Memory and Object Management in RAMCloud

Steve Rumble

December 3rd, 2013

RAMCloud Introduction

- **General-purpose datacenter storage system**
- **All data in DRAM at all times**
- **Pushing two boundaries:**
  - Low Latency: 5 – 10μs roundtrip (small reads)
  - Large Scale: To 10,000 servers, ~1PB total memory
- **Goal:**
  - Enable novel applications with 100 – 1,000x increase in serial storage ops/sec
- **Problem:**
  - How to store data while getting high performance, high memory utilisation, and durability in a multi-tenant environment?

Thesis & Key Results

- **Structuring memory as a log allows DRAM-based storage systems to achieve:**
  - High allocation performance
  - High memory efficiency
    - Even under changing workloads
  - Durability

- **RAMCloud's log-structured memory:**
  - 410k durable 100-byte writes/s using 90% of memory
  - 2% median latency increase due to management
  - Applicable to other DRAM-based systems

Contributions

- **Log-structured memory**
  - High performance, high memory utilisation, durability
- **Two-level cleaning**
  - Durability with low disk & network I/O overhead
- **Cost-Benefit Improvements**
  - Improved heuristic for selecting segments to clean
- **Parallel cleaning**
  - Fast memory allocation, overheads off critical path
- **Cleaner balancing**
  - Policies for choosing how much of each cleaner to run

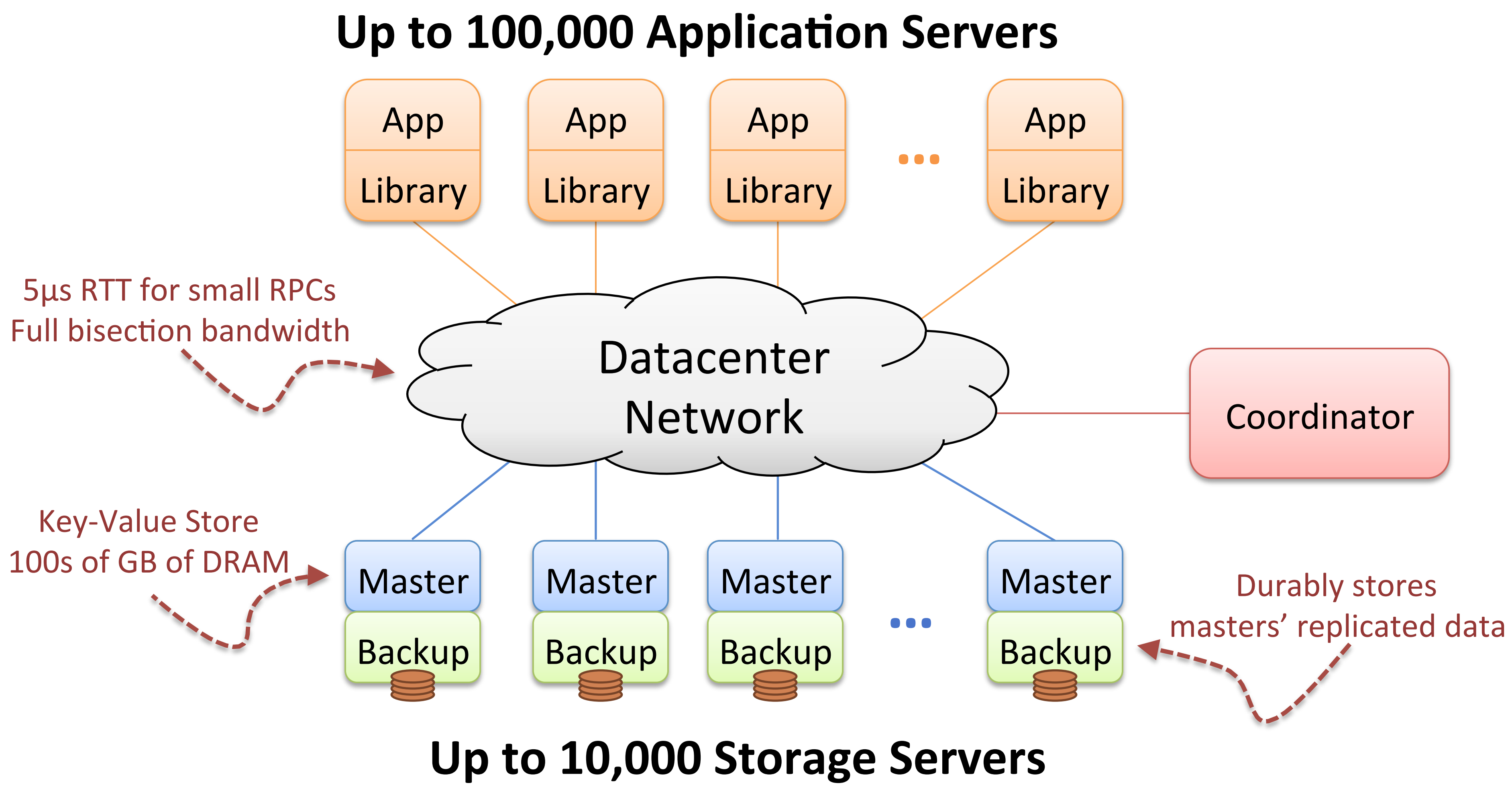
Outline

- **RAMCloud Background**
- **Motivation**
  - Goals & problems with current memory managers
- **Contributions**
  - Log-structured memory
  - Two-level Cleaning
  - Evaluation
  - Cost-Benefit Improvements
- **Conclusion**
  - Related Work
  - Future Work
  - Acknowledgements
  - Summary

Outline

- **RAMCloud Background**
- **Motivation**
  - Goals & problems with current memory managers
- **Contributions**
  - Log-structured memory
  - Two-level Cleaning
  - Evaluation
  - Cost-Benefit Improvements
- **Conclusion**
  - Related Work
  - Future Work
  - Acknowledgements
  - Summary

RAMCloud Architecture



Distributed Key-Value Store

- **Data model: key-value**
  - Key: 64KB binary string
  - Value: Binary blob up to 1MB
  - Keys scoped into tables
    - Tables may span multiple servers ("tablets")
  - Addressing: (tableId, "key")

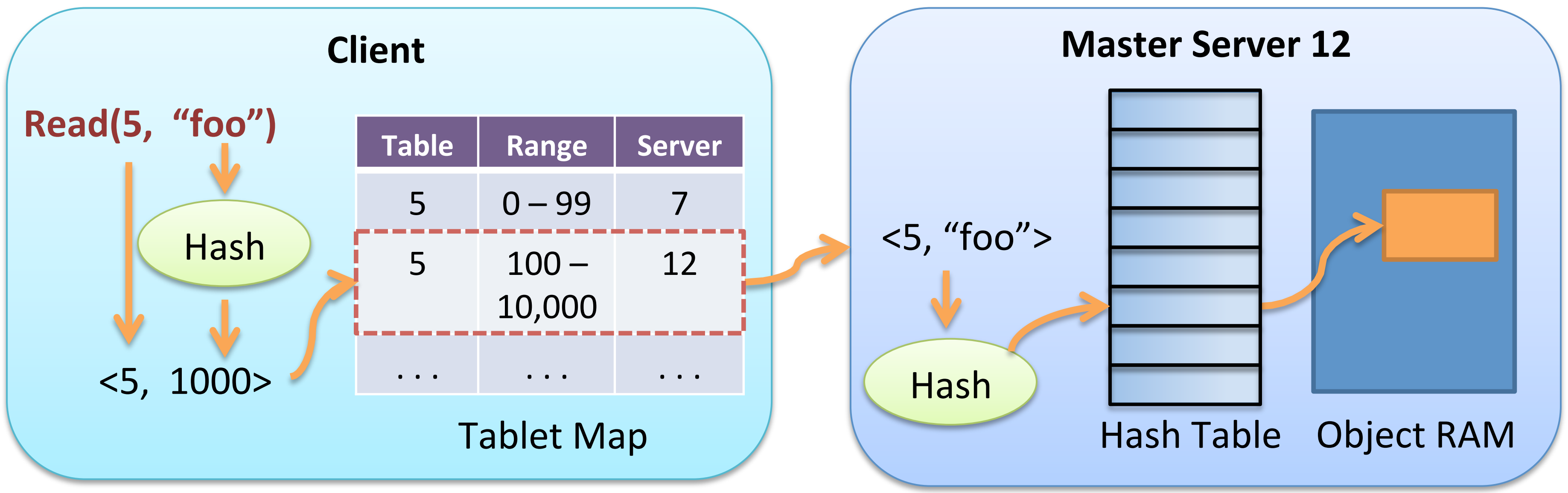


Table	Range	Server
5	0 – 99	7
5	100 – 10,000	12
. . .	. . .	. . .

Outline

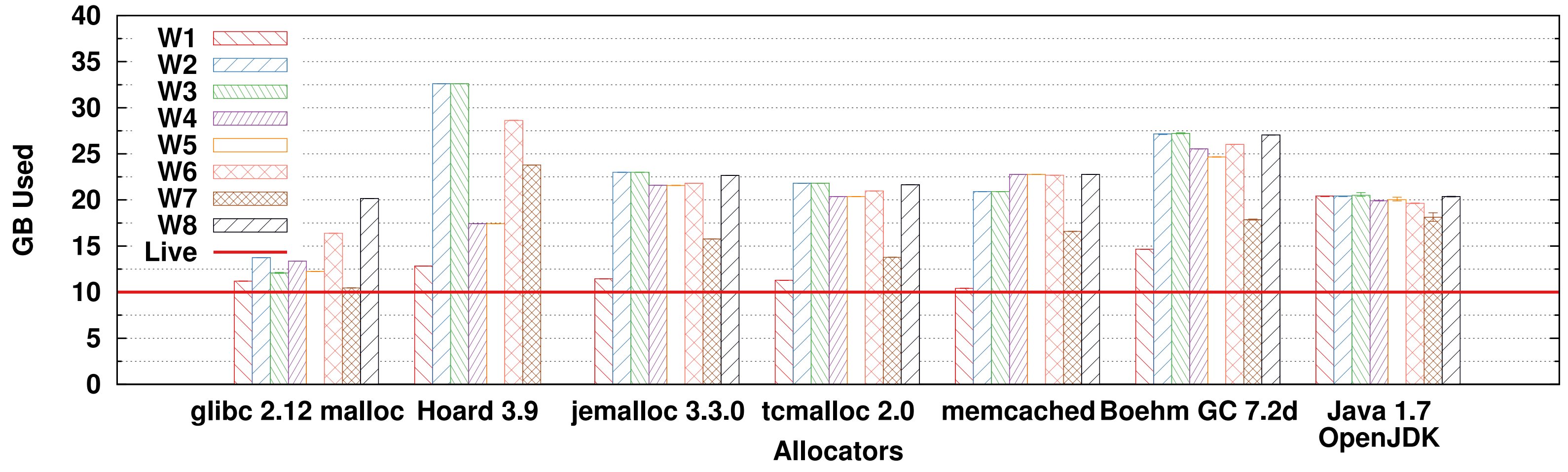
- **RAMCloud Background**
- **Motivation**
  - Goals & problems with current memory managers
- **Contributions**
  - Log-structured memory
  - Two-level Cleaning
  - Evaluation
  - Cost-Benefit Improvements
- **Conclusion**
  - Related Work
  - Future Work
  - Acknowledgements
  - Summary

