We present Dinomo, a novel key-value store for disaggregated persistent memory (DPM). Dinomo is the first key-value store for DPM that simultaneously achieves high common-case performance, scalability, and lightweight online reconfiguration. We observe that previously proposed key-value stores for DPM had architectural limitations that prevent them from achieving all three goals simultaneously. Dinomo uses a novel combination of techniques such as ownership partitioning, disaggregated adaptive caching, selective replication, and lock-free and log-free indexing to achieve these goals. Compared to a state-of-the-art DPM key-value store, Dinomo achieves at least 3.8x better throughput at scale on various workloads and higher scalability, while providing fast reconfiguration.

PVLDB Reference Format:
Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory. PVLDB, 15(13): 4023 - 4037, 2022.
doi:10.14778/3565838.3565854

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/utsaslab/dinomo.

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 [30, 52, 69, 74]. In the face of dynamic and bursty workloads, scheduling tasks such that resource utilization is high proves challenging [90]. For example, memory utilization can be as low as 60% [15, 74, 76].

One promising way to increase resource utilization is to disaggregate resources [4, 24, 36, 49]. 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: for example, memory can be added without the need to also add CPU or storage. Such disaggregation has long been practiced for storage in the form of network-attached storage (NAS) [25] and Storage Area Networks (SAN) [7]. In this work, we take the idea one step further and consider a cluster where Persistent Memory is disaggregated.

Persistent Memory (PM) is a new memory technology that provides durability like traditional storage, with performance close to DRAM [32, 56, 86]. Since PM has much higher cost per GB than conventional storage [3], 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.

Disaggregated Persistent Memory (DPM) is still under active research and development, and hence there are different kinds of DPM to build upon. In this work, we assume that DPM is available as a centralized, reliable pool accessible via the network [37]. We further assume that DPM includes some limited computational capability, as prior work shows such capability is critical for achieving good performance [54, 75, 89].

We are interested in using 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 [79]. An ideal KVS for DPM 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, KNs incur expensive network round trips (RTs) for accessing data and metadata in DPM. Despite these overheads, the KVS must provide high performance. 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 without bottlenecks due to load imbalance at KNs or from non-uniform workload patterns. Prior DPM KVSs [54, 75] make design trade-offs that make these goals difficult to satisfy simultaneously. For example, Asymmetric Virtualization (AVM) [54] 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 or rebalancing their load, thus limiting elasticity and efficient load balancing. Similarly, Clover [75] supports straightforward load balancing and 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 (including cache coherence, contention, and synchronization overheads due to sharing) in the common case [65].

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, selective replication, and lock-free and log-free PM indexing.

**Data organization on DPM (§3.2).** 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 to 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 [46] which provides lock-free reads and log-free in-place-writes; the lock-free reads allow us to eliminate synchronization overheads between KNs and log-free in-place-writes allow DPM processors to concurrently update the metadata.

**Disaggregated Adaptive Caching (DAC) (§3.3).** Similar to other disaggregated systems, Dinomo reduces network RTs by caching data and metadata in the local DRAM of each KN. Data is cached by storing the key-value pair, and metadata is cached by storing a pointer to the data on DPM (termed *shortcuts* [75]). 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 allows Dinomo to make efficient use of the DRAM at KNs without making any assumptions about the workload.

**Ownership Partitioning (OP) (§3.4).** 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, 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, combined with our principled reconfiguration protocol, Dinomo achieves linearizable reads and writes. Similar ideas have been proposed before in other contexts [1, 7, 13, 80], but we are the first to adapt it for DPM. With OP, Dinomo achieves high performance/scalability from locality-preserving KN-side caching without consistency overheads and high elasticity from lightweight reconfiguration.

**Selective Replication (§3.4).** Ownership partitioning, however, may experience performance or scalability bottlenecks at KNs due to load imbalance under highly skewed workloads (i.e., the maximum throughput for requests on a single key is limited by the processing capacity of a single KN). To avoid this issue and provide scalable performance for highly skewed workloads, Dinomo selectively replicates the ownership of hot keys across multiple KNs. Dinomo has a separate monitoring/management node that identifies hot keys, initiates their ownership replication to other KNs, and thus balances the load from hot keys across available KNs.

**Alleviate network and CPU bottlenecks (§3.6).** Dinomo’s data path uses one-sided RDMA operations with asynchronous post processing. All reads to DPM by KNs use one-sided RDMA operations on a shortcut hit or a cache miss. Dinomo writes multiple log entries in a batch in the critical path using a one-sided RDMA operation, and delegates the merging of the writes into the metadata index to the DPM processors asynchronously. Asynchronous post-processing reduces write latency and amortizes DPM processing utilization across multiple writes, reducing how much DPM computing power is needed in the critical path. These optimizations decrease the network messages per operation and alleviate the processing bottleneck from DPM, increasing the efficiency of Dinomo in addition to techniques like DAC and OP.

**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. Finally, while our work provides mechanisms for scaling KNs, it does not tackle the policies for when KNs should be scaled. We consider these areas ripe for future work.

**Evaluation.** We implement Dinomo in 10K lines of C++ code. We compare the end-to-end performance and scalability of Dinomo with Clover [75], a state-of-the-art DPM KVS. Our experiments show that Dinomo achieves both better common-case performance and scalability than Clover. Dinomo’s throughput scales to 16 KNs, while Clover’s throughput does not scale beyond 4 KNs. With 16 KNs, Dinomo outperforms Clover by at least 3.8× on all workloads we evaluate. We also show that Dinomo elastically scales out KNs, balances the load across KNs, and handles KN failures quickly.

In summary, this paper makes the following contributions:

- We present Dinomo, the first DPM key-value store that simultaneously achieves high performance, scalability, and lightweight online reconfiguration (§3).
- We present DAC, a novel adaptive caching policy that helps utilize the KN-side memory effectively without any assumptions on workload patterns (§3.3).
- We adapt OP for DPM KVSs to achieve high performance, scalability, and lightweight reconfiguration (§3.4).
- We experimentally show that Dinomo can efficiently react to both KN failures and load imbalance, and automatically scale the number of KNs by capturing load dynamics (§5).
2 BACKGROUND AND MOTIVATION

We describe persistent memory (PM) and how it can be used in disaggregated settings. We then discuss prior key-value stores (KVSs) for disaggregated PM (DPM) and motivate the need for a new KVS.

