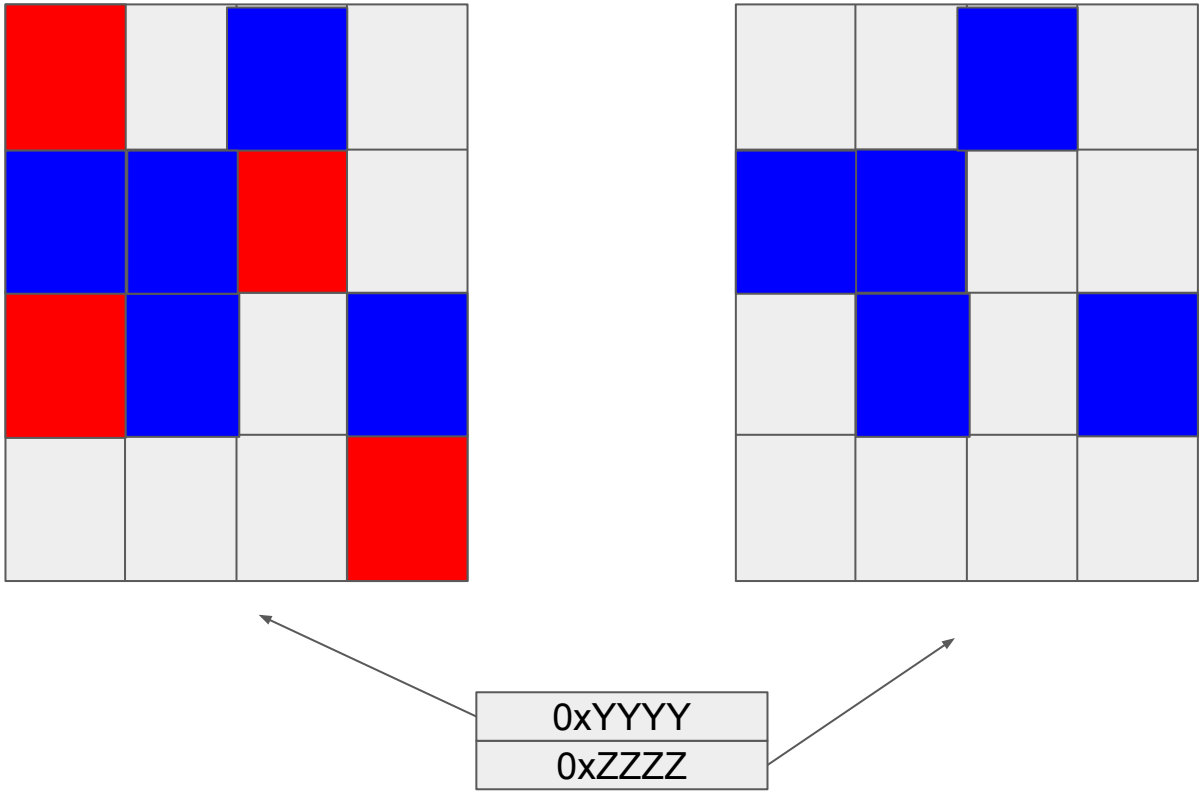
- Challenge - C/C++ allocators cannot move objects the way java or other managed runtimes can.
  - In a managed runtime, you can solve fragmentation by having the GC move live objects to a dense region
- Solution: Mesh pages together
- Mesh does not support hugepages
  - “LLAMA reduces fragmentation over Mesh on huge pages by an order of magnitude”
    - Maybe they mean compared to Mesh without hugepages?
    - Maybe they extend Mesh in some way?

Bobby Powers, David Tench, Emery D. Berger, and Andrew Mc-Gregor. 2019. Mesh: Compacting Memory Management for C/C++ Applications.

