Carbink: Fault-Tolerant Far Memory

Yang Zhou[†*] Hassan M. G. Wassel[‡] Sihang Liu[§*] Jiaqi Gao[†] James Mickens[†] Minlan Yu[†‡]
Chris Kennelly[‡] Paul Turner[‡] David E. Culler[‡] Henry M. Levy[‖‡] Amin Vahdat[‡]
[†]*Harvard University* [‡]*Google* [§]*University of Virginia* [‖]*University of Washington*

1 Motivation

In a datacenter, matching a particular application to just enough memory and CPUs is hard. A commodity server tightly couples memory and compute, hosting a fixed number of CPUs and RAM modules that are unlikely to exactly match the computational requirements of any particular application. Even if a datacenter contains a heterogeneous mix of server configurations, the load on each server (and thus the amount of available resources for a new application) changes dynamically as old applications exit and new applications arrive. Thus, even state-of-the-art cluster schedulers [25, 26] struggle to efficiently bin-pack a datacenter's aggregate collection of CPUs and RAM. For example, Google [26] and Alibaba [18] report that the average server has only ~60% memory utilization, with substantial variance across machines.

Disaggregated datacenter memory [3, 5, 7, 8, 11, 23, 24] is a promising solution. In this approach, a CPU can be paired with an arbitrary set of possibly-remote RAM modules, with a fast network interconnect keeping access latencies to far memory small. From a developer's perspective, far memory can be exposed to applications in several ways. For example, an OS can treat far RAM as a swap device, transparently exchanging pages between local RAM and far RAM [5,11,24]. Alternatively, an application-level runtime like AIFM [23] can expose remotable pointer abstractions to developers, such that pointer dereferences (or the runtime's detection of high memory pressure) trigger swaps into and out of far memory.

Much of the prior work on disaggregated memory [3,23,28] has a common limitation: a lack of fault tolerance. Unfortunately, in a datacenter containing hundreds of thousands of machines, faults are pervasive. Many of these faults are planned, like the distribution of kernel upgrades that require server reboots, or the intentional termination of a low-priority task when a higher-priority task arrives. However, many server faults are unpredictable, like those caused by hardware failures or kernel panics. Thus, any *practical* system for far memory has to provide a scalable, fast mechanism to recover from unexpected server failures. Otherwise, the failure rate of an application using far memory will be much higher than the failure rate of an application that only uses local memory; the reason is that the use of far memory increases the set of machines whose failure can impact an application [6].

Some prior far-memory systems do provide fault tolerance via replication [5, 11, 24]. However, replication-based approaches suffer from high storage overheads. Hydra [15] uses erasure coding, which has smaller storage penalties than replication. However, Hydra's coding scheme stripes a single memory page across multiple remote nodes. This means that a compute node requires multiple network fetches to reconstruct a page; furthermore, computation over that page cannot be outsourced to remote memory nodes, since each node contains only a subset of the page's bytes.

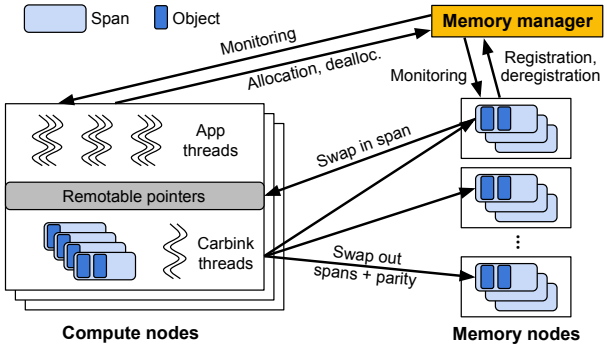

Figure 1: Carbink's high-level architecture.

In this paper, we present Carbink,[1] a new framework for far memory that provides efficient, high-performance fault tolerance via span-based memory management and erasure coding. Unlike Hydra, Carbink also allows computation to be offloaded to remote memory nodes. We have implemented Carbink atop our datacenter infrastructure. Compared to Hydra, Carbink has up to 29% lower tail latency and 48% higher application performance, with at most 35% more remote memory usage.

2 Carbink Design

Figure 1 depicts the high-level architecture of Carbink. **Compute nodes** execute single-process (but potentially multi-threaded) applications that want to use far memory. **Memory nodes** provide far memory that compute nodes use to store application data that cannot fit in local RAM. A logically-centralized **memory manager** tracks the liveness of compute nodes and memory nodes. The manager also coordinates the assignment of far memory **regions** to compute nodes. When a memory node wants to make a local memory region available to compute nodes, the memory node *registers* the region with the memory manager. Later, when a compute node requires far memory, the compute node sends an *allocation* request to the memory manager, who then assigns a registered, unallocated region. Upon receiving a *deallocation* message from a compute node, the memory manager marks the associated region as available for use by other compute nodes. A memory

[*]Contributed to this work during internships at Google.