2.1 Persistent Memory and Disaggregation

PM is a non-volatile memory technology with unique characteristics [32, 86]. PM is connected directly to the memory bus – it is byte addressable, and has performance close to DRAM. It has high capacity: Intel’s Optane DC PM is available up to 512GiB per NVDIMM [56]. The per-GB cost of PM is higher than high-end solid state drives, but less than DRAM [3]. To improve cost efficiency and PM utilization, prior work proposes DPM [38, 51, 54, 75, 79, 89]. We note that our work is agnostic to the choice of PM technology and specific PM product (e.g., PCM [83], STT-MRAM [5], Memristor [87], Optane DC PM [56], Memory-Semantic CXL SSD [21]).

Disaggregated PM. In disaggregated settings, PM is available as a central, reliable pool of memory accessible over a network. KVS nodes (KNs) are used to access the data in DPM; KNs have limited DRAM and use network primitives like RDMA to access the PM pool over a fast interconnect such as InfiniBand [4], PMoF [26, 28], or Gen-Z [16]. Disaggregation allows independent scaling of PM and KNs and introduces separate failure domains, where KN failures do not cause DPM failures.

DPM can be classified as active or passive. Active DPM has small processing units such as ARM SOCs, ASICs, or FPGAs, with high-bandwidth network ports. In active DPM, DPM compute capacity is used for local processing, including network, application-level, and data store processing [39, 54, 70]. Prior work has proposed data stores for active DPM that leverage this limited computational power [29, 51, 54, 75, 89]. In contrast, passive DPM has no computational abilities at the DPM pool. KNs can use only one-sided RDMA operations to access and modify the data in DPM. Data stores for passive DPM [75] have poor performance and scalability due to the limited functionality of the one-sided network primitives [2], showing that active DPM is a more practical deployment.

2.2 DPM Key-Value Stores

Previously proposed DPM key-values stores differ based on how they handle data, metadata, and ownership. Metadata is information used to locate and access data (like an index). Ownership determines if a data item can be read or written.

AsymNVM. AsymNVM [54] adopts a shared-nothing architecture. Data in DPM is partitioned, and each partition is exclusively accessed by a single KN. Every KN uses its local memory to cache data from its partition (Table 1); caching helps reduce expensive network round trips to DPM. As KNs have exclusive ownership over data, their caches can preserve locality and can be consistent without incurring additional consistency overheads. Thus, shared-nothing architectures provide high performance and scalability in the common case by effectively using KN caches to process requests. However, reconfiguring the number of KNs or balancing load across KNs requires physical reorganization of data and metadata [9, 30, 41, 54]. For example, adding a new KN may require the metadata of a partition to be split, resulting in expensive data copies at DPM. Thus, AsymNVM offers performance at the expense of elasticity and fast reconfiguration.

Clover. Clover [75] adopts a shared-everything architecture. All KNs share the ownership of data in DPM, and every KN can access and modify all data and metadata (Table 1). KNs can use local memory to cache data. However, due to sharing, KNs have poor cache locality and need to keep their caches consistent, incurring significant consistency overheads that reduce the common-case performance and scalability [65]. Nevertheless, Clover can support lightweight reconfiguration without re-partitioning data or metadata and allow straightforward load balancing across KNs. Overall, Clover offers elasticity and lightweight reconfiguration at the expense of high common-case performance and scalability.

In summary, prior DPM key-value stores sacrifice one of high common-case performance, scalability, or lightweight reconfiguration for the other two (Table 1). This motivates our design for a new DPM key-value store, Dinomo, which achieves the three properties simultaneously.

3 DINOMO

We now present Dinomo, a key-value store (KVS) for DPM. We first describe its API, target workloads, goals, and the guarantees it provides. Then, we explain how Dinomo achieves its goals (Table 2).

API. Dinomo allows applications to perform insert(key, value), update(key, value), lookup(key), or delete(key) on variable-sized key-value pairs. We refer to the lookup operations as reads, and the insert, update, and delete operations as writes.

Target workloads. Dinomo targets applications with dynamic working sets and sizes, and non-uniform workloads with varying skew [60, 67, 84]. Large variations in workloads require DPM KVSs to allow the elastic deployment of resources (e.g., KNs) in response to those dynamics [12, 90].

Goals. Dinomo aims to achieve the following goals:

- High common-case performance, in the absence of failures or reconfiguration
- Scalability of performance when the number of KNs increases

| KVS property | DINOMO | Clover | AsymNVM |
|--------------|--------|--------|---------|
| Data         | shared | shared | partitioned |
| Metadata     | shared | shared | partitioned |
| Ownership of data | partitioned | shared | partitioned |
| High performance | ✓ | × | ✓ |
| Scalability   | ✓ | ✓ | ✓ |
| Lightweight reconfiguration | ✓ | ✓ | × |

Table 1: Design choices and properties of different DPM KVS

| Goals                          | DINOMO techniques |
|-------------------------------|-------------------|
| High performance              | Ownership partitioning, DAC |
| Lightweight reconfiguration and scalability | Ownership partitioning |

| Linearizable reads and writes | Shared DPM, Ownership partitioning |

Table 2: Dinomo goals and design techniques
- Lightweight online reconfiguration to effectively handle KN failures, bursty workloads, and load imbalance on available KNs
- Linearizable reads and writes

Guarantees. Dinomo guarantees that once committed, data will not be lost or corrupted regardless of KN failures. It also ensures data remains available if at least one KN and DPM are available.

3.1 Architecture
Figure 1 shows the high-level architecture of Dinomo. Dinomo consists of clients, routing nodes (RNs), KVS nodes (KNs), DPM, and a monitoring/management node (M-node). We describe these components and how a request flows between them.

Applications and users interact with Dinomo through clients. RNs are the client-facing tier that provides cluster membership and isolates clients from the internal variation of the KVS cluster. A client first contacts an RN to obtain cluster membership and caches the mapping of key ranges to various KNs. The client contacts the appropriate KN, which will then perform the read or write operation on its behalf. Each KN is equipped with general-purpose processors and a small amount of DRAM relative to the DPM capacity. The KN uses one-sided or two-sided RDMA primitives to access DPM over the interconnect [4]; note that the one-sided RDMA primitive can read or write data without involving the DPM processors. DPM has the large shared PM pool and limited computational power relative to KNs [54, 75, 89]. This asymmetry is deliberate: KNs are intended to run complex operations in the critical path, whereas DPM is intended to execute lightweight tasks outside the critical path, while keeping the cost of provisioning DPM low. The KN caches the data it fetches from DPM in its local DRAM, and responds to client requests. The M-node observes KNs statuses and workload characteristics to detect KN failures, load imbalance, or workload skew, and triggers a suitable reconfiguration.