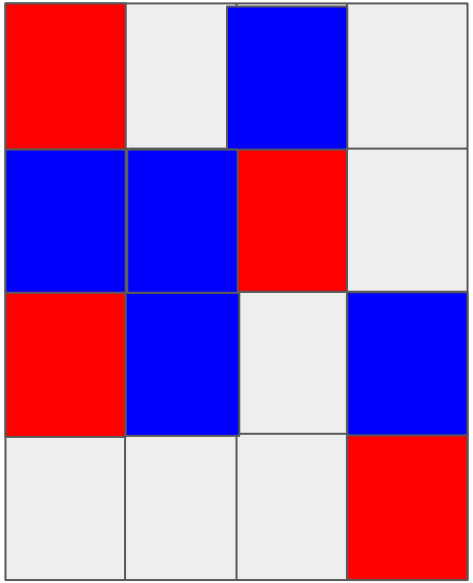
# Mesh (visualized)

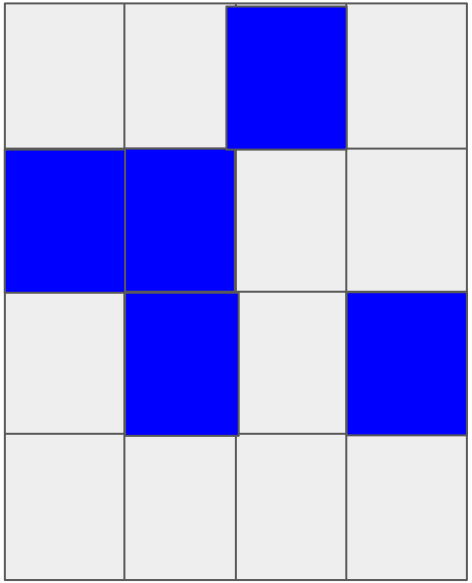


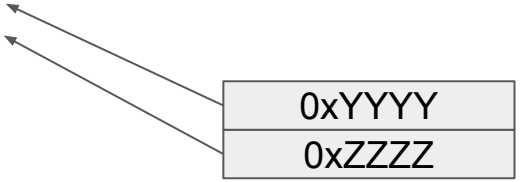
# Mesh (visualized)



# Mesh (visualized)







# Learning to predict lifetimes

- The key idea of this paper is to use learning techniques to predict lifetimes and place objects with similar lifetime on the same page
- Prior approaches - build table of allocation sites, collect lifetime info via instrumentation
  - Overhead
  - Accuracy
  - Instability - rebuilding/libraries/etc can modify the stack traces in a way that makes stability hard
- This work - use learning to predict lifetimes based on the stack instead (assumes symbols are present)

# Collecting training data

- Connect to some application for a sampling period and observe allocations and lifetimes
  - Aggregating across this data requires being able to correlate stacks across different software versions/code changes
- TCMalloc is instrumented to collect a fixed set of features and TCMalloc's built-in heap profiling tools are augmented to generate training data

# Learning from the Stack

- LSTM used to predict lifetime from stack
- Tokenization based on common C++ separators (, | :: | …). Add a “@” token between stack frames
- Only “common” tokens are preserved
- Embedder trained to group “similar” tokens together - to allow for generalization when encountering new stack traces

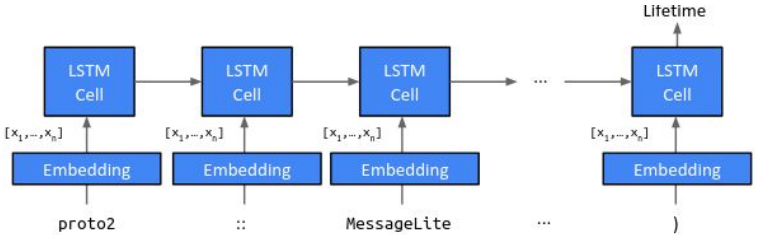



**Figure 4.** LSTM-based model architecture

# Low latency prediction

- Hash stack trace with return address, stack height, and object size
- Lookup cached prediction
- Use compiled model (runtime in 100s of us)
- Hashes might cause false hits, so periodically purge