[1]Carbink is a Pokémon that has a high defense score.

node can ask the memory manager to *deregister* a previously registered (but currently unallocated) region, withdrawing the region from the global pool of far memory.

Carbink does not require participating machines to use custom hardware. For example, any machine in a datacenter can be a memory node if that machine runs the Carbink memory host daemon. Similarly, any machine can be a compute node if that node's applications use the Carbink runtime.

From the perspective of an application developer, the Carbink runtime allows a program to dynamically allocate and deallocate memory objects of arbitrary size. Programs access those objects through AIFM-like remotable pointers [23]. When applications dereference pointers that refer to non-local (i.e., swapped-out) objects, Carbink pulls the desired objects from far memory. Under the hood, Carbink's runtime manages objects using **spans** and **spansets**. A span is a contiguous run of memory pages; a single region allocated by a compute node contains one or more spans. Similar to slab allocators like Facebook's jemalloc [9] and Google's TCMalloc [10, 12], Carbink rounds up each object allocation to the bin size of the relevant span, and aligns each span to the page size used by compute nodes and memory nodes. Carbink swaps far memory into local memory at the granularity of a span; however, when local memory pressure is high, Carbink swaps local memory out to far memory at the granularity of a spanset (i.e., a collection of spans of the same size). In preparation for swap-outs, background threads on compute nodes group cold objects into cold spans, and bundle a group of cold spans into a spanset; at eviction time, the threads generate erasure-coding parity data for the spanset, and then evict the spanset and the parity data to remote nodes. We term this approach **EC-Batch**, as spans are evicted in batches.

In Carbink, each span lives in exactly one place: the local RAM of a compute node, or the far RAM of a memory node. Thus, swapping a span from far RAM to local RAM creates dead space (and thus fragmentation) in far RAM. Carbink runs pauseless defragmentation threads in the background, asynchronously reclaiming space to use for later swap-outs.

Carbink disallows cross-application memory sharing. This approach is a natural fit for our target applications, and has the advantage of simplifying failure recovery and avoiding the need for expensive coherence traffic [24].

3 Evaluation

Microbenchmarks: To get a preliminary idea of Carbink's performance, we created a synthetic benchmark that wrote 15 million 1 KB objects (totalling 15 GB) to a remotable array. The compute node's local memory had space to store 7.5 GB of objects (i.e., half of the total set). The compute node spawned 128 threads on 32 logical cores to access objects; the access pattern had a Zipfian-distributed [22] skew of 0.99.

Figure 2 shows the 99th-percentile latency with various object access loads. Both of the fault-tolerant schemes eventually hit a "hockey stick" in tail latency growth when the

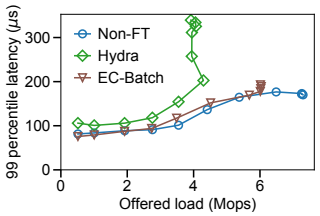

Figure 2: Microbenchmark load-latency curves.

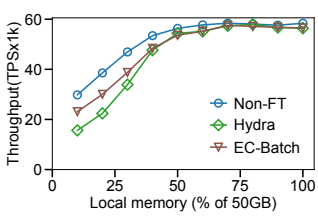

Figure 3: KV-store transaction throughput.

schemes could no longer catch up with the offered load. EC-Batch had the highest sustained throughput (6.0 Mops), which was 40% higher than the throughput of the state-of-the-art Hydra (4.3 Mops). Hydra had worse performance because it had to issue four RMA requests to swap in one span; thus, Hydra quickly became bottlenecked by network IO. In contrast, EC-Batch only issued one RMA request per swap-in. EC-Batch had 18%-29% lower tail latency than Hydra under the same load (before reaching the "hockey-stick"). The reason was that Hydra's larger number of RMAs per swap-in left Hydra more vulnerable to stragglers [15]. Also recall that EC-Batch can support computation offloading [4, 14, 23, 30], which is hard with Hydra (§1).

Macrobenchmarks: We further evaluated Carbink using an in-memory transactional key-value store that would benefit from remote memory. This application implemented a transactional in-memory B-tree, exposing it via a key/value interface similar to that of MongoDB [20]. Each remotable object was a 4 KB value stored in a B-tree leaf. The application spawned 128 threads, and each thread processed 20 K transactions. The compute node provisioned 32 logical cores, with the application overlapping execution of the threads for higher throughput [13, 21, 23, 29]. Each transaction contained three reads and three writes, similar to the TPC-A benchmark [27]. Each update created a new version of a particular key's value; asynchronously, the application trimmed old versions.