Memory Management Goals

- **Problem**
  Need a way to manage memory in RAMCloud that:

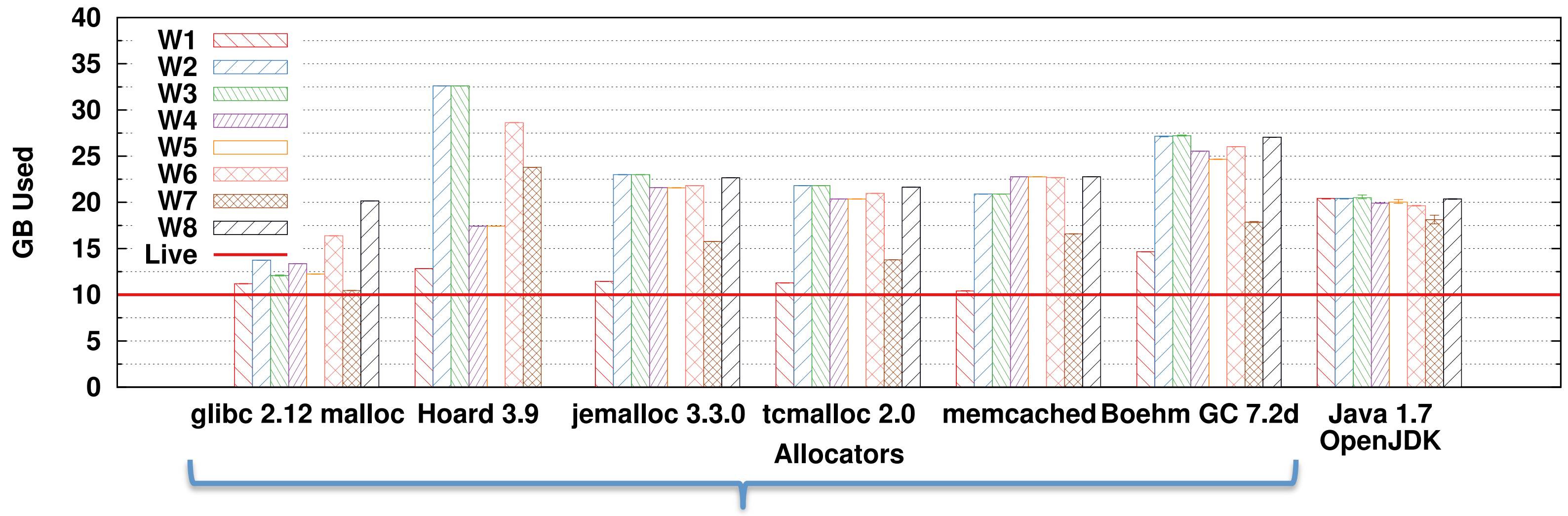
  1. Does not waste space
     (DRAM is expensive)
  2. Has high throughput
     (Supports high write rates)
  3. Adapts to changing workloads
     (Gracefully/predicatbly handles workload changes)
  4. Accommodates backups
     (Stored data must be persisted for durability/availability)

Existing Allocators Unsuitable

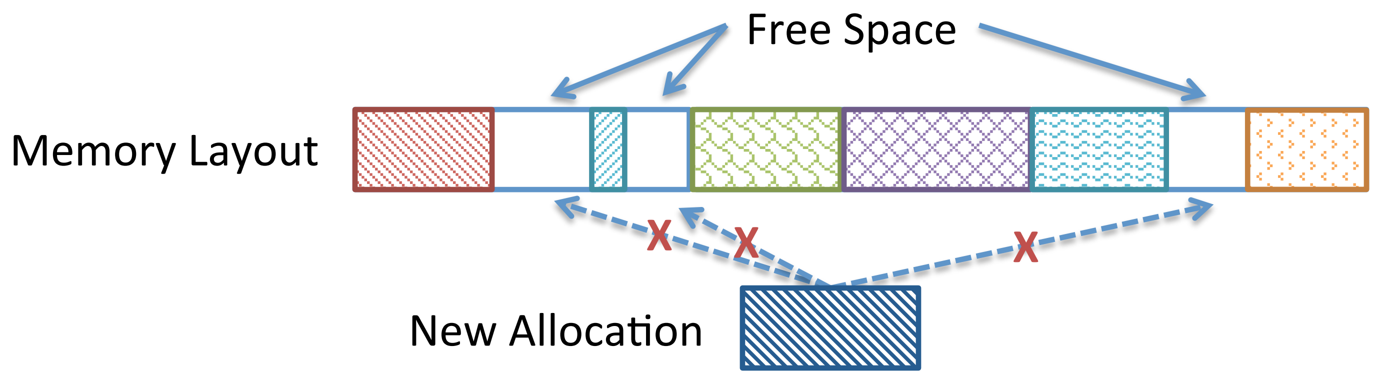


- **Existing allocators handle workload changes poorly**
  - All waste 50% of memory, or more
  - E.g., W2: Replace 100-byte objects with 130-byte
  - “High performance” allocators tend to be worse

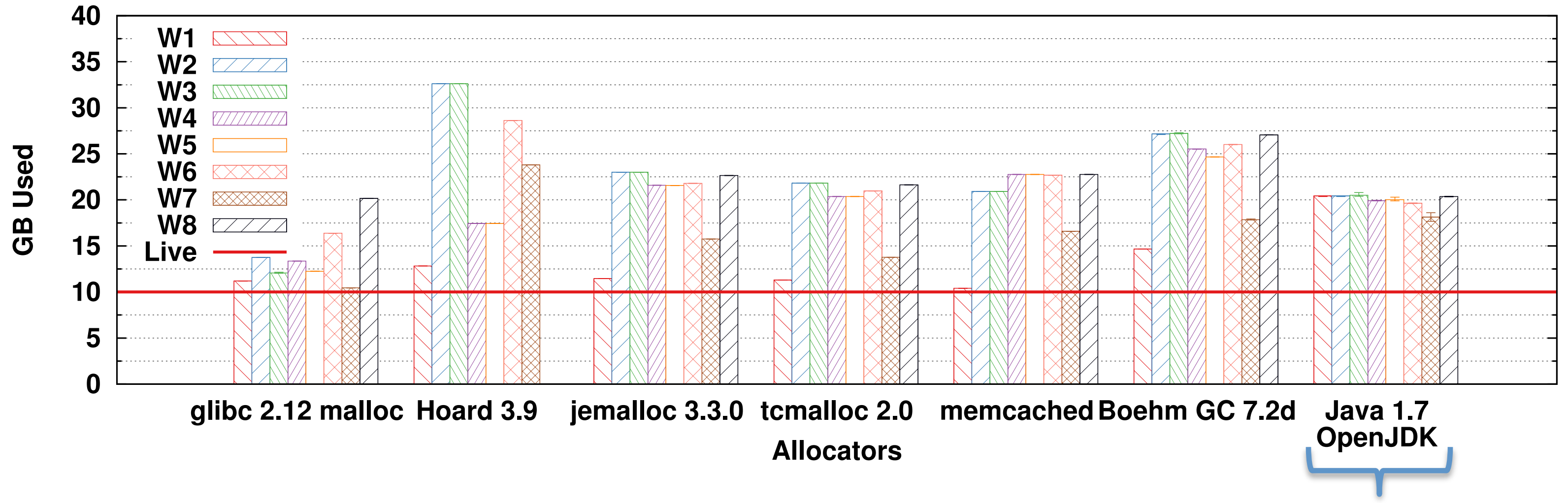
Fragmentation



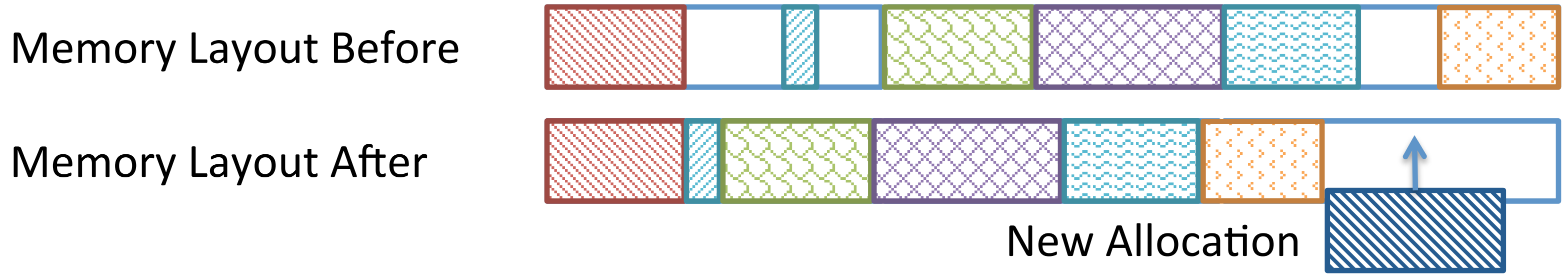
- **Non-copying allocators cannot relocate allocations**
  - Fragmentation wastes memory



Copying Collectors



- **Copying garbage collectors can defragment memory**



- **Why still so wasteful? Garbage collection is expensive!**
  - Scan memory, copy live data, update pointers
  - Cheaper when space used inefficiently: overcommit 3-5x

Goals Revisited

- **Problem**
  Need a way to manage memory in RAMCloud that:

  1. ~~Does not waste space~~
     All allocators wasted 50% of memory or more
     Cannot trade off performance for efficiency in Java's collector

  2. ~~Has high throughput~~
     Copying garbage collection is expensive

  3. ~~Adapts to changing workloads~~
     Might waste a few %, might waste 50% or more, might crash

  4. ~~Accommodates backups~~
     Existing allocators are for volatile memory

Space/Time Dilemma

- **Efficiency/Adaptability → copying memory manager**
  - Defragment memory, adapt to workload changes
- **Problem: Fundamental space vs. time trade-off**

	Low Efficiency	High Efficiency
Low Performance	Easy	Medium
High Performance	Medium	Hard

↑ Garbage Collectors (Medium) ↑ RAMCloud (Hard)

