DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory

Sekwon Lee*, Soujanya Ponnapalli*, Sharad Singhal‡,
Marcos K. Aguilera◇, Kimberly Keeton†, Vijay Chidambaram*◇
†Google, ‡Hewlett Packard Labs, *University of Texas at Austin, ◇VMware Research

1 INTRODUCTION

Large cloud providers operate at a much larger scale than traditional enterprise data centers and aim to optimize their infrastructures for high utilization. However, recent work indicates that resources in cloud data centers remain underutilized [12, 16].

One promising way to increase resource utilization is to disaggregate resources [2, 5, 9, 11]. In a disaggregated cluster, resources such as CPU, memory, and storage are each collected into a separate central network-attached pool. By sharing these resources across users and applications, utilization can be increased significantly. Furthermore, each resource can be scaled up or down independently of the others. Such disaggregation has long been practiced for storage in the form of network-attached storage (NAS) [6] and Storage Area Networks (SAN) [3].

Persistent Memory (PM) is a new memory technology that provides durability like traditional storage, with performance close to DRAM [7, 14, 20]. Since PM has much higher cost per GB than conventional storage [1], it is critical to achieve high utilization in PM deployments. Similar to traditional storage, the utilization of PM would increase from disaggregation. However, the DRAM-like latencies of PM make disaggregation challenging, since the network latency is an order of magnitude higher than PM latency.

We are interested in using Disaggregated Persistent Memory (DPM) to build persistent key-value stores (KVSs), which are critical pieces of software infrastructure. The KVS consists of a number of KVS nodes (KNs) equipped with general-purpose processors, a relatively small amount of local DRAM, and high-performance network primitives like RDMA to access DPM over the network [18]. An ideal DPM KVS would have a number of properties: high common-case performance, scalability, and quick reconfiguration that allows handling failures, bursty workloads, and load imbalance efficiently.

Building a KVS that achieves all the goals simultaneously is challenging. First, the KVS must provide high performance, despite expensive network round trips (RTs) for accessing data and metadata in DPM. Second, to benefit from independent scaling of KNs and PM, the KVS must be elastic and support lightweight reconfiguration of resources. Finally, the KVS must provide scalable performance with an increase in active resources (*e.g.,* KNs).

Prior DPM KVSs make design trade-offs that make these goals difficult to satisfy simultaneously. For example, AsymNVM [13] achieves high performance by adopting a shared-nothing architecture to enable high cache locality at KNs. However, expensive data reorganization is needed when changing the number of KNs, thus limiting elasticity. Clover [17] supports high elasticity using a shared-everything architecture where data is shared across KNs, and any KN can handle any request. However, performance and scalability suffer as a result of poor cache locality and consistency overheads (*e.g.,* cache coherence, contention, and synchronization overheads due to sharing) in the common case [15].

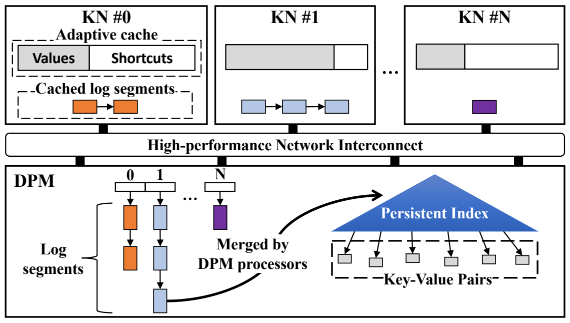

Figure 1: DINOMO data plane

In this work, we present **DINOMO**, the first DPM KVS that simultaneously achieves high common-case performance, scalability, and lightweight online reconfiguration. DINOMO also provides linearizable reads and writes. To achieve these goals, DINOMO carefully adapts techniques from the storage research community, including caching, ownership partitioning, and lock/log-free PM indexing.

Data organization on DPM. Figure 1 shows the data plane components in DINOMO. DINOMO stores data and metadata on DPM to enable concurrent and consistent access by all KNs. Because DPM is shared among all KNs, it functions as the source of ground truth in the system. To enable consistent updates, data is written into the log segments in DPM in the form of log entries by the KNs. These log entries are asynchronously merged in order into the metadata index by the processors at DPM. For its metadata index, DPM uses a concurrent PM index [10] that provides lock-free reads and log-free in-place-writes; the lock-free reads eliminate synchronization overheads between KNs and log-free in-place-writes allow DPM processors to concurrently update the metadata.