Note that we separately deploy the different functional components of Dinomo to enable us to independently scale them up and down as required. It is also possible to co-locate some components at the expense of reducing the efficiency of policy decisions when scaling resources.

Assumptions. We assume all components in the Dinomo cluster are inter-connected through a reliable local network (either over TCP/IP or RDMA RC). The interconnect bandwidth between KNs and DPM is lower than the memory bandwidth of the PM itself, usually making network the bottleneck [3, 33]. KN failures are fail-stop and independent of DPM failures; when an KN fails, its DRAM contents are lost. DPM has internal mechanisms or hardware support to ensure high availability [38, 47, 54, 75, 91] and hardware-level memory protection [59, 72, 78, 88]. The M-node is always alive; this can be ensured via consensus and replication [43, 44, 63]. As the M-node deals with infrequent lightweight tasks, using consensus does not introduce performance bottlenecks.

3.2 Data organization on DPM
Figure 2 shows the data-plane components in Dinomo. Dinomo stores data (key-value pairs) and metadata (indexing structures) in DPM for providing durability and as the source of ground truth.

Storing data in logs. In response to a write request, a KN writes data to an exclusive log on DPM. This write is performed with a single one-sided write operation in the critical path. The log is broken into a series of segments. Since each KN handles requests on exclusive logical data partitions (§3.4), two KNs will never log a write for the same key. The DPM processors asynchronously merge the write operations in a log segment in order into the metadata index. Logs of different KNs may be merged into the index simultaneously.

Metadata index. The metadata index in DPM must satisfy the following requirements. First, KNs should not hold locks while performing index traversals; locks cause cross-node synchronization overheads. Next, even if a KN fails while performing an index traversal, other KNs should be able to make progress. Finally, the index should support concurrent and consistent updates, allowing DPM threads to perform non-conflicting updates in parallel. Most state-of-the-art concurrent PM indexes satisfy these requirements [6, 14, 31, 46, 53]; these PM indexes provide lock-free reads and log-free in-place writes. Thus, with such PM indexes, Dinomo provides a globally consistent view of data in a scalable manner, independent of the number of KNs.

Consistency. Dinomo guarantees linearizability, the strongest consistency level for non-transactional stores [77]. Dinomo ensures that a successful write request commits the data atomically in DPM, and that subsequent reads return the latest committed value. To satisfy linearizability, Dinomo merges data logs in request order to the metadata index. Other core design decisions like ownership partitioning across KNs (§3.4), and using indirect pointers for selective replication (§3.4), help provide linearizability. Before reconfiguration or after failure, Dinomo merges all pending logs from the KNs involved before allowing the other KNs to serve reads.
3.3 Disaggregated Adaptive Caching

It would be prohibitively expensive for KNs to do network round trips (RTs) for every read operation. To avoid these overheads, KNs use local DRAM to cache data and metadata. Because KNs have limited memory, efficient caching is crucial for high common-case performance. We introduce Disaggregated Adaptive Caching (DAC), a novel caching scheme to efficiently use DRAM at KNs.

Motivation. As DPM is directly accessible to KNs via one-sided RDMA operations with low latency owing to its byte addressability, KNs can cache not only data in the form of values but also metadata in the form of shortcuts. A value entry keeps the entire copy of a DPM value, so the KN can access everything locally. A shortcut entry keeps a fixed 64-bit pointer to the value in DPM; accessing the data incurs a one-sided operation to DPM. If neither value nor shortcut are cached, accessing the value incurs significant overhead: the KN needs to traverse a metadata structure in DPM to find the value’s location and then access the value. Traversing metadata structures like trees, skip lists, or chaining lists in hash tables, requires multiple RTs to DPM or remote procedures in DPM, both of which have much higher overheads than a single one-sided operation [2, 81, 92, 93].

Caching values improves performance relative to caching shortcuts, but requires more cache space. This raises an interesting question: is it better to cache a few values with no overheads upon cache hits, or a larger number of shortcuts with fixed hit overheads?

The answer is simple in extreme cases: in highly skewed workloads, where a small number of hot key-value pairs can fit in the cache, storing values is better. When workloads are close to uniform distribution with total size larger than the cache, storing shortcuts is better. Unfortunately, most workloads fall between these two extremes and offer no clear answer. A simple static caching policy may reserve some fixed ratio of cache space for storing values and devote the rest to shortcuts. What should this ratio be? We observe that the efficient ratio is dependent on workload patterns and aggregate memory available for caching. In a disaggregated system like Dinomo that has autoscaling, neither workload patterns nor memory available is known ahead of time, ruling out static policies.

Adaptive Policy. We introduce DAC, a novel caching policy that dynamically selects the ratio of values to shortcut entries as needed. This policy automatically adapts to the changes in workload patterns and to the changes in the aggregate memory space for caching at KNs due to cluster reconfiguration, as shown in Figure 2.

Insight. 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 cost of an index lookup) to one, while caching a value instead of a shortcut reduces RTs from one to zero. Thus, 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. Table 3 details the policy.

In DAC, values can be demoted to shortcuts and shortcuts can be evicted. Shortcuts can also be promoted to values.

Demotions. Demotions occur on cache misses to make space for a new cache entry. To demote a value to a shortcut, we pick the least-recently-used key, leveraging temporal locality. To evict a shortcut, we pick the least-frequently-used key, in order to preserve frequently used keys in the cache and cater to skewed workloads.

Promotions. Promotions depend on whether the benefits from caching a value outweigh the benefits from caching a suitable number of shortcuts. To determine if a shortcut $P$ needs to be promoted to a value, we use the following calculation. If at least $N$ least-frequently-used shortcuts need to be evicted to make space for caching one value, then the shortcut $P$ needs to satisfy the following relation to be promoted:

$$\text{Hits}(P) \times \text{Avg. shortcut hit RTs} \geq \sum_{i=1}^{N} \text{Hits(Shortcut)} \times \text{Avg. cache miss RTs}$$

This formula accounts for the two elements of the trade-off: the differences in the value and shortcut sizes, and the differences in the cost of a value miss and a shortcut miss. The left side of the inequality is the number of round-trips saved if we promote shortcut $P$ to a value; the right side is the number of additional round-trips incurred if we evict $N$ shortcuts to make space for the promotion of $P$. We promote if the savings are greater than the penalty. Note that the Avg. shortcut hit RT is always one, but the Avg. cache miss RT needs to be determined experimentally, which is done by keeping a moving average of past requests.