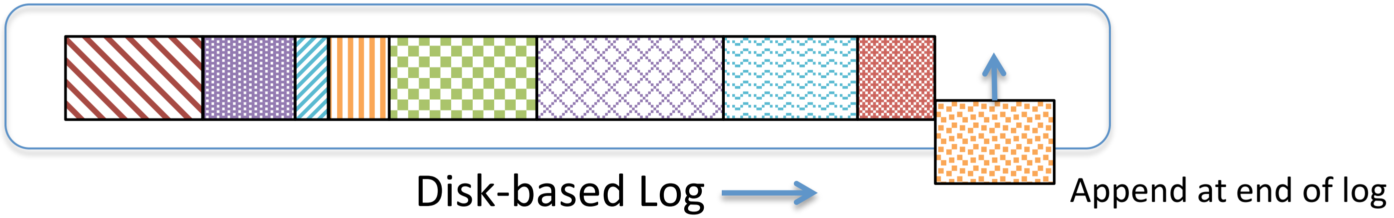
- **Insight: Storage systems have key advantages**
  - Pointer use is restricted (indexing structures only)
    - Truly incremental GC (need not scan all of memory)
  - Allocations are explicitly freed
- **Should be able to build faster/more efficient manager**

Outline

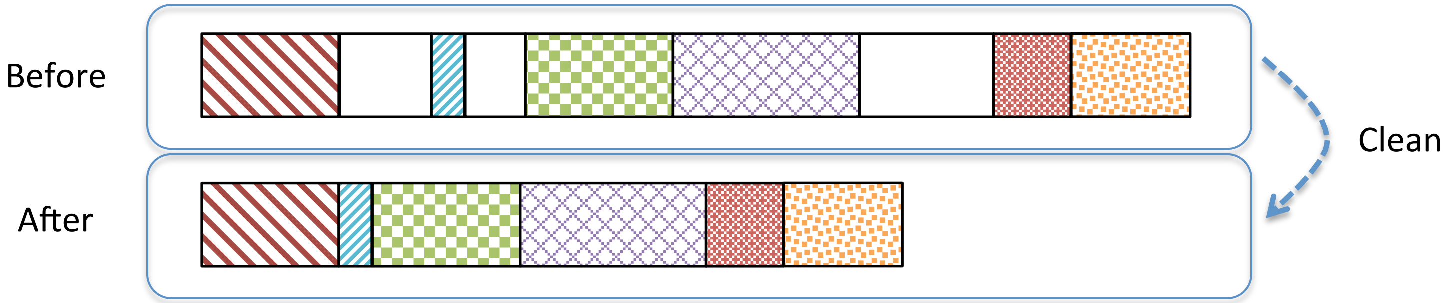
- **RAMCloud Background**
- **Motivation**
  - Goals & problems with current memory managers
- **Contributions**
  - Log-structured memory
  - Two-level Cleaning
  - Evaluation
  - Cost-Benefit Improvements
- **Conclusion**
  - Related Work
  - Future Work
  - Acknowledgements
  - Summary

Durability: Log Structure

- **Durability → Writing to disks → Log structure**
  - Large sequential I/Os for high performance
  - Append-only: no update in place → no random I/Os

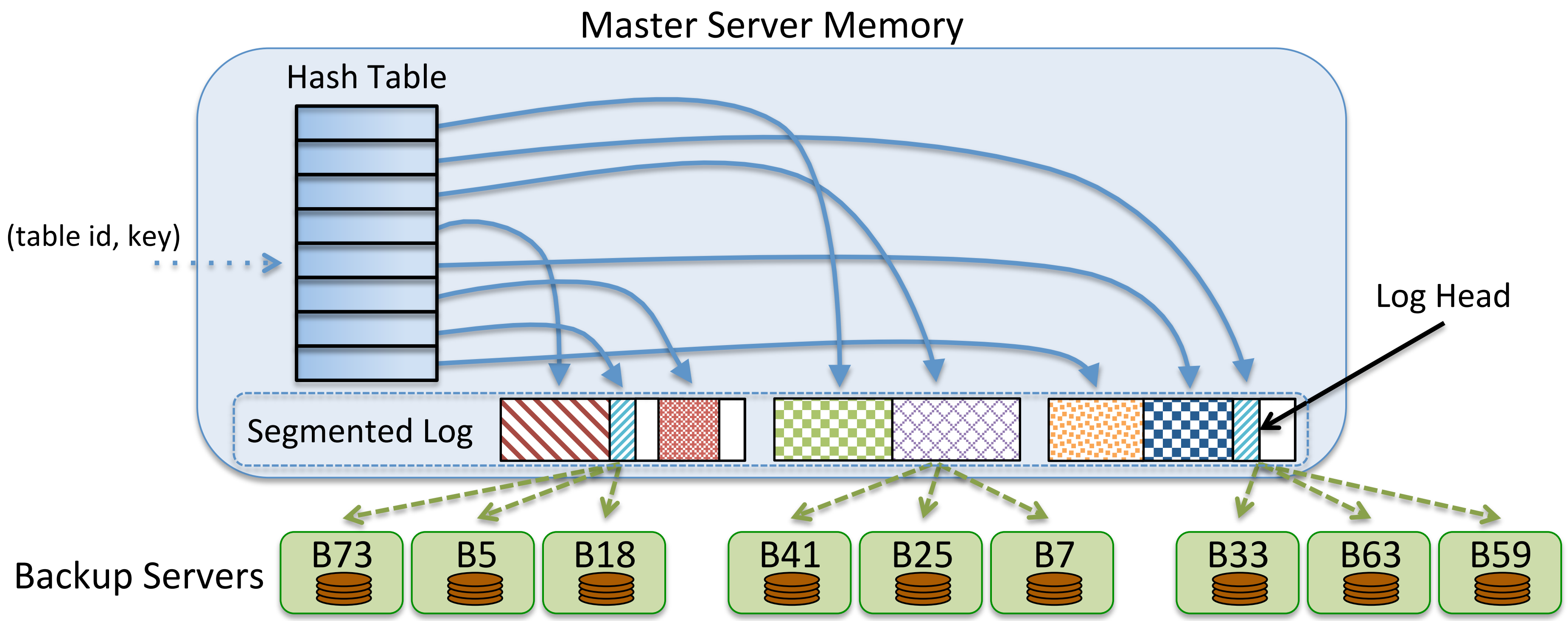


- **Logging requires a defragmenter (*cleaner*)**
  - Reclaims dead space, curbs length of log



- **Insight: Already need a copying allocator for disk**
  - Can use the same technique to manage DRAM

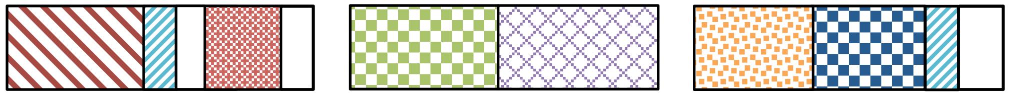
Log-structured Memory



- Master memory: hash table & segmented log
  - Segments are 8MB
- Each segment replicated on 3 remote backups
  - Unified log structure in DRAM & on disk (→ easy for masters to track disk contents)
- Append only: write objects to end of log (head segment)
  - No in-place updates, no small random disk I/Os
- Head segment full? Allocate another
  - If no free space, reclaim by cleaning log segments

Benefits of Segments

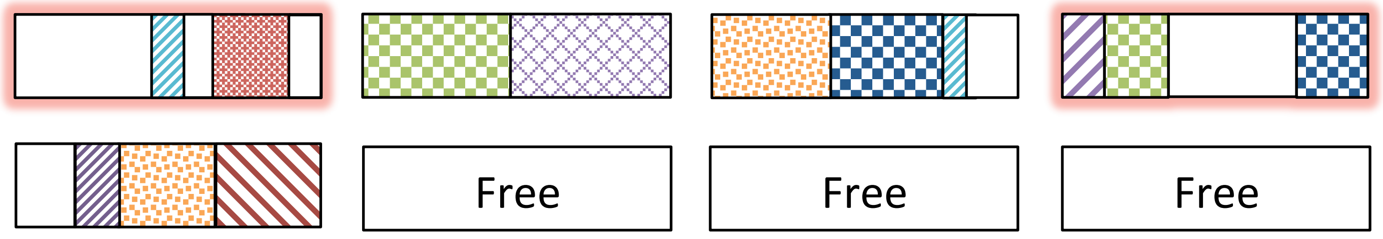
- **Key advantages of dividing log into segments:**



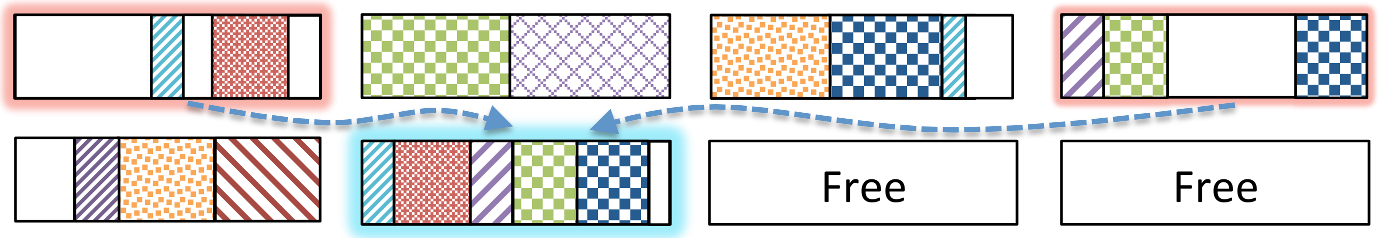
  - More efficient log cleaning
    - Can choose best segments to clean
    - Fully incremental: Process small portion of log at a time
  - High write bandwidth
    - Striped across many different backup servers
  - High read bandwidth
    - Crucial for fast crash recovery (no in-memory replication)

Log Cleaning in 3 Steps

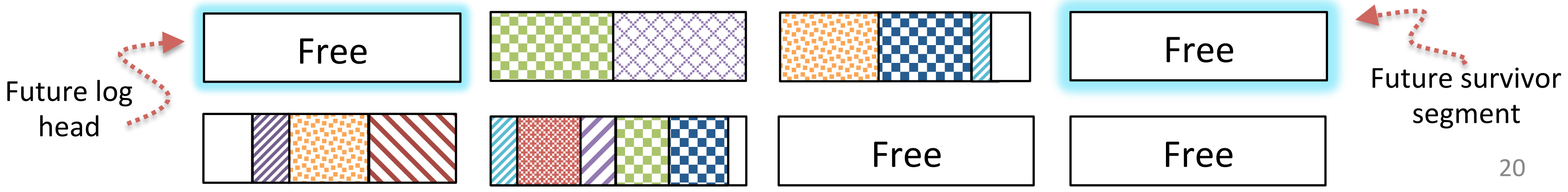
- **Cleaning incrementally defragments segments**
  1. Select segments to clean