Disaggregated Adaptive Caching (DAC). Similar to other disaggregated systems, DINOMO reduces network RTs by caching data and metadata in the local DRAM of each KN, as shown in Figure 1. Data is cached by storing the key-value pair, and metadata is cached by storing a pointer to the data on DPM (termed *shortcuts* [17]). To determine how best to divide the cache space between data and metadata, DINOMO uses DAC, a novel adaptive caching policy that actively maintains the right balance between caching values and shortcuts based on the workload patterns and available memory at KNs. DAC is based on the following insight. Performance is highly correlated with the number of network RTs, so we seek to minimize that. Caching a shortcut reduces RTs from M (where M is the cache-miss cost of an index lookup) to one, while caching a value instead of a shortcut reduces RTs from one to zero. Thus,

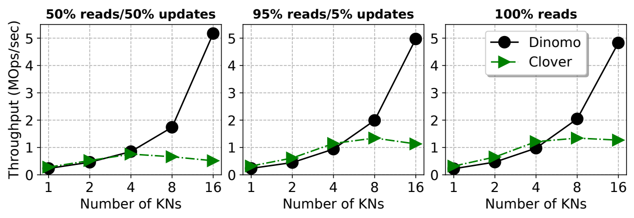

Figure 2: Performance scalability comparison with Clover

caching shortcuts provides the bigger gain. We treat value caching as an optimization on top of shortcut caching. Value caching is used when we have spare space in the cache, or when we observe that storing a value can serve more requests than storing an equivalent number of shortcuts. With DAC, Dinomo makes efficient use of the memory at KNs without any assumptions about the workload.

Ownership Partitioning (OP). While caching at the KNs can reduce network RTs, it can incur significant consistency overheads when KNs can share the same data. To handle this concern, Dinomo partitions the *ownership* of data across KNs (*i.e.,* a KV pair may only be read or written by its owner), while data and metadata are shared via DPM. This provides three benefits. First, it allows KNs to cache the data they own, thus providing high cache locality without consistency overheads. Second, by sharing the data and metadata, OP supports changing the number of KNs or rebalancing their load by repartitioning only the ownership of data among KNs, without expensive data reorganization at DPM. Finally, since each key is only accessed by one KN at any given point, Dinomo achieves linearizable reads and writes. With OP, Dinomo achieves high performance and scalability from locality-preserving KN-side caching without consistency overheads, and high elasticity from lightweight reconfiguration.

Limitations. Our work has a number of limitations. First, while we address the challenge of scaling KNs, we do not tackle how to make DPM reliable or scalable. Second, Dinomo targets key-value store functionality for DPM systems. Many of its ideas may be equally applicable for a broader range of DPM-based storage systems as well as disaggregated DRAM systems, but we have not explored this. We consider these areas ripe for future work.

2 EVALUATION

We now evaluate the performance and scalability of Dinomo. As our baseline, we use Clover [17], a state-of-the-art and open-source DPM KVS. Clover has a shared-everything architecture and caches only shortcuts at its KNs. Besides Clover, we implement Dinomo-N to compare the elasticity of Dinomo with a shared-nothing counterpart; it uses DAC but partitions data and metadata in DPM, where each partition is exclusively accessed by a single KN.

Experiment setup. We use InfiniBand-enabled (IB-enabled) servers as hosts for KNs and DPM; each two-socket server has Intel Xeon E5-2670v3 processors, 24 cores at 2.30 GHz in total, and 128 GB DRAM. The shared DPM uses a maximum of 4 threads and 110 GB of DRAM as a proxy for the PM, which is registered to be RDMA-accessible. Each KN uses a maximum of 8 threads and 1 GB of DRAM for caching (≈1% of the DPM size). DPM and the KNs are connected by Mellanox FDR ConnectX-3 adapters with 56 Gbps per

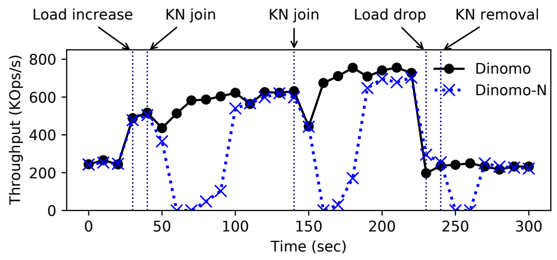

Figure 3: Throughput of Dinomo and Dinomo-N over time while changing the load and number of KNs

port. We emulate PM using DRAM, as performance is constrained by the network rather than PM or DRAM [1, 8]. For Clover, we use an extra IB-enabled server for its metadata server with 6 threads (4 workers, 1 epoch thread, 1 GC thread).