3.4 Ownership Partitioning

If multiple KNs cached the same value, this would incur consistency overheads (e.g., cache invalidation) from ensuring linearizability. Dinomo sidesteps this via ownership partitioning (OP). Owing to the DPM architecture, where KNs are disaggregated from the shared PM pool, data access and ownership can be independent considerations: it is possible to partition ownership while sharing access to data. This insight motivates OP, which strikes a balance between shared everything and shared nothing. OP allows KNs to cache unique data, avoid consistency overheads, and thereby achieve high scalability. Although similar ideas have been previously used in other contexts [1, 7, 13, 80], we are the first to adapt it for DPM.

Central Idea. KNs have exclusive but temporary ownership of logical, disjoint partitions of data. At any time, a partition is accessed by only one KN—its designated owner. OP allows KNs to scale without reorganizing data and metadata.
Partitioning the ownership. Routing nodes maintain the mapping of key ranges to their owner KNs. Clients’ requests are routed to the appropriate owner KN. The owner KN can use its local DRAM to cache data and metadata with high cache locality and provide good read performance. Dinomo does not require cache coherence protocols at KNs, as KNs have exclusive access to their partitions. As scaling KNs increases the total DRAM available for caching, OP scales performance by utilizing the DRAM cache effectively (no redundant copies) and avoiding consistency overheads.

Ownership metadata. Dinomo uses consistent hashing to assign the primary owners for key ranges; Dinomo is compatible with other (e.g., key-range or hash-based) partitioning algorithms. Within a KN, a key range is further partitioned among various threads. Both KNs and RNs maintain the partitioning metadata in a global hash ring, which stores key-to-KVS node-IP mappings, and a local hash ring, which stores key-to-thread mappings.

Whenever the mapping changes, RNs are updated together with KNs. Clients cache routing information; when the mapping changes, the KN they contact will direct them to a routing node to get the latest mapping information. Each KN always knows the key range it is supposed to handle, and will refuse requests for other key ranges.

Benefits. Ownership partitioning provides multiple benefits: High performance. Dinomo achieves high performance in the common case by partitioning the ownership across KNs, allowing multiple KNs to cache unique data partitions with high cache locality. Scalability. By avoiding the overhead for maintaining consistency at KN caches, Dinomo achieves scalability. Lightweight reconfiguration. Dinomo can quickly change the number of KNs without physically reorganizing data or metadata; the current owner empties its cache, completes outstanding operations, hands ownership to the new KN, and the new owner begins serving requests. If a KN fails, partitions owned by the failed KN can be assigned to new owners that can immediately serve data.

Selective replication. Partition-based systems may suffer from load imbalance with highly skewed workloads. In these circumstances, adding more KNs does not distribute the load across available KNs. Even if a popular key’s value is cached in a KN, performance is bottlenecked by that KN’s processing or network capacity. Dinomo recognizes such scenarios and shares the ownership of highly popular keys across multiple KNs, effectively replicating such keys to provide scalability beyond a single node’s abilities. The replication metadata is stored along with the mapping information at RNs and KNs and handled similarly. Clients cache and use this metadata to route requests to primary and secondary owners.

Dinomo uses indirect pointers to allow KNs to share ownership and read or write the shared key-value pairs consistently. An indirect pointer points to a location in DPM that stores a pointer to the value at a new location and atomically updates the indirect pointer and then read the value; thus, shared keys pay a cost that is avoided by default. Removing sharing from the key requires the KNs that own the shared key to invalidate it in their caches. Once the invalidation is done, the indirect pointer is removed in DPM.

3.5 Reconfiguration

The M-node triggers reconfigurations to improve performance when SLOs are violated, to release under-utilized resources, or to tolerate KN failures. We first present those policy details and then explain our principled reconfiguration protocol.

Policy engine. The policy engine in the M-node governs when and what kind of reconfigurations to trigger. Our policy engine follows prior autoscaling work [84], with simplifications for Dinomo; for example, memory consumption is not a consideration in scaling KNs since the memory in a KN is used as a cache without overflow. The policy engine allows the configuration of the following parameters: average/tail latency SLOs, over-utilization lower bound, under-utilization upper bound, key hotness lower bound, and key coldness upper bound. The M-node periodically collects latency information from clients, the average KN occupancy (i.e., CPU working time per monitoring-epoch interval), and the average access frequency for keys from KNs. It then proactively detects the latency SLO violations and corrects them dynamically. Table 4 summarizes the reconfiguration scenarios.

Cluster membership changes. In Dinomo, cluster membership is changed under the following scenarios. First, the M-node may detect a KN failure and notify the alive nodes. Second, the M-node may detect a latency SLO violation (average or tail latency SLO) and find that all the KNs are over-utilized (the minimum occupancy of all KNs is larger than the over-utilization lower bound), which triggers the addition of a new KN. Third, the M-node may detect that there is an under-utilized KN (its occupancy is lower than the under-utilization upper bound); if the latency SLOs are not violated, this triggers that KN’s removal. While ownership mapping is being redistributed due to the membership changes, clients’ request latencies can briefly increase. To prevent the policy engine from over-reacting during the ownership redistribution, Dinomo adds or removes at most one node per decision epoch and applies a grace period to allow the system to stabilize before the next decision.

Ownership replication changes. If the M-node detects an SLO violation and notices that all KNs are not over-utilized, then the M-node identifies highly popular keys and increases their replication factor. In detail, the M-node considers a key to be highly popular if its average access frequency is greater than the key hotness lower bound. Dinomo increases the replication factor $R$ (the number of

| SLO          | KN occupancy | Key access freq. | Action          |
|--------------|--------------|------------------|-----------------|
| Satisfied    | Low          | -                | Remove KN       |
| Violated     | High         | -                | Add new KN      |
| Violated     | Normal       | High             | Replicate key   |
| Satisfied    | Normal       | Low              | De-replicate key|

Table 4: Policy violations and M-node action
secondary owners) of a hot key, based on the ratio between the average latency of the hot key and the average latency SLO. The M-node considers a key to be cold if its access frequency is below the key coldness upper bound. If the latency SLOs are met and none of the KNs are under-utilized (the M-node cannot remove any KN), the M-node identifies cold keys with high replication factors ($R > 1$) and dereplicates them ($R=1$).