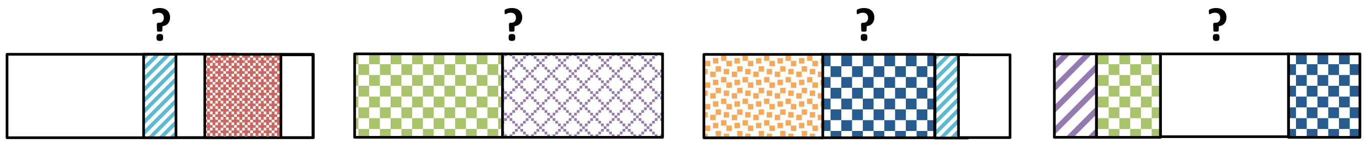
  2. Relocate survivors (live objects) to new segment(s)



  3. Free cleaned segments (reuse to write new objects)



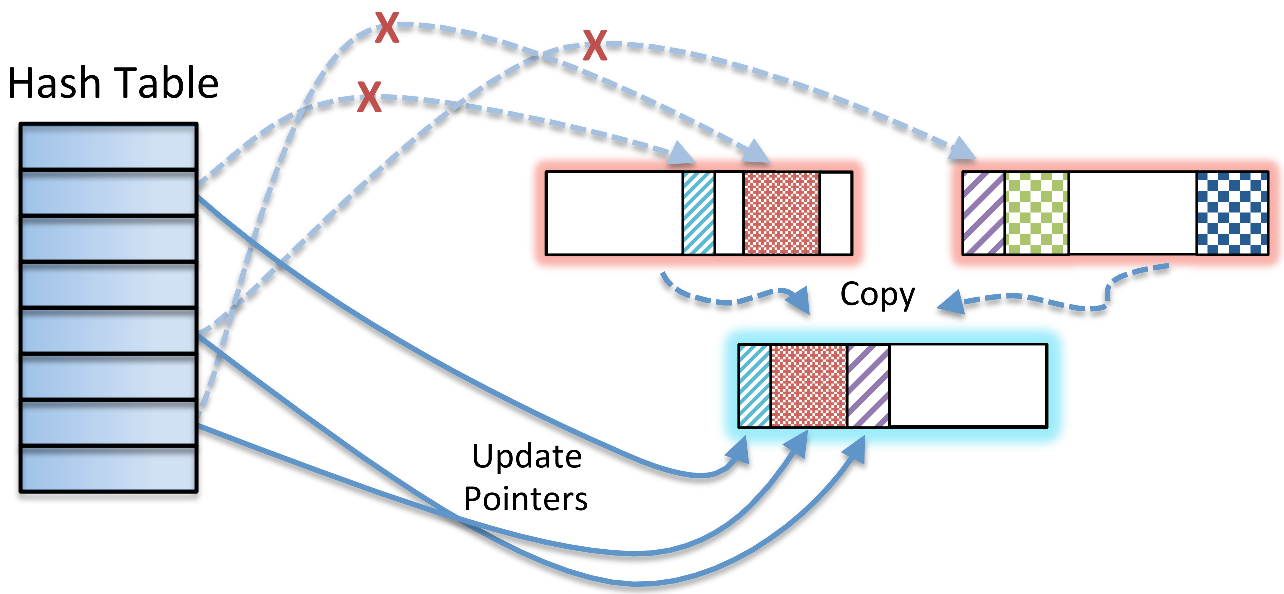
1. Selecting Segments



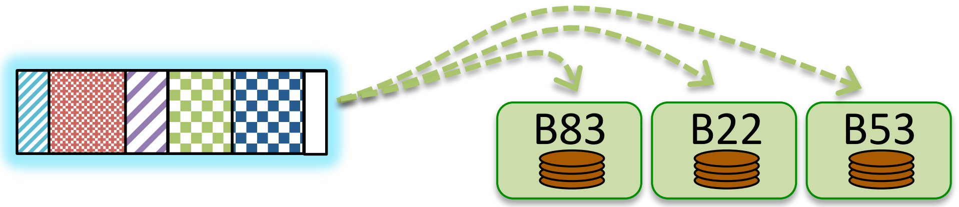
- **Greedily picking emptiest segments is suboptimal**
  - Like LFS, RAMCloud selects by cost-benefit
  - Takes free space and stability of data into account

- **Will revisit this later in the talk**
  - Slightly different formula than LFS (with a fun story)

2. Relocating Survivors



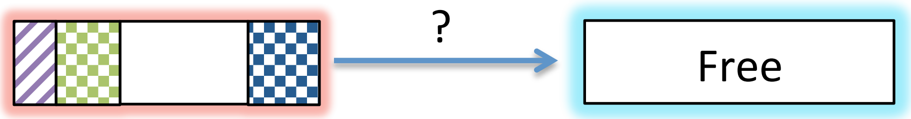
- Relocate: copy to new segment & update hash table
- When survivor segment is full, flush to backups



- Survivor objects sorted by age (not depicted)
  - Segregate objects by est. lifetime to improve future cleaning

3. Freeing Segments

- **When is it safe to free/reuse the cleaned segments?**

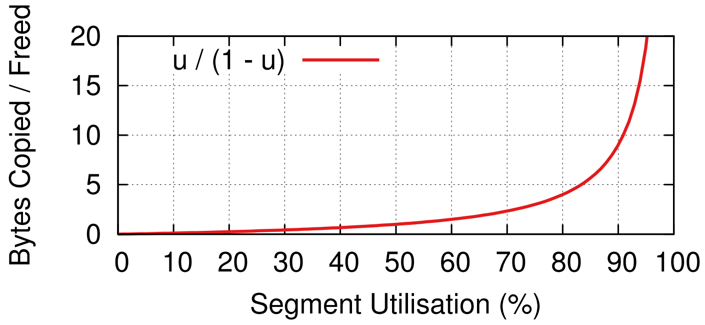


- **In-memory: when RPCs done reading them**
  - Concurrent RPCs could be reading cleaned segments
  - RCU-like mechanism: delay reuse until no readers
- **On-disk: when recovery will not try to replay them**
  - *Log digest* written with each new head segment
    - Records which segments are in the log
    - Used by recovery to figure out which segments to replay
  - Next log digest does not include cleaned segments
    - Once written, issue RPCs to backups to free disk space

Cost of Cleaning

- **Cleaning cost rises with memory utilisation**
  - *u*: fraction of live bytes in a segment

Bytes copied by cleaner	*u*	0.5	0.8	0.9
Bytes freed	1 - *u*	0.5	0.2	0.1
Bytes copied / bytes freed	*u*/(1 - *u*)	1	4	9

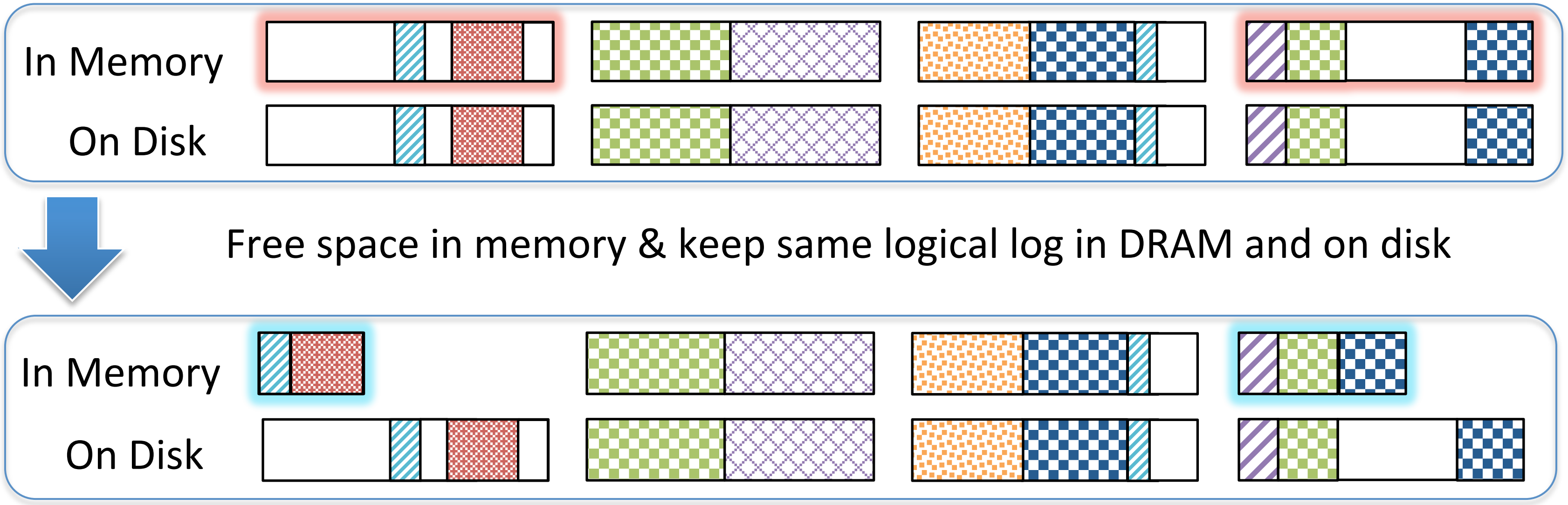


For every 10 bytes to backups: 9 from cleaner, 1 for new data
Only using 10% of bandwidth for new data!

