LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs

Zerui Guo
University of Wisconsin-Madison

Hua Zhang
Beihang University

Chenxingyu Zhao
University of Washington

Yuebin Bai
Beihang University

Michael Swift
University of Wisconsin-Madison

Ming