Throughput: Figure 3 shows the KV-store throughput when varying the size of local memory (normalized as a fraction of the maximum working set size). In scenarios with less than 50% local memory, EC-Batch achieved higher transactions per second (TPS) than Hydra. For example, TPS for EC-Batch was 1.5%-48% higher than that of Hydra; this was because EC-Batch only needed one RMA request to swap in a span. EC-Batch was at most 29% slower than Non-FT, mainly due to the additional parity update required for fault tolerance.

Remote memory usage: Compared to Hydra, EC-Batch used up to 35% more remote memory. EC-Batch defragmented remote memory using compaction, but when local memory space was less than 50%, remote compaction could not immediately defragment the spanset holes created by frequent span swap-ins. As local memory grew larger, span fetching became less frequent, making it easier for remote compaction to reclaim space. In this less hectic environment, EC-Batch's remote memory usage was similar to Hydra.

4 Discussions

Memory sharing across processes/applications: Carbink relies on locks to synchronize remotable object accesses among different application threads and background swap-out threads in the same process. To extend Carbink cross processes/applications, one possible approach is to rely on shared memory across different processes to implement such synchronization. Another possible approach is to rely on a separate process that handles and synchronizes object accesses from all processes in one compute node. This approach is similar to the microkernel approaches to host networking [19] and filesystem [17].

Emerging hardware for disaggregated memory: Emerging high-performance compute-memory interconnects like CXL [2] and CCIX [1] support pooled memory shared by multiple CPU sockets and hardware-based cache coherence between CPU memory and device memory. These interconnects have enabled small-scale (i.e., 8-16 CPU sockets) memory disaggregation with nearly no performance loss [16]. In this context, Carbink's span-based erasure coding approach can be applied to make the pooled memory tolerate individual DIMM failure.

Scalability of the memory manager: Carbink memory manager tracks the liveness of compute nodes and memory nodes, and handles memory registration/deregistration and allocation/deallocation. We expect the memory manager to provide second-level liveness monitoring, while individual compute nodes may implement millisecond- or microsecond-level monitoring for the memory nodes they use. Thus the liveness monitoring on the memory manager will less likely become a bottleneck. Besides, the memory manager registers/deregisters and allocates/deallocates far memory in the unit of GB-level regions; this avoids frequent involvements of the memory manager and the manager only needs to maintain small in-memory states for these region assignments.

References

[1] CCIX Consortium. https://www.ccixconsortium.com/.

[2] Compute Express Link (CXL). https://www.computeexpresslink.org/.

[3] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, and et al. Remote Regions: A Simple Abstraction for Remote Memory. In *Proceedings of USENIX ATC*, pages 775–787, 2018.

[4] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of ACM HotOS*, pages 120–126, 2019.

[5] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of ACM EuroSys*, pages 1–16, 2020.

[6] Cristina Băsescu and Bryan Ford. Immunizing Systems from Distant Failures by Limiting Lamport Exposure. In *Proceedings of ACM HotNets*, pages 199–205, 2021.

[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of USENIX NSDI*, pages 401–414, 2014.

[8] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[9] Jason Evans. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of BSDCan Conference*, 2006.

[10] Google. TCMalloc Open Source. https://github.com/google/tcmalloc.

[11] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with INFINISWAP. In *Proceedings of USENIX NSDI*, pages 649–667, 2017.

[12] Andrew Hamilton Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond Malloc Efficiency to Fleet Efficiency: A Hugepage-Aware Memory Allocator. In *Proceedings of USENIX OSDI*, pages 257–273, 2021.

[13] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[14] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Proceedings of Conference on Innovative Data Systems Research*, 2022.

[15] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. *arXiv preprint arXiv:1910.09727*, 2019.

[16] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. 2023.

[17] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of ACM SOSP*, pages 819–835, 2021.

[18] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Proceedings of IEEE International Conference on Big Data*, pages 2884–2892, 2017.

[19] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[20] MongoDB Inc. MongoDB Open Source. https://github.com/mongodb/mongo.

[21] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[22] David M.W. Powers. Applications and Explanations of Zipf's Law. In *Proceedings of New Methods in Language Processing and Computational Natural Language Learning*, 1998.

[23] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of USENIX OSDI*, pages 315–332, 2020.

[24] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of USENIX OSDI*, pages 69–87, 2018.

[25] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, and et al. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of USENIX OSDI*, pages 787–803, 2020.

[26] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of ACM EuroSys*, pages 1–14, 2020.

[27] Transaction Processing Performance Council (TPC). TPC-A. http://tpc.org/tpca/default5.asp.

[28] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated Managed Runtime. In *Proceedings of USENIX OSDI*, pages 261–280, 2020.

[29] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid Is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[30] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship Compute or Ship Data? Why Not Both? In *Proceedings of USENIX NSDI*, pages 633–651, 2021.