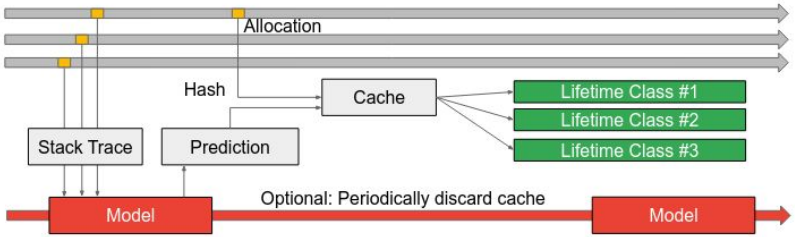
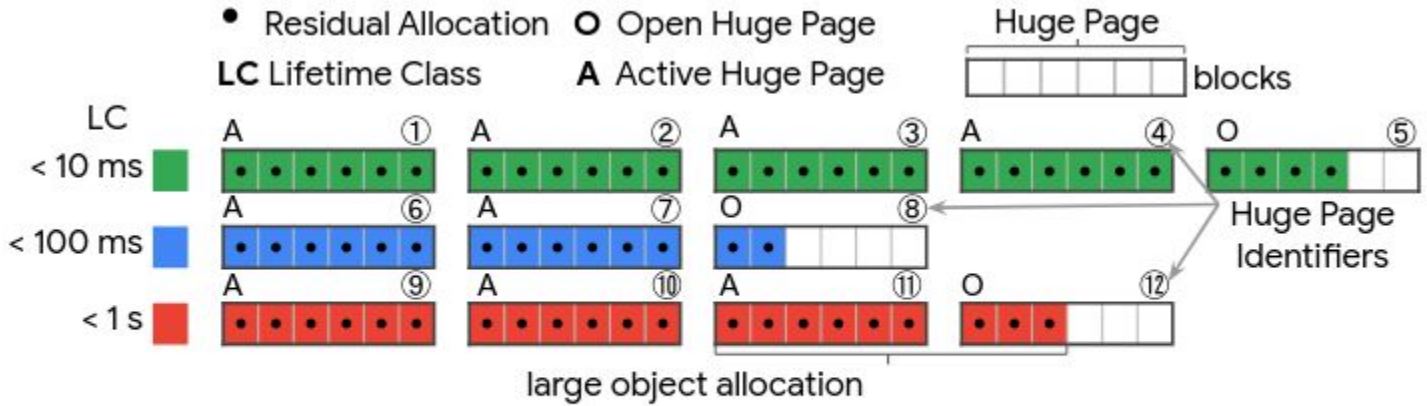


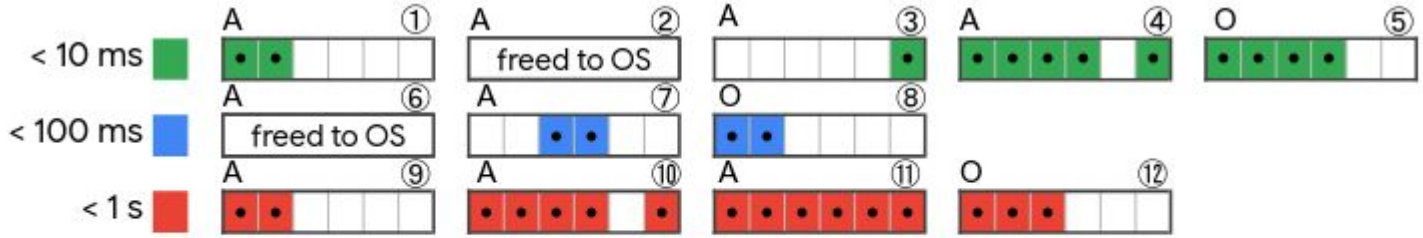

**Figure 7.** High-level overview of low-latency prediction. We use the model only when the hash of the current stack trace is not in the cache. Discarding cache entries periodically helps dynamically adapting to workload changes.

# Designing a lifetime aware allocator



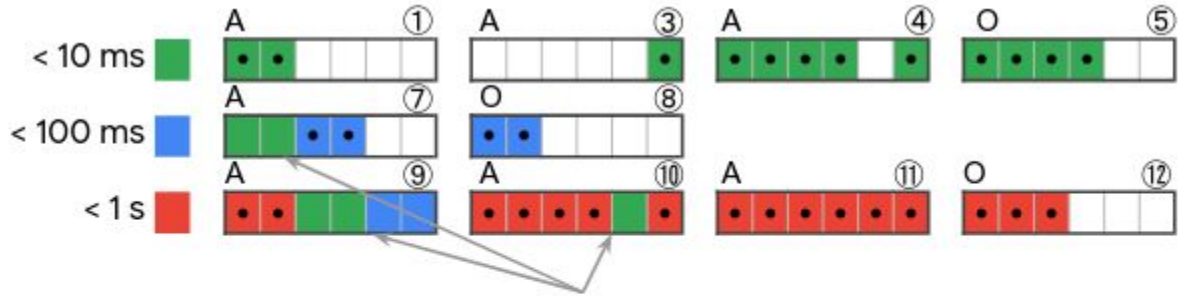

**(a)** Initial allocations. Huge pages are bump-pointer allocated into LC regions. Each huge page is first filled with same LC blocks, marked residual with a dot.

