Carbink: Fault-Tolerant Far Memory

Yang Zhou\textsuperscript{1*} Hassan M. G. Wassel\textsuperscript{2} Sihang Liu\textsuperscript{3*} Jiaqi Gao\textsuperscript{1} James Mickens\textsuperscript{1} Minlan Yu\textsuperscript{1,2} Chris Kennelly\textsuperscript{2} Paul Turner\textsuperscript{2} David E. Culler\textsuperscript{2} Henry M. Levy\textsuperscript{2,4} Amin Vahdat\textsuperscript{2}

\textsuperscript{1}Harvard University \hspace{0.5cm} \textsuperscript{2}Google  
\textsuperscript{3}University of Virginia \hspace{0.5cm} \textsuperscript{4}University of Washington

* Contributed to this work during internships at Google.
Memory-Intensive Applications in Data Centers

Graph processing: # of nodes [1]

<table>
<thead>
<tr>
<th>Orkut 3M</th>
<th>Friendster 65M</th>
<th>Clueweb 955M</th>
<th>Videos 92B</th>
<th>Webpages 854B</th>
<th>~TB</th>
</tr>
</thead>
</table>

Memory provisioning is hard, as memory is limited by server physical boundary

- Over-provisioning memory for peak usage $\rightarrow$ 40%-60% memory utilization [3]
- Growing data in one process even exceeds single-server memory limit

Can applications dynamically utilize the unused memory on other servers?

Background: Far Memory on Commodity Servers [1,2,3,...]

Benefits of far memory:
- Dynamically provisioning unused memory to memory-intensive apps
- Apps can use much more memory than single-machine limit

Data center network (100Gbps, few-μs RTT)

Background: Far Memory on Commodity Servers [1,2,3,...]

...
Application Interface: Remotable Pointers

What if remote nodes fail?

Application-Integrated Far Memory [1]

The Must-Have Feature: Fault Tolerance

Probability of application crash grows almost linearly with N

How to build a fault-tolerant far memory system?
... assume fail-stop faults and no partial network failures
Talk Outline

Direction: in-memory erasure coding for fault tolerance

Carbink: making erasure coding work in practice

Evaluation: performance and cost of Carbink
Replication vs. Erasure Coding

**Replication**
- High memory overheads (3x)

**Erasure coding (EC)**
- Much smaller memory usage (1.5x)
- Single core achieves 4GB/s encoding tput [1]

SSD vs. Memory

SSD would become bottleneck during bursty workloads or failure recovery [1]

Talk Outline

Direction: in-memory erasure coding for fault tolerance
  ● High performance & low memory usage

Carbink: making erasure coding work in practice

Evaluation: performance and cost of Carbink
Challenge 1: Removable Objects Have Different Sizes

Erasure coding irregular-sized objects is hard

Carbink approach: grouping similar-sized objects into spans (like TCMalloc [1])

- Spans are page-aligned and regular-sized

Grouping Similar-Sized Objects into Spans

Span-centric memory pooling

- Applying spans to object management and data swapping
- Spans are page-aligned, and never end with a partial object
Challenge 2: Efficient Swapping under Erasure Coding

EC-Split (Hydra [1]):
erasure codes individual spans

Multiple network IOs to swap-in/out a span
● Stressing network stack → slow swapping
● Stragglers → high tail latency

EC-Batch (Carbink):
erasure codes spansets

Single network IO to swap-in a span
● Fast swapping and low tail latency

Swap-In&Out Granularity Mismatch $\rightarrow$ Remote Fragmentation
Swap-In&Out Granularity Mismatch → Remote Fragmentation
Remote Compaction for Defragmentation

No impacts on span swapping perf: off the critical path of swap-ins/outs

Penalty: may consume more memory; dead spans not compacted immediately

Zero-copy span merging
Talk Outline

Direction: in-memory erasure coding for fault tolerance
- High performance & low memory usage

Carbink: making erasure coding work in practice
- Span-centric memory pooling → managing arbitrary-sized objects
- Erasure coding spansets → achieving swapping efficiency

Evaluation: performance and cost of Carbink
Evaluation Overview

Workloads:
- An internal transactional KV-store doing TPC-A transactions
- Graph connected components (skipped here due to time limit)
- A microbenchmark dereferencing remotable objects

Metrics: throughput, tail latency, memory usage

Testbed:
- Servers with 50 Gbps NIC and PonyExpress [1] user-space network stacks
- One-sided RMAs for span swapping; RPCs for remote compaction

Throughput (KV-store)

Perform similarly because working set fits local memory

EC-Batch achieves up to 1.5x speedup over EC-Split
Tail Latency (Microbenchmark)

EC-Batch achieves 1.2x-1.4x tail latency reduction over EC-Split (before knee point)
Other Results

Remote memory usage:

● EC-Batch consumes at most 35% more memory than EC-Split
● ... but still only $\frac{2}{3}$ of replication memory usage

More in the paper!

● Remote compaction resource usage
● Failure recovery times
● AIFM (swapping individual objects) vs. Carbink
Carbink Summary

Fault tolerance is a must-have feature for applications to use far memory

**Carbink**: making erasure coding FT work in practice for far memory system
- Grouping objects into spans → handle arbitrary-sized objects
- Erasure coding spansets → single network IO data-fetch

Up to 1.5x application speedup and 1.4x tail latency reduction with up to 35% more memory usage (compared to state-of-the-art EC-Split)
Thank You!

**Carbink**: making erasure coding FT work in practice for far memory system

Up to 1.5x application speedup and 1.4x tail latency reduction with up to 35% more memory usage (compared to state-of-the-art EC-Split)