**Fault tolerance.** DPM is the source of ground truth in DINOMO; it persistently stores data (key-value pairs), metadata (indexing data structures), and other policy information (ownership/replication metadata). KNs and RNs store soft state that can be reconstructed if a node fails. When a KN or RN fails, it retrieves the up-to-date policy information from DPM and rebuilds the ownership mapping of key ranges before resuming. Unlike RNs, a KN failure changes the ownership mapping among the alive KNs. The M-node ensures that the ownership mapping is corrected before allowing the failed KN to resume. After detecting a KN failure, the M-node picks one of the alive KNs; this KN sends a request to DPM to complete the pending operations in the log segments from the failed KN. Upon completion, the M-node broadcasts the failure to all DINOMO components. On receiving a failure message, KNs and RNs repartition the ownership mapping by updating their hash rings.

**Reconfiguration steps.** We now describe how DINOMO performs reconfigurations. Broadly, the following steps occur:

1. KNs participating in the reconfiguration are identified (KNs for which the ownership mapping changes)
2. The KNs become unavailable
3. DPM synchronously merges the data in logs for these KNs
4. The KNs get their new mapping information
5. The KNs become available, and the cluster continues operation
6. The mapping information in the remaining KNs (not participating in the reconfiguration) is updated asynchronously
7. The RNs are asynchronously updated with the new mapping information

The cluster can continue operation at step five because KNs will reject requests for key ranges they do not own. Thus, other KNs can be updated without blocking the nodes undergoing reconfiguration. In certain special cases, DINOMO can perform reconfiguration without blocking any KNs. This can happen when a new partition is being added to DINOMO (no previous owner to race with) or when a KN fails and its partitions are being redistributed. Note that there is no expensive data copying or movement during reconfiguration. This is the key property that enables lightweight reconfiguration for DINOMO.

### 3.6 Optimizations

DINOMO includes optimizations in its data path to reduce CPU bottlenecks and network utilization from DPM.

**One-sided & asynchronous post processing.** To minimize the CPU bottlenecks and network utilization, DINOMO’s data path uses *one-sided operations with asynchronous post processing*. With a one-sided operation (e.g., RDMA read, write, and atomic verbs), a KN executes directly on DPM without involving the DPM processor. One-sided operations have lower latency and higher bandwidth than two-sided operations (e.g., RDMA send and receive verbs) [20, 34, 57, 61, 82], but one-sided operations are limited in functionality [2]. For the best performance, DINOMO uses one-sided operations in the data path and delegates the post-processing of writes to the DPM processors asynchronously.

**One-sided reads.** For reads, an KN directly returns the value from its cache upon a value hit. On a shortcut hit, it performs a single one-sided operation to retrieve the value in DPM from the shortcut pointer. On a cache miss, the KN performs multiple one-sided operations to find the address of the value (index traversals), and uses another one-sided operation to fetch the value from that address.

**Asynchronous post processing of writes.** DINOMO batches multiple log entries into a log segment unit and writes them to DPM using a one-sided RDMA write operation. With OP, DINOMO can batch the writes for the keys in the same partition without consistency concerns. The post-processing to merge the writes into the metadata index is asynchronously handled by DPM processors off the critical path. DINOMO’s KNs cache the committed log segments to aid the subsequent reads to be served locally at the KNs without expensive network RTs to read the large log segments remotely. These optimizations have two benefits. First, they reduce the latency as well as network costs per operation. Second, they amortize the merging operation across all the write operations in a log segment (typically several megabytes in size). Because the merging is done asynchronously, the DPM processors can have lower computing power without significantly affecting DINOMO performance.

### 4 IMPLEMENTATION

We implement DINOMO in 10K lines of C++ code. We use the standard C++ library and several open-source libraries including ZeroMQ [66], Google Protocol Buffers [10], libibverbs [18], and the PMDK library [40]. This section discusses DINOMO’s DPM data structures, DAC implementation, and cluster management.

**DPM metadata index.** DINOMO uses RECIPE’s P-CLHT (Persistent Cache Line Hash Table) [46], which supports lock-free reads and log-free in-place writes, as its metadata index in DPM. P-CLHT is a chaining hash table aimed at minimizing the CPU-cache coherence and persistence overheads on PM. Each bucket in P-CLHT has the size of a single cache line and holds three key-value pairs [19]. The design allows each access/update to the hash table to incur only a single cache-line access/flush in the common case. For lock-free reads, P-CLHT employs atomic snapshots of key-value pairs. We modify the index to use RDMA reads for lookups. On hash collisions, KNs may have to perform multiple one-sided RDMA reads to traverse the hash chain and read the value. The cache-line-conscious bucket design of P-CLHT, cache-coherent DMA [20, 34], and out-of-place value updates allow us to avoid memory-access races [57, 71] between the updates by DPM processors and one-sided RDMA reads by KNs.

**DPM log segments.** DINOMO implements 8 MB log segments and handles variable length key-value pairs. KNs proactively preload log segments for their own use using two-sided operations. KNs log write operations into DPM log segments and cache them; upon cache misses in DAC, KNs have to search cached log segments to find the latest value. DINOMO implements Bloom filters atop cached log segments for quick membership queries. DINOMO maintains the
following invariant: unmerged log segments are cached in the KNs that wrote them. Due to OP, other KNs will not access these log segments, thus eliminating the need for read operations to check the unmerged log segments on other nodes. KNs can add a new log segment to DPM without blocking until their unmerged log-segment length reaches a certain threshold (default is 2); when the threshold is reached, the critical write paths are blocked until the DPM processors complete merging below the threshold. DINOMO logs write operations with commit-markers (e.g., a seal byte at the end of the entry [20, 50]) to DPM log segments to ensure crash consistency and to aid recovery. The DPM index directly points to the values stored in the log entries. Since KNs know the address of the log segments they write (and therefore where values are stored), they can produce and locally cache shortcuts to values in DPM without an extra round trip. To garbage collect stale log segments, DINOMO maintains per-log-segment counters that reflect the number of valid and invalid values in each log segment. Once the number of invalid values matches the total number of values in a log segment, a DPM processor garbage collects the log segment.