- **Problem:**
  Cleaning bottlenecks on disk and network I/O first
  - Disk and memory layouts are the same
  - Disk & network I/O needed to reclaim space in DRAM
  - Higher *u* → more cleaning → less I/O for client writes
- **Dilemma: Higher utilisation or higher performance?**

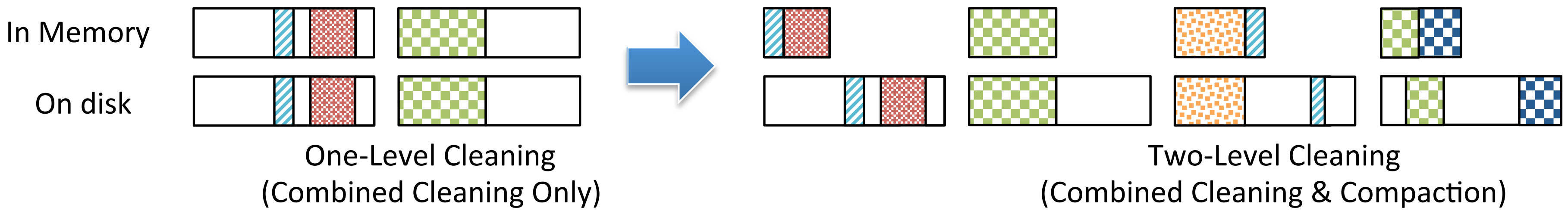
Two-Level Cleaning

- **Solution: Reclaim memory without changing disk log**
  - No I/Os to backups
- **Two cleaners:**
  1. Compactor: Coalesce in-memory segments


In Memory
On Disk
Free space in memory & keep same logical log in DRAM and on disk
In Memory
On Disk

  2. Combined Cleaner: same 3 step cleaner as before
     - Clean multiple segments, write survivors to DRAM & disks

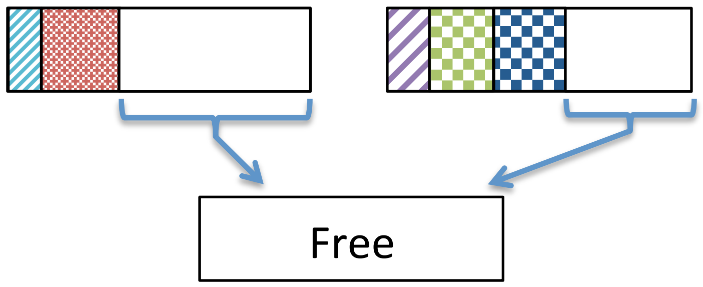
Benefit of Two-level Cleaning



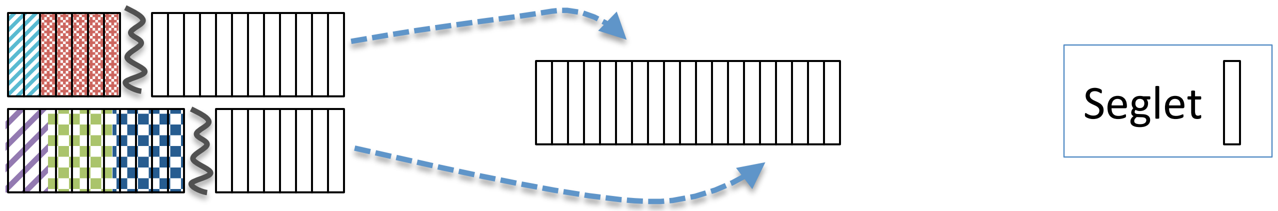
- **The best of both worlds:**
  - High utilisation of DRAM
    - DRAM has much higher bandwidth than network / disk
    - High compaction costs affordable: can copy more to free less
  - Low I/O overhead for on-disk log (avoids bottlenecks)
    - Disk has much higher capacity
    - Compaction lowers utilisation on disk (compared to memory)
    - Result: Reduced combined cleaning cost, more I/O for writes

Seglets

- **Problem:**
  How to reuse space freed by compaction?



- **Solution:**
  Discontiguous in-memory segments
  - In-memory segment: one or more 64KB *seglets*
    - Starts out with 128 seglets (8MB)
    - Compaction frees unused seglets after coalescing

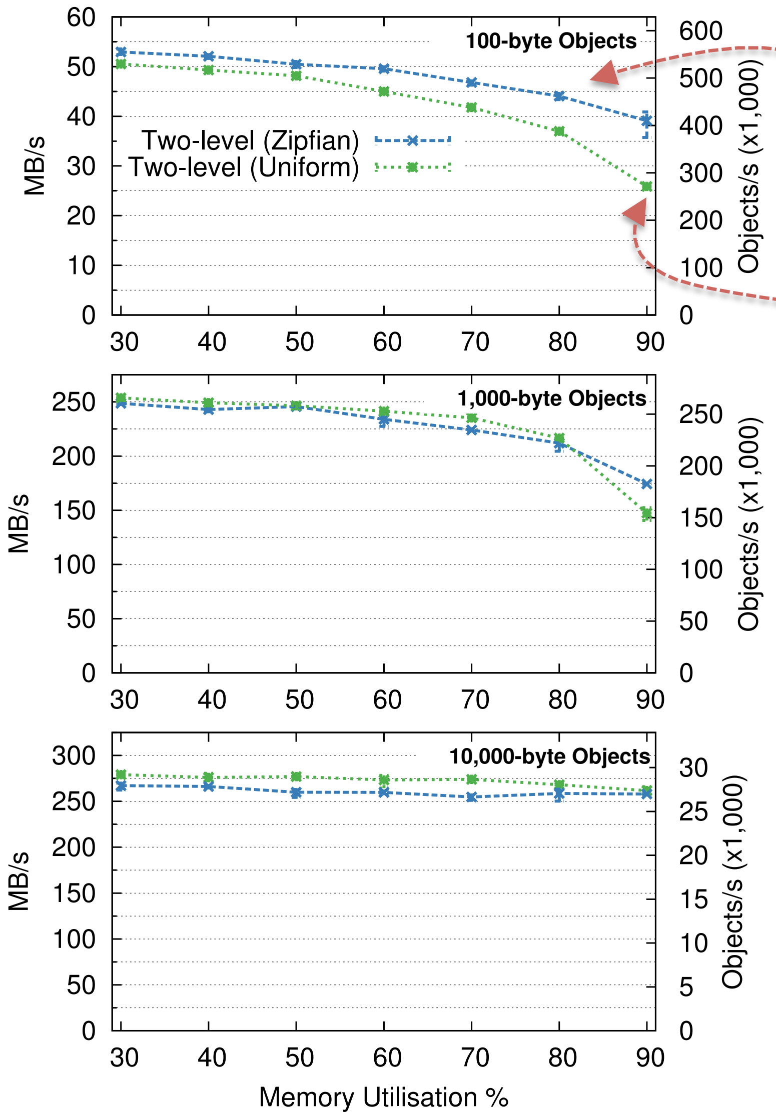


  - Discontiguity handled in software
    - Initial attempt used MMU, but too slow

Outline

- **RAMCloud Background**
- **Motivation**
  - Goals & problems with current memory managers
- **Contributions**
  - Log-structured memory
  - Two-level Cleaning
  - Evaluation
  - Cost-Benefit Improvements
- **Conclusion**
  - Related Work
  - Future Work
  - Acknowledgements
  - Summary

Client Write Throughput



Memory Utilisation	Performance Degradation (Zipfian)	Performance Degradation (Uniform)
80%	17%	27%
90%	26%	49%

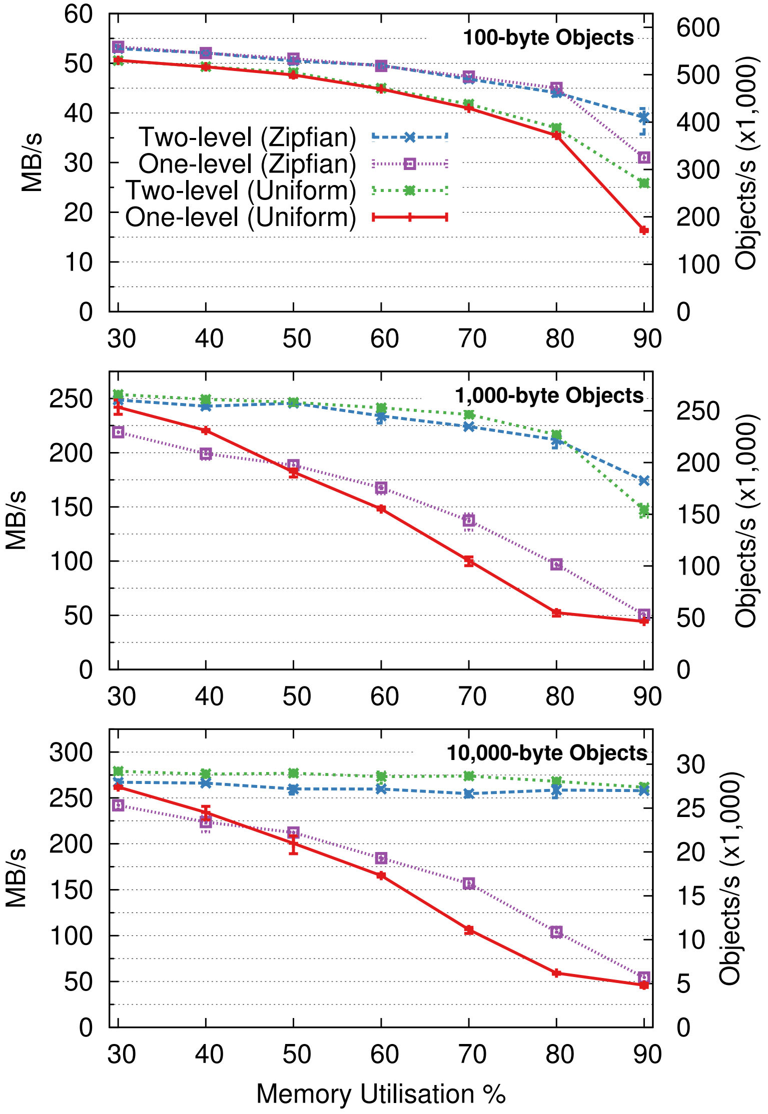
Memory Utilisation	Performance Degradation (Zipfian)	Performance Degradation (Uniform)
80%	15%	14%
90%	30%	42%

Memory Utilisation	Performance Degradation (Zipfian)	Performance Degradation (Uniform)
80%	3%	4%
90%	3%	6%

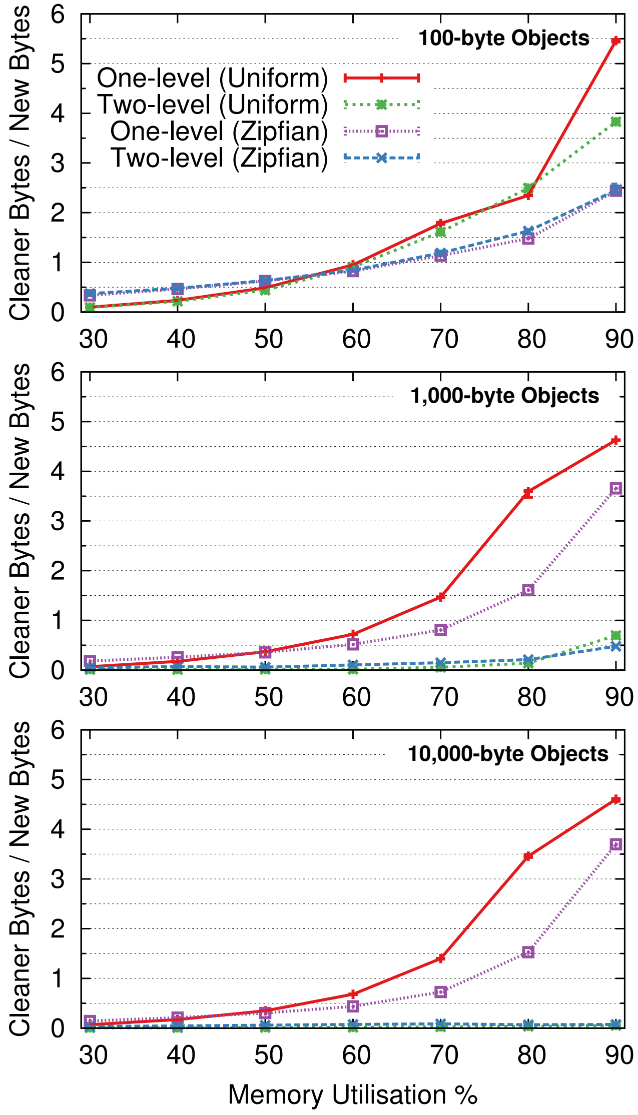
3 - 30% penalty at 90% under typical workloads