# Designing a lifetime aware allocator



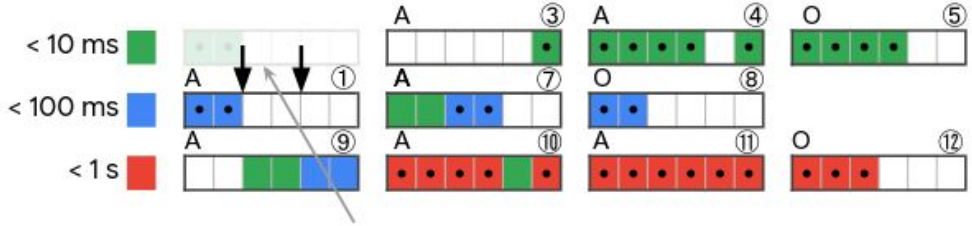

**(b)** After objects free, some blocks and huge pages are free (white). LLAMA immediately returns free huge pages to the OS to control maximum heap size.

# Designing a lifetime aware allocator

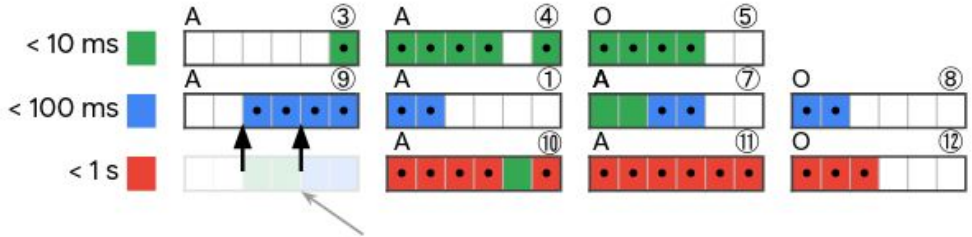



**(c)** Subsequent allocations of shorter LC small objects first fill free blocks in the highest LC in A(ctive) huge pages 9 and 10, and then blocks in huge page 7. These blocks are not residual (no dot) and expected to be freed before the residual blocks. O(pen) pages 5, 8, and 12 are ineligible for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

# Lifetime Class Promotion/Demotion




**(d)** When huge page 1's deadline expires, residual blocks are still live (mis-prediction). 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.

# Evaluation

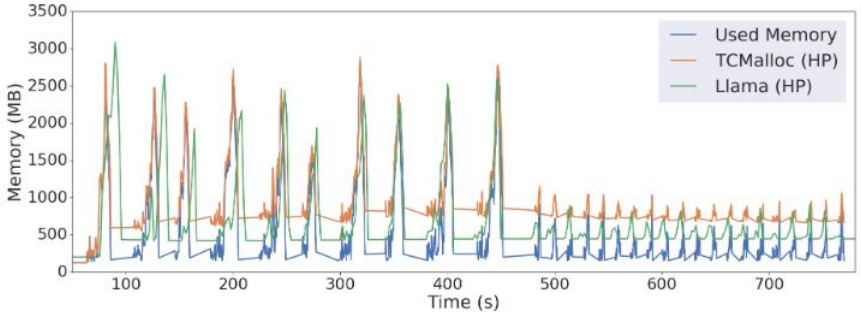
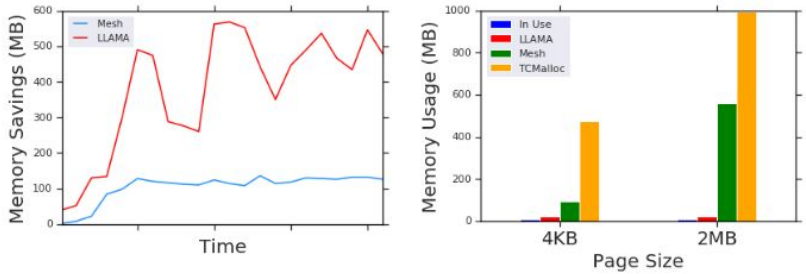



**Figure 8.** LLAMA reduces huge page (HP) fragmentation compared to TCMalloc on the Image Processing Server. TCMalloc numbers optimistically assume all free spans are immediately returned to the OS, which is not the case.




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

**Figure 12.** LLAMA reduces fragmentation compared to Mesh.

# Conclusion

- Can you combine life time prediction with meshing?
- Can you use the learned model to recompile the application to tag allocations with lifetime hints directly?