**DPM persistence.** While merging log segments, DINOMO’s DPM processing threads persist all the writes to the DPM index structure using CLWB, sfence, and non-temporal store instructions [68]. RDMA currently does not support durable RDMA writes. However, the proposed durable write in the IETF standards working document [73] behaves similar to a non-durable write, requiring one network round trip. Our implementation currently uses nondurable writes, and we plan to update these to durable writes once they become available [39].

**DAC.** DAC is implemented using standard C++ libraries. DAC uses two unordered maps to store values and shortcuts. Least recently used values and least frequently used shortcuts are evicted. The key access frequency is tracked using a multimap. The shortcut entries contain a pointer to a DPM value, and the DPM value length. The value entries have two more extra fields, an access count and a copy of the DPM value. Demoted values are cached as shortcuts, and shortcuts being promoted inherit their access counts to preserve their access history.

**Cluster management.** DINOMO uses Kubernetes [27] for cluster orchestration. Pods are the smallest deployable units in Kubernetes. Each DINOMO component is instantiated in a separate Kubernetes pod with a corresponding Docker [23] container. DINOMO uses Kubernetes to add/remove KN pods and restart failed pods. The M-node pod is colocated with the Kubernetes master. The M-node’s policy engine adds/removes KN pods by running simple bash scripts executing kubectl [62] commands to the Kubernetes master. The Kubernetes master keeps track of pod status using heartbeats, and the M-node uses this information to detect failures in KN pods.

5 EVALUATION

We evaluate the performance of DINOMO and study the breakdown of the benefits from Ownership Partitioning (OP), Disaggregated Adaptive Caching (DAC), and selective replication. We design our experiments to answer the following questions:

- Does DAC help reduce network round trips? How does it fare against other caching policies?
- How much does the DPM compute capacity impact DINOMO’s overall throughput?
- How does DINOMO fare against the state-of-the-art in terms of performance and scalability?
- What fraction of DINOMO’s benefits can be attributed to the OP architecture and the DAC caching?
- How elastic and responsive is DINOMO while handling bursty workloads, load imbalance, and KN failures?

**Comparison points.** As our baseline, we use Clover [75], a state-of-the-art and open-source key-value store designed for DPM. Clover has a shared-everything architecture with a shortcut-only cache at its KNs. KNs perform out-of-place updates to the data in DPM, and incur additional overheads to provide strong consistency. For example, stale cached entries require KNs to walk through a chain of versions to find the most recent data in DPM.

Besides Clover, we compare DINOMO with two variants, DINOMO-S and DINOMO-N. DINOMO uses three techniques: DAC, OP, and selective replication. DINOMO-S caches only shortcuts; it is otherwise identical to DINOMO. As the source code of AsymNVM [54] is not publicly available, we implement DINOMO-N to compare 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 without selective replication.

Comparing DINOMO-S with Clover highlights the benefits of partitioning ownership in OP, and comparing DINOMO with DINOMO-S shows the benefits from DAC. We also investigate the trade-off from sharing data in OP by comparing DINOMO with DINOMO-N.

**Experiment setup.** We use Kubernetes pods to represent all of the node instances in the DINOMO cluster. We restrict the host resources assigned to the pods depending on the node types’ features to emulate the asymmetric DPM architecture (i.e., KNs have more-capable computation but smaller memory than DPM). Each individual pod is pinned to a separate server for resource isolation purposes.

We deploy DINOMO on the Chameleon Cloud [35], an experimental large-scale testbed for cloud research. 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 port. We emulate PM using DRAM, as performance is constrained by the network rather than PM or DRAM: network latency (1–20 us) is at least 10x higher than DRAM or PM latencies (100s of ns); network bandwidth (7GB/s) is lower than PM bandwidth (32GB/s Read / 11.2GB/s Write) [3, 33].

The external servers that run application workloads, henceforth termed **client nodes**, and the routing service do not need a high-speed interconnect with the KNs or DPM. Hence, for client nodes and routing nodes (RNs), we use two-socket servers with AMD EPYC 7763 processors, 128 cores at 2.45 GHz in total, 256 GB of DRAM, and a 10 Gbps Ethernet NIC. Each client node uses 64 threads to run a closed-loop workload with one or more outstanding requests per thread. We use a single RN with 64 threads. The same routing layer is used across all KVS variants in our evaluation. In
Figure 3: Performance comparison of cache policies

Figure 4: Performance impact of DPM compute capacity. The log-write throughput approaches the max with 4 threads on DRAM, while requiring more threads on PM.

addition to the data plane components (KNs and DPM), Dinomo, Dinomo-N, and Dinomo-S use a control-plane instance for the M-node, which is deployed on a server (same server configuration as the RNS) with a single thread. For Clover, we use an extra IB-enabled server (same server configuration of the KNs) for its metadata server with 6 threads (4 workers, 1 epoch thread, 1 GC thread).

Workloads and configurations. We use YCSB-style workloads [17, 84] with five request patterns: read-only (100% reads), read-mostly (95% reads/5% updates and 95% reads/5% inserts), write-heavy (50% reads/50% updates and 50% reads/50% inserts). These workloads use 8B keys and 1KB values and the following Zipfian coefficients: 0.99 (the YCSB-default value) for moderate skew, 2 for high skew, and 0.5 for low skew (close to uniform). For each experiment, we first load 32GB 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 the client nodes and measure system throughput and other profiling metrics, averaging them over a 10-second interval.

5.1 Microbenchmark

We use micro-benchmarks to investigate several issues. We first consider whether DAC is an effective caching strategy. Next, we explore how much compute capacity DPM requires to prevent the asynchronous merging of writes from becoming the bottleneck; based on the results, we also discuss how using DRAM to emulate PM affects our results.

DAC. The KN caches can be used to store values, shortcut pointers, or a mix of both. To evaluate DAC against different caching strategies, we use a single KN with 16 threads. We first load 30M key-value pairs into Dinomo with 8B keys and 64B values. We then run a read-only workload with a working set of 1.5M uniformly-distributed keys (5% of the dataset) to evaluate performance. We generate the workload locally and measure the peak throughput within the KN by varying the available DRAM for caching from 1%–16% of the dataset size. We configure Dinomo to use different caching policies (Figure 3). The static-X policies reserve X% of their cache size for storing values; the rest of the cache is used for shortcuts. All non-DAC policies use LRU to evict entries.