I/O Overhead Reduction

Throughput



Cleaning I/O Overhead



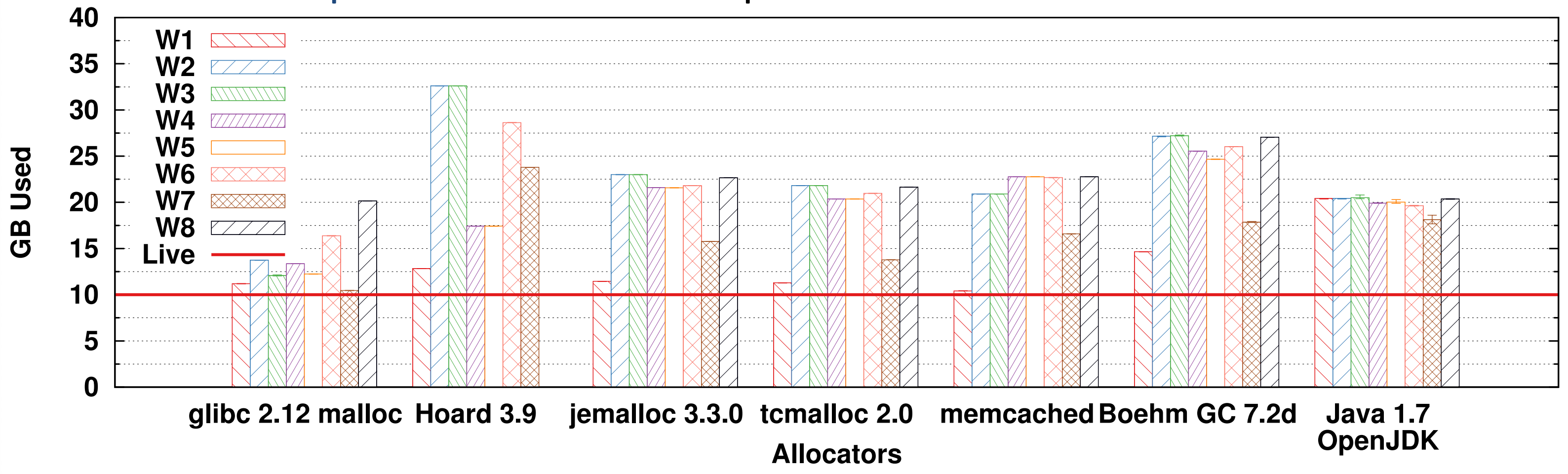
90%: 1.5 - 2.2x

90%: 6.1 - 7.2x

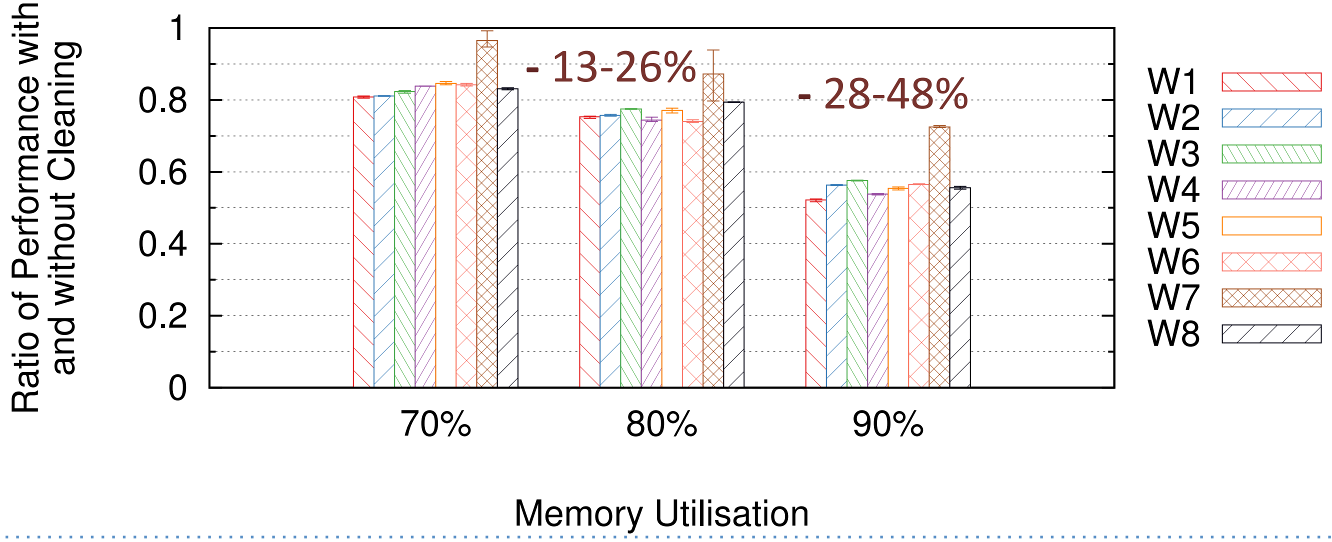
90%: 65 – 87x

Handling Workload Changes

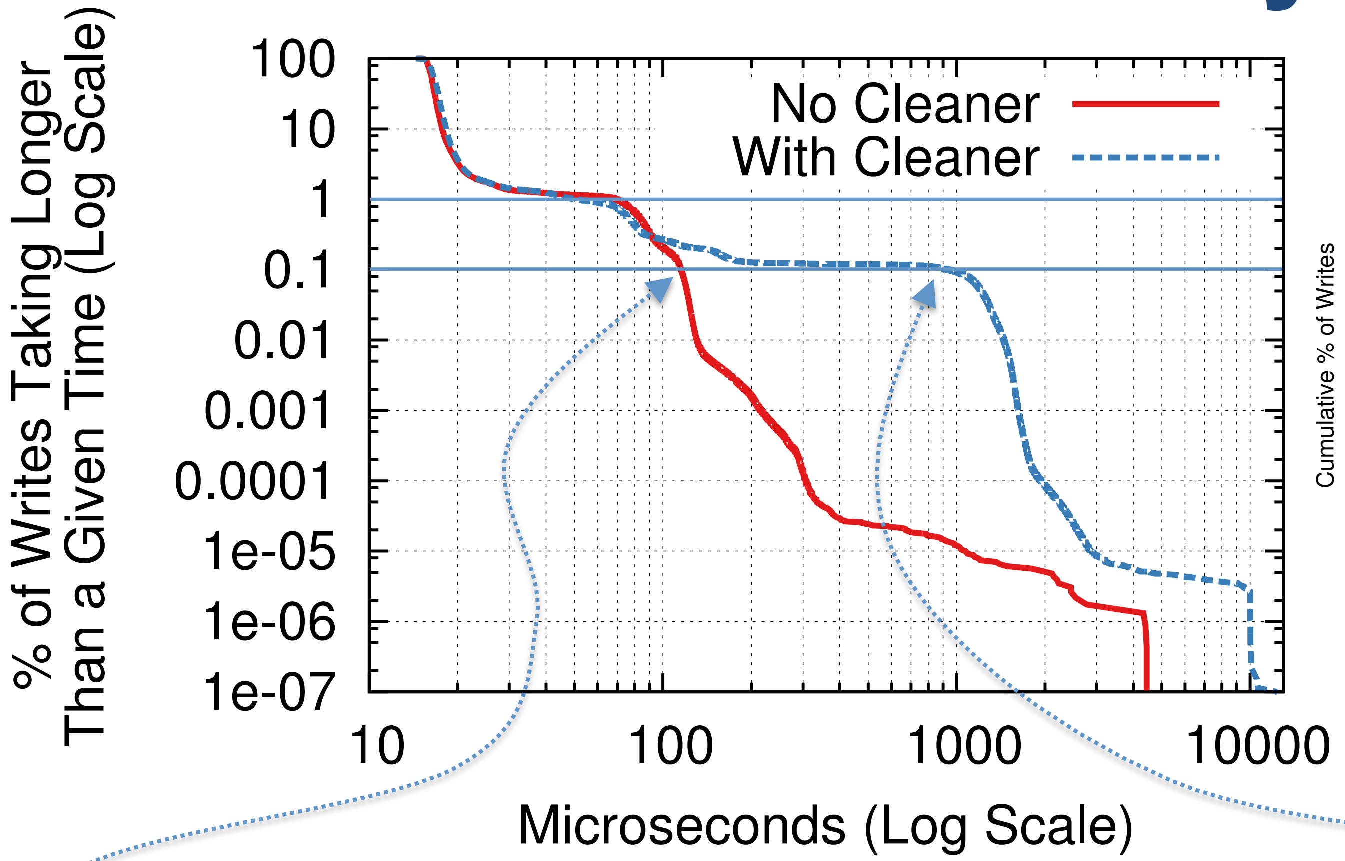
Old question: How much space is needed to store 10 GB?

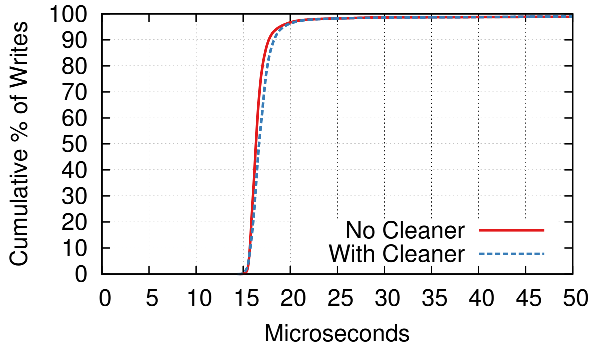


New question: How good is performance when 10 GB is X% of memory?