Workloads and configurations. We use YCSB-style workloads [4, 19] with three request patterns: read-only (100% reads), read-mostly (95% reads/5% updates), and write-heavy (50% reads/50% updates). These workloads use 8B keys and 1KB values. For each experiment, we first load 32 GB of data (key-value pairs) and then write up to 100GB of data during the workload including inserts. With 16 KNs, each equipped with a 1GB cache, the KNs can cache up to 50% of the loaded dataset. We generate the workload from separate client nodes with 64 threads respectively.

Performance and scalability. We compare the end-to-end performance and scalability of Dinomo and Clover. We use workloads with moderate skew (Zipf 0.99) to observe the performance and scalability in the common case. We use 8 client nodes to run these workloads and measure the peak throughput. As shown in Figure 2, Dinomo's throughput scales to 16 KNs. In contrast, Clover's throughput does not scale beyond 4 KNs due to either a network bottleneck or the CPU bottleneck from its metadata server. With 16 KNs, Dinomo outperforms Clover by at least 3.8× across all workloads. Dinomo has a higher cache hit rate (from values) with more KNs and takes fewer RTs/op than Clover, owing to DAC and OP. Clover has higher network costs due to shortcut-only caching and a lack of locality caused by the shared-everything architecture that results in consistency overheads and redundant caching.

Elasticity. We evaluate Dinomo with bursty, irregular workloads and compare its elasticity in scaling KNs with Dinomo-N. We use the write-heavy workload with low skew (Zipf 0.5). Figure 3 shows the behavior of Dinomo and Dinomo-N during this experiment. To produce the bursty, irregular workload, we start running the workload using 1 client node for 20 seconds, then increase the load on the systems by 7× by adding 7 additional client nodes. At the 230-second mark, we remove 7 client nodes to lower the load by 7× again. When the load increases at 30 seconds, Dinomo and Dinomo-N react by adding new KNs at 40 and 140 seconds, respectively, to spread the load. Once the new KNs come online, Dinomo shows brief throughput dips, as the nodes update their hash rings. However, Dinomo-N experiences throughput dips for 30-40 seconds, where the throughput drops to 0 due to the processing delay during data reorganization. At 230 seconds, the load is suddenly reduced. While removing the under-utilized KN, Dinomo-N shows a 20-second throughput dip before stabilizing.

REFERENCES

[1] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2020. Assise: Performance and Availability via Client-Local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 1011–1027.

[2] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the Application. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing*. Article 15.

[3] Richard Barker and Paul Massiglia. 2001. *Storage Area Network Essentials: A Complete Guide to Understanding and Implementing SANs* (1st ed.). Wiley Publishing.

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154.

[5] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. 249–264.

[6] Garth A. Gibson and Rodney Van Meter. 2000. Network Attached Storage Architecture. *Commun. ACM* 43, 11 (nov 2000), 37–45.

[7] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. https://doi.org/10.48550/ARXIV.1903.05714 Accessed: 2022-09-19.

[8] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 105–119.

[9] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-Centric Computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. Article 1, 1 pages.

[10] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 462–477.

[11] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. 267–278.

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

[13] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 757–773.

[14] Intel Optane DC Persistent Memory. 2022. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed: 2022-02-16.

[15] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. 2012. OLTP on Hardware Islands. *Proc. VLDB Endow.* 5, 11 (jul 2012), 1447–1458.

[16] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. Article 30, 14 pages.

[17] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 33–48.

[18] Haris Volos, Kimberly Keeton, Yupu Zhang, Milind Chabbi, Se Kwon Lee, Mark Lillibridge, Yuvraj Patel, and Wei Zhang. 2018. Memory-Oriented Distributed Computing at Rack Scale. In *Proceedings of the ACM Symposium on Cloud Computing*. 529.

[19] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2019. Autoscaling Tiered Cloud Storage in Anna. *Proc. VLDB Endow.* 12, 6 (feb 2019), 624–638.

[20] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies*. 169–182.