Figure 3 shows the read throughput obtained with the different cache policies. With an aggregate cache size of 2% of the dataset, a shortcut-only cache performs best, whereas with a cache size 4X as large, a value-only cache performs best. The aggregate cache size is dependent on the number of active KNs, which may dynamically change with cluster reconfiguration or KN failures. Therefore, a static caching policy is not a good fit. The right policy depends upon the workload patterns and aggregate cache size.

Despite not knowing the workload patterns or the aggregate cache size, DAC is within 16% of the best performing policy, in all settings. With a medium-sized cache that is 4% of the dataset size, DAC exceeds the performance of both shortcut-only and value-only caching policies by taking advantage of both.

Asynchronous post processing. A delay in merging log segments due to the limited compute capacity in DPM can block the critical path of KNs writing logs. To evaluate this impact from the worst-case scenario in our setup, we run an insert-only workload using 16 KNs and 8 client nodes; this is the most compute-intensive workload, as it incurs structural changes to the DPM index (e.g., resizing hash table). We first load 32GB of data and then run the workload writing up to 100GB of data into DPM with 8B keys and 1KB values. We measure the peak throughput of log writing and merging for different DPM thread counts. For the log-write throughput, we collect the aggregate throughput across 16 KNs every 10 seconds for 30 seconds and average them; the log-write max is the maximum throughput the KNs can obtain if they never wait for DPM to merge logs. To measure the merge throughput, we pre-generate log segments locally on DPM for the dataset and then measure the performance of merging those log segments. As our testbed has no IB-enabled PM machines, we measure the merge throughput on PM using a local PM machine (Intel Xeon Silver 4314 CPU with 16 cores at 2.4GHz and 512GB Intel Optane DC PM on 4 NVDIMMs) to estimate the impact from using PM for DPM.

We make a number of observations based on the results in Figure 4. First, we observe that to write logs at the maximum rate, DPM should have enough computing capability to merge at the log-write max rate; four or more threads are required for our setup. Second, we observe that because of PM’s higher access latency, PM merge throughput is lower than DRAM; when using four threads, the lower PM merge throughput can become the bottleneck. Third, we confirm despite in-DIMM write amplifications [33, 86], merge operations consume PM write bandwidth only up to 2GB/s (monitored by PCM [64]); 9.2GB/s out of the maximum (11.2GB/s) is still available to absorb incoming writes from the KNs over the network, making the network (7GB/s) the bottleneck rather than PM.

We conclude that, in some scenarios, using PM instead of DRAM requires a higher number of DPM threads to prevent the merging
We use workloads with moderate skew (Zipf 0.99) to observe the performance of Dinomo KNs in read-dominated workloads because of network bottlenecks. With 16 KNs, to either a network bottleneck or the CPU bottleneck from its nodes to run these workloads and measure the peak throughput by persisting multiple write operations in a batch with 1 RT on DPM. Overall, we see that Dinomo outperforms Clover by at least 3.8× across all workloads. With 1 KN, Dinomo caches more shortcuts, incurring 1 RT at a cache hit, while with 16 KNs, Dinomo caches more values, and hence takes fewer RTs/op (0.1 RTs/op across all workloads). Dinomo has fewer RTs/op in write-heavy workloads on average than read-dominated workloads, as KNs persist multiple write operations in a batch with 1 RT on DPM. Overall, we see that DAC is effective in reducing RTs to DPM.

5.3 Elasticity

We now demonstrate Dinomo can elastically scale the number of KNs, balance loads across KNs, and tolerate failures. We use a workload with 50% reads and 50% updates with three different skew distributions. When a reconfiguration is triggered in this workload, any pending writes must be merged to DPM before the reconfiguration can proceed. We run a client node with one outstanding request per thread at a time.

Policy variables. We set the parameters of the policy engine (§3.5) and design the experiments to trigger various forms of reconfiguration. We use an average latency SLO of 1.2ms and a tail latency SLO (99-percentile latency) of 16ms. The over-utilization lower bound is configured to be 20% KN occupancy, and the under-utilization upper bound is set to 10% KN occupancy. Furthermore, we configure the key-hitness lower bound to 3 standard deviations above the mean key access frequency and the key-coldness upper bound to 1 standard deviation below the mean. Note that the goal of the experiments is to study the elasticity of Dinomo under various scenarios; we chose these policy parameters as simple triggers for these scenarios, not as an indication of the best policies.

OP enables scalable performance. We observe that increasing the number of KNs from 1 to 16 reduces the cache hit ratio in Clover across all workloads. This performance drop is counterintuitive, as the DRAM available for caching increases with the number of KNs. However, in shared-everything architectures KNs can handle any request, so multiple KNs may incur cache misses on the same key. With more KNs, even with moderate skew, the redundant cache misses increase. In summary, shared-everything architectures do not provide good cache locality and prevent the efficient use of KN-side memory for caching. In contrast, OP partitions the ownership of keys across KNs, providing high cache locality for requests and eliminating redundant shortcuts at multiple KNs. Note that, for these workloads, Dinomo-S sees a 100% hit ratio across all KNs and with any number of KNs.

DAC boosts performance and scalability. Dinomo has a higher cache hit rate (from values) with more KNs and takes fewer RTs/op, compared to both Dinomo-S and Clover. Dinomo has higher network costs: up to 10× more RTs/op than Dinomo. Clover is even worse: from 4× to 87× more RTs/op than Dinomo, due to shortcut-only caching and a lack of locality that results in consistency overheads and redundant caching. The aggregate memory available for caching increases with KNs for all systems. However, DAC helps KNs cache more values (as opposed to shortcuts), and thus incur fewer round trips to DPM per operation. In Dinomo, the cache hit % from values increases from 52% with 1 KN up to 88% with 16 KNs across all workloads. With 1 KN, Dinomo caches more shortcuts, incurring 1 RT at a cache hit, while with 16 KNs, Dinomo caches more values, and hence takes fewer RTs/op (0.1 RTs/op across all workloads). Dinomo has fewer RTs/op in write-heavy workloads on average than read-dominated workloads, as KNs persist multiple write operations in a batch with 1 RT on DPM. Overall, we see that DAC is effective in reducing RTs to DPM.