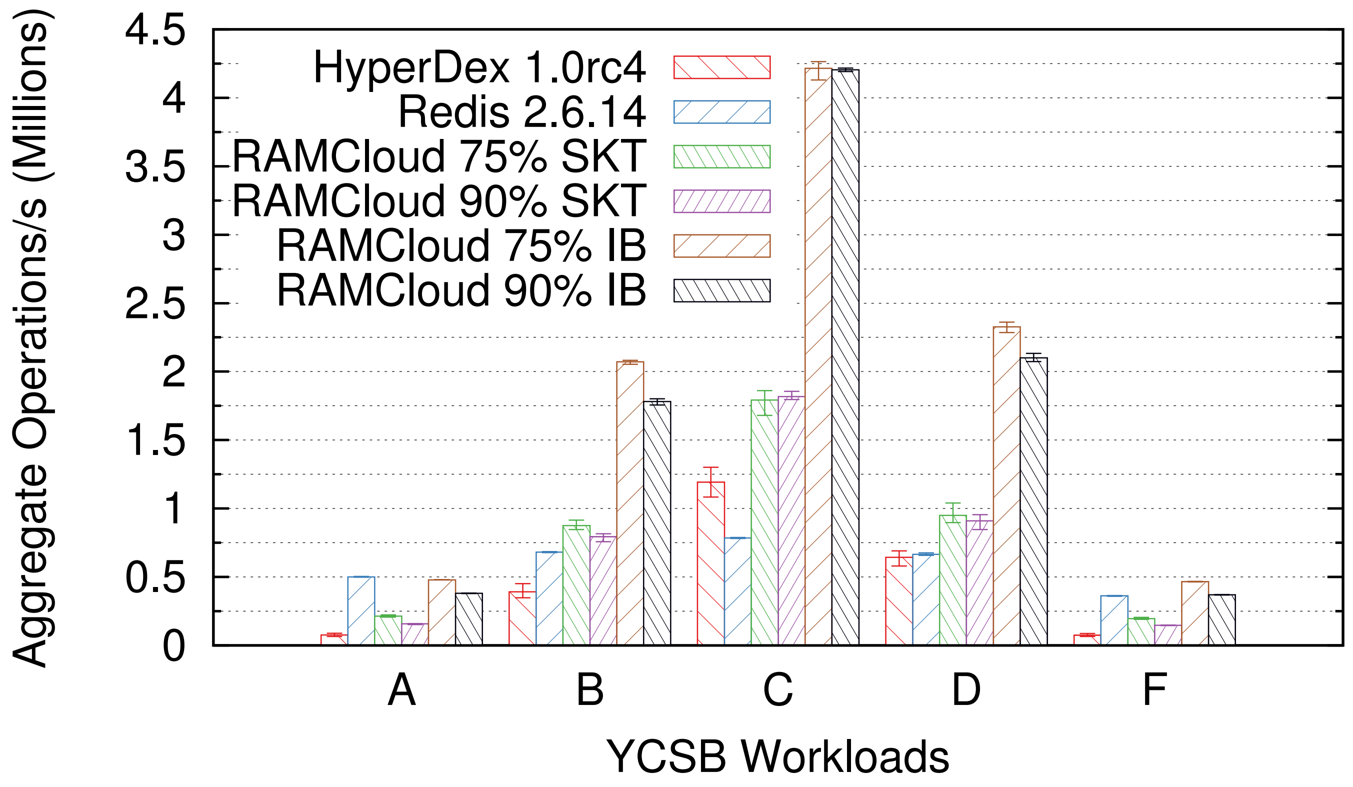
Write Latency





- **No locality, 100-byte objects, 90% util, back-to-back:**
  - Cleaning adds 2% to median latency (17.35 µs)
  - 99.9th percentile: 115 µs without cleaning, 900 µs with
    - NIC contention, backup queueing delays?

Compared to Other Systems



Workload	Description
A	50% Read 50% Update
B	95% Read 5% Update
C	100% Read
D	95% Read 5% Insert
F	50% Read 50% RMW

- **Using Yahoo!'s Cloud Storage Benchmark (YCSB)**
  - Faster in all read-dominated cases (B, C, D)
  - Faster than HyperDex in all cases, even w/o Infiniband (1.2 - 2.8x)
  - Need Infiniband to match Redis' write performance
    - Redis sacrifices consistency and durability for performance
  - At most 26% performance hit from 75% → 90% memory utilisation

LSM in Other Systems?

- **Does LSM make sense only in RAMCloud?**
  - Ported RAMCloud's log to memcached
    - Slab allocator & rebalancer → log & cleaner
- **Different use case**
  - No durability → no seglets / two-level cleaning
  - Cache: cleaner does pseudo-LRU
  - Select segments by hit rate, keep at most 75% data
- **YCSB Results:**
  - Identical throughput
  - 14% to 30% more space efficient
  - Marginal CPU overhead (5%)
  - Adapts faster to workload changes

Goals Revisited

- ✓ **High Memory Efficiency**
  - Choice is up to you
  - Experiments run up to 90% util with 25-50% perf loss
- ✓ **High Performance**
  - 410k 100-byte writes/sec at 90% utilisation
  - Competitive with other systems
- ✓ **Durable**
  - All experiments run with 3x replication
- ✓ **Adapts to workload changes**

Outline

- **RAMCloud Background**
- **Motivation**
  - Goals & problems with current memory managers
- **Contributions**
  - Log-structured memory
  - Two-level Cleaning
  - Evaluation
  - Cost-Benefit Improvements
- **Conclusion**
  - Related Work
  - Future Work
  - Acknowledgements
  - Summary

What is Cost-Benefit?

- **Cleaner needs policy to choose segments to clean**
- **LFS: a *cost-benefit* approach better than greedy**
  - Greedy: Lowest utilisation ($u \in [0,1]$)
  - Cost-Benefit: *u* and stability of data
- **Score segments by**

$$\frac{benefit}{cost} = \frac{(1-u)\times age}{1+u}$$

Stability factor (youngest file in segment)

Read segment from disk before cleaning

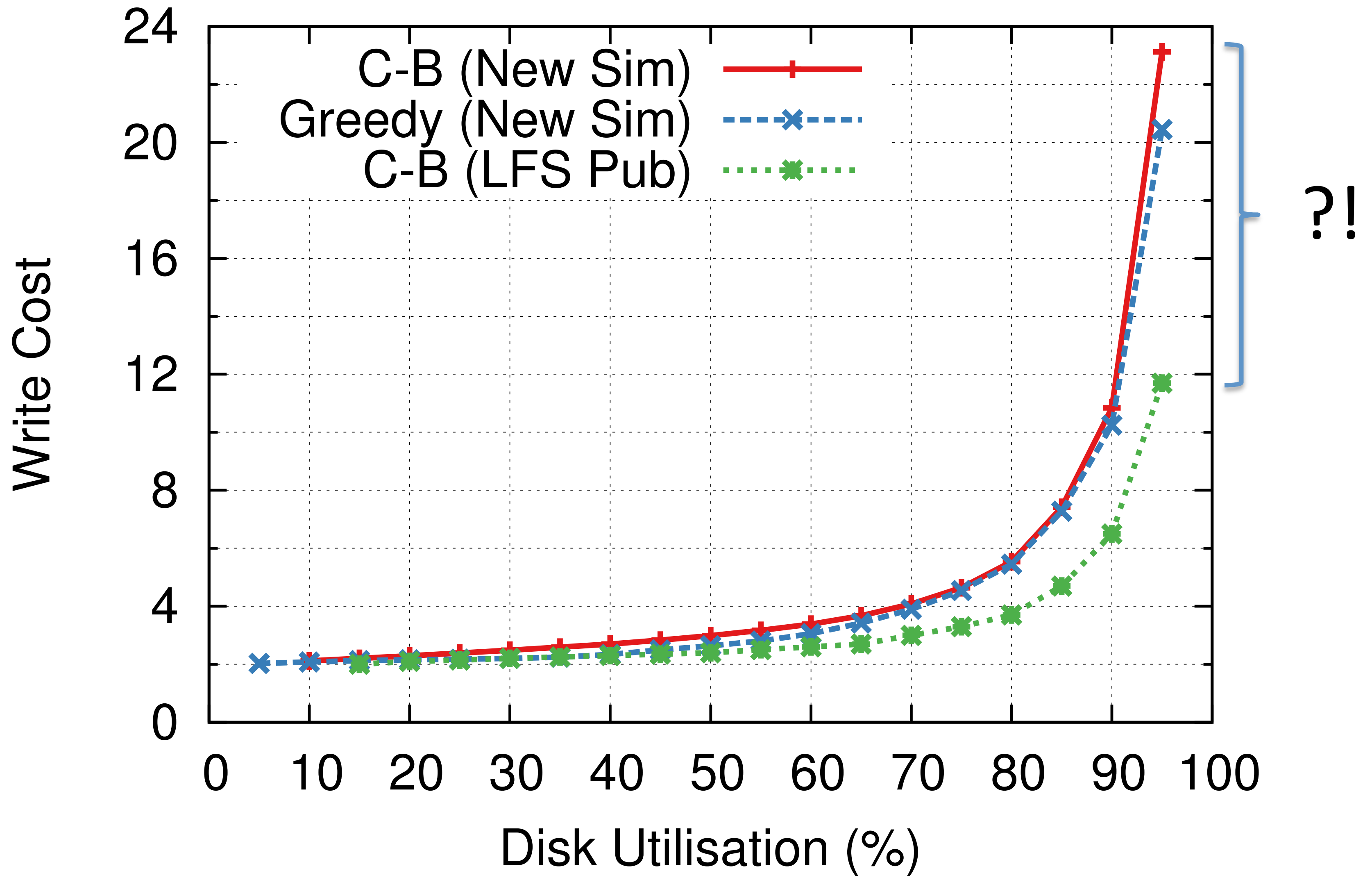
- **Choose ones with highest scores**
- **Intuition: Free space in cold segments more valuable**

LFS Simulator

- **Re-implemented LFS simulator from dissertation:**
  - Quick, fun way to:
    - Gain insight into cleaning
    - Test RAMCloud's combined cleaner
- **Simulates writing & cleaning on a LFS file system**
  - Fixed 4KB files, 2MB segments, 100 segs of live data
  - Input:
    - Disk utilisation (*u*)
    - Access pattern (uniform random, hot-and-cold, exponential)
    - Cleaning policy (greedy, cost-benefit)
  - Output: *Write Cost*
    - AKA *write amplification*
    - For LFS, ~2.0 usually optimal