5.2 Performance and Scalability

We now demonstrate Dinomo-S vs. Clover highlights the benefits from OP. Dinomo vs. Dinomo-S highlights the benefits from DAC. Dinomo vs. Dinomo-N shows the performance trade-offs between sharing data and OP.
Auto scaling. We evaluate DIAMOMO with bursty, irregular workloads and compare its elasticity in scaling KNs with DIAMOMO-N. We were unable to run Clover for this experiment because Clover has no implementation for auto-scaling KNs. We produce scenarios where a new KN is required or an existing KN is no longer needed. Recall that DIAMOMO adds new KNs automatically only if a latency SLO is violated. The KNs are over-utilized, and an additional KN is available. DIAMOMO automatically evicts a KN only if the latency SLOs are met and the KN is underutilized. The grace period after each reconfiguration is configured to 90 seconds.

To produce a bursty workload, we start running the workload with low skew (Zipf 0.5) on DIAMOMO using 1 client node for 20 seconds. We then increase the load on DIAMOMO by 7× by adding 7 additional client nodes. We observe the performance of DIAMOMO for a few minutes until it stabilizes, and at the 230-second mark, we remove 7 client nodes to lower the load by 7× again. Figure 6 shows the behavior of DIAMOMO and DIAMOMO-N during this experiment. DIAMOMO and DIAMOMO-N meet the latency SLOs until the load increases at 30 seconds, when the M-node detects a latency SLO violation: the tail latency SLO is exceeded. The M-node then observes that KNs are over-utilized (the minimum KN occupancy in DIAMOMO is about 35%), and hence corrects the situation by adding a new KN. Once the new KN comes online at 40-50 seconds, DIAMOMO shows a brief latency increase and throughput dip, as the nodes update their hash rings. However, DIAMOMO-N experiences a 40-second latency spike and throughput dip at 60 seconds, where the throughput drops to 0 due to the processing delay during data reorganization. After a 90-second grace period, although the average latency SLO is met, the tail latency SLO is still violated. DIAMOMO and DIAMOMO-N react to the situation by adding another KN. Again, DIAMOMO only sees a brief increase in latency, while DIAMOMO-N’s latency increases for 30 seconds. After the grace period, as both latency SLOs are met, DIAMOMO and DIAMOMO-N do not take any further actions.

At 230 seconds, the load is suddenly reduced. In the next 10 seconds, the M-node detects an under-utilized KN with lower than 10% occupancy. As the latency SLOs are met, the policy engine triggers the KN eviction. While removing the under-utilized KN, DIAMOMO sees a brief rise in average and tail latency without violating SLOs. However, DIAMOMO-N shows a 20-second throughput dip and latency spike before stabilizing.

Overall, we see that DIAMOMO is more responsive with fewer throughput and latency disruptions than DIAMOMO-N and can automatically scale KNs as required by changes in load.

Load balancing. We now describe how DIAMOMO handles non-uniform load on its KNs and scales its throughput for hot spots, in comparison to DIAMOMO-N and Clover. To handle these scenarios, recall that DIAMOMO uses selective replication; this mechanism is triggered only if a latency SLO is violated due to a few hot keys and the KNs are not over-utilized.

For these experiments, we use a skewed workload with 8 client nodes and 16 KNs. We start the experiments with a low-skew workload (Zipf 0.5) and then switch to a highly-skewed workload (Zipf 2). DIAMOMO’s policy engine checks that the KNs are not over-utilized (the minimum KN occupancy is lower than 10%) and identifies that the latency SLO is violated due to 4 hot keys. As a result, the policy engine triggers the selective replication of the 4 keys. Figure 7 shows the KVSs’ behavior during the experiment.
We observe that Dinomo stops serving requests while reshuffling data. The time to reorganize performance at 50 seconds, where the throughput drops to 0 as it across other alive KNs. These steps take less than 10 ms.

Latency increases by 1.2 for 2 minutes using 8 client nodes, and simulate a KN failure at cluster with 16 KNs, we run a moderate skew (Zipf 0.99) workload Dinomo resilience and elasticity of Fault tolerance. Finally, we induce a KN failure to compare the spots as a better alternative to shared-everything. cation with OP non-hot keys and continue to cache their values adaptively. selectively replicates only the hottest keys while restricting KNs to processing hot keys restricts KNs from caching values. Hence, Dinomo's performance stabilizes in about 1 minute and it outperforms Clover by almost Dinomo's performance, hot keys Dinomo's performance quickly, and satisfies all SLOs.

Overall, compared to Dinomo-N, Dinomo recovers from KN failure faster since it is not required to reorganize data owing to the data sharing in OP. Similar to Clover, Dinomo stabilizes its performance quickly, and satisfies all SLOs.

6 RELATED WORK

We place our contributions in the context of relevant prior work. DPM architectures. OP follows the idea that just because you can share, it does not mean you should share. This observation has been made before in other contexts. Storage Area Networks provide storage disaggregation in a data center [7], where volumes could be shared among hosts, but often they are not [13]. Key-value stores provide storage disaggregation in the cloud, where data can be shared among nodes, but applications may choose not to [80]. Fine-grained logical partitioning has been proposed to support live reconfigurations in in-memory key-value stores [1, 42], in-memory databases [22], and graph processing [85]. Even multiprocessor shared-memory systems sometimes forgo sharing of data structures among threads, choosing instead to partition data [8, 11, 48]. Our work demonstrates that partitioning logical ownership while sharing physical data and metadata in DPM provides high performance and lightweight reconfigurability.

DAC. Adaptive caching policies have been explored in other contexts, illustrating how a single cache can be used for multiple purposes or how a replacement policy can consider multiple behaviors. For example, the Sprite operating system shared its memory between the file system buffer cache and the virtual memory system [58]. The Adaptive Replacement Cache (ARC) uses a replacement policy that balances between recency and frequency of accesses [55]. In contrast to these systems, which use fixed-size cache entries with uniform miss penalties, DAC manages a cache where different types of entries (e.g., values vs. shortcuts) have different sizes and varying miss penalties. The novelty of our scheme arises from a new setting (DPM) where adaptivity is essential.

7 CONCLUSION

DPM is a promising new architecture for building KVSs. Prior DPM KVSs have had to sacrifice at least one of three desirable characteristics: high common-case performance, scalability, and lightweight reconfiguration. We present the Dinomo KVS, which uses a novel combination of techniques to achieve these properties simultaneously, which we demonstrate through empirical evaluation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and members of the UT Systems and Storage Lab for their insightful comments and constructive feedback. This work was supported by NSF CAREER #1751277 and donations from VMware, Google, and Meta. Sekwon Lee was also supported by the Microsoft Research PhD Fellowship.
[92] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data. 741–758.

[93] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. Onesided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In Proceedings of the 2021 USENIX Annual Technical Conference. 15–29.