$$wc = \frac{WriteIO + CleanerIO}{WriteIO}$$

Fun, but not so quick

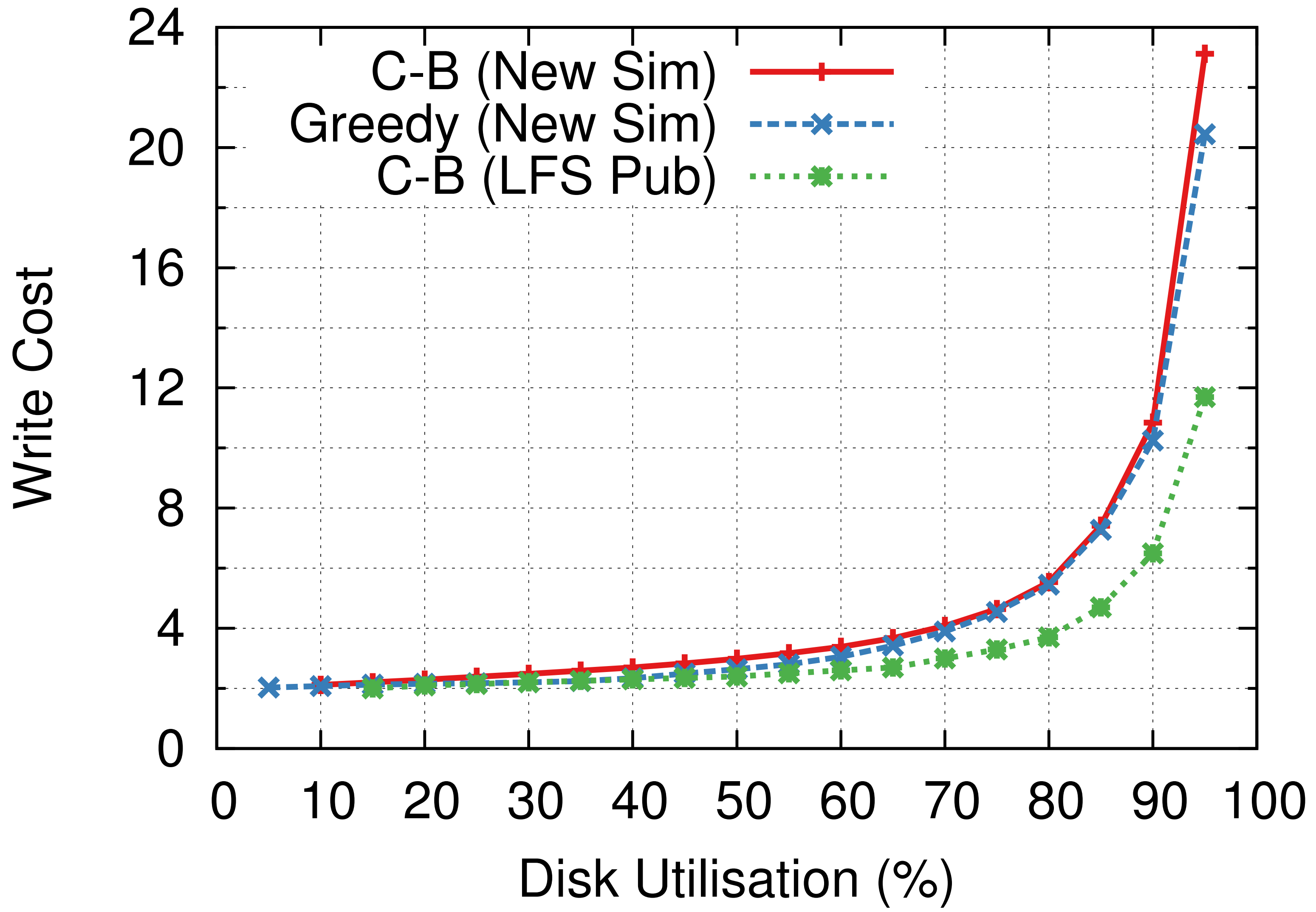


(Hot-and-Cold access pattern: 90% of writes to 10% of files)

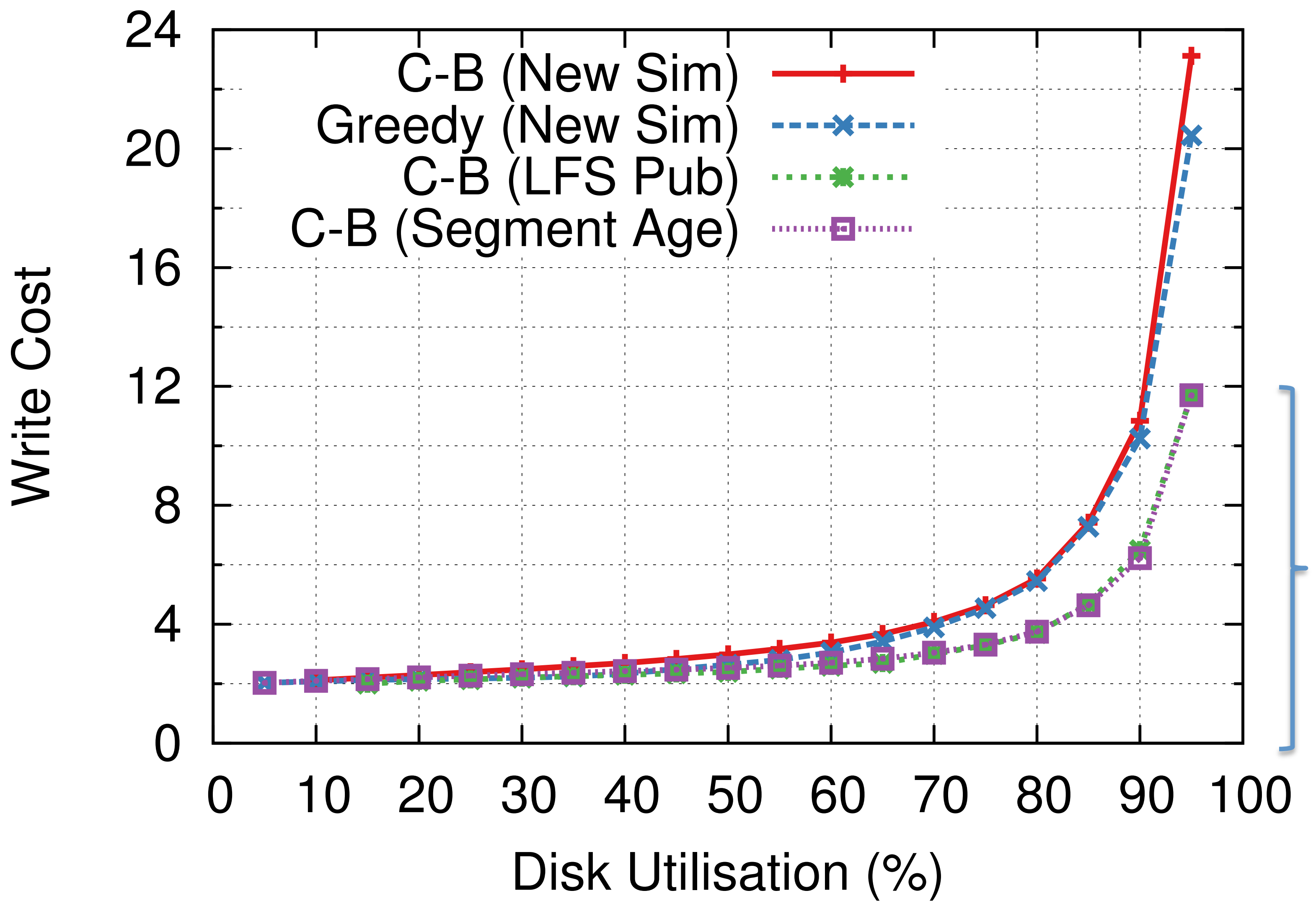
Need a Simple Explanation

- **Looked like age was dominating utilisation**
  - Forcing too many high-utilisation segments to be cleaned $age >> \frac{(1-u)}{1+u}$
- **Confident original simulator didn't use age as described**
  - RAMCloud reproduced new simulator results
- **Started looking for simple bugs**
  - Unlikely that the actual formula was drastically different
  - What subtle changes could improve cleaning?
- **Tried resetting object ages when cleaned**
  - When live object moved, object.age = now
  - Surprisingly good, but hurt future cleaning (can't sort objects by age to segregate hot/cold data)
- **Insight: resetting object ages = using the segments' ages**
  - Same as above, but can still sort objects by their ages
  - Why not try that?

Using Segment Age



Using Segment Age



Rediscovered?

- **Appears likely original simulator used segment age**
- **Supported by a later publication:**
  - "The cost-benefit policy chooses the segment which minimizes the formula

    $$\frac{1+u}{a \times (1-u)}$$

    where *u* is the utilization of the segment and *a* is the age of the segment."
    - *Improving the Performance of Log-Structured File Systems with Adaptive Methods*, SOSP '97
  - Based on descendent of original LFS simulator
- **Still unclear why nobody noticed discrepancy**

Outline

- **RAMCloud Background**
- **Motivation**
  - Goals & problems with current memory managers
- **Contributions**
  - Log-structured memory
  - Two-level Cleaning
  - Evaluation
  - Cost-Benefit Improvements
- **Conclusion**
  - Related Work
  - Future Work
  - Acknowledgements
  - Summary

LSM Related Work

- **Log-structured File Systems**
  - Log, segments, cleaning, cost-benefit technique
  - Applied DB and GC techniques to file systems
  - Zebra extended LFS techniques to clusters of servers

- **Garbage Collectors**
  - Cleaner ≈ generational copying garbage collector
  - Bump-a-pointer allocation
  - Much more difficult / general problem to solve

RAMCloud Related Work

- **In-Memory Databases**
  - "Large" datasets entirely in DRAM since early 80s
- **"NoSQL" Storage Systems**
  - Sacrifice data models for other features
    - Performance, scalability, durability, etc.
- **Low-latency Distributed Systems**
  - Supercomputing interconnects $\rightarrow$ commodity distributed systems since early 90s
    - Active Messages, U-Net, etc.

Future Work

- **Analysis of production RAMCloud workloads**
  - What are object size & access distributions like?
  - What read:write ratio do we need to support?

- **Optimisations**
  - Can we tighten up fast paths in the cleaner?
  - Are there better balancers for two-level cleaning?
  - Decouple in-memory and on-disk structures?
  - Scaling write throughput (multiple logs, or log heads?)
  - What is causing tail latency (with & without cleaning)?
  - Hole-filling techniques from Adaptive Methods LFS work?

Conclusion

- **Log-structured Memory for DRAM-based storage**
  - High memory utilisation: 80 - 90%
  - High performance: 410k small writes/sec at 90%
  - Durability: Can survive crashes, power failures
  - Adaptability: Handles workload changes
- **Two-level cleaning allows same DRAM and disk format**
  - Reduces I/O overheads (up to 87x)
  - Higher memory efficiency (cheap to track disk log)
- **Useful technique for other DRAM-based stores**

Acknowledgements

- **John, David, Mendel, Christos, Leonard**
- **RAMClouders:**
  - Gen 1: Ryan, Diego, Aravind
    (Cup Half Empty)
  - Gen 1.5: Ankita, Nandu, Elliot, Ali, Alex, Asaf, Christian, Stephen, Satoshi, Arjun
  - Gen 2: Jonathan, Behnam, Henry, Ashish, **Collin**, Adam

(You asked for it)

- **Aston**
- **Mom, Dad**

Summary

- **Contributions**
  - Log-structured memory
    - High performance, high memory utilisation, durability
  - Two-level cleaning
    - Optimizing the utilisation/write-cost trade-off
  - Cost-Benefit Improvements
    - Improved heuristic for selecting segments to clean
  - Parallel cleaning
    - Concurrent cleaning & writing, overheads off critical path
  - Cleaner balancing
    - Choosing how much of each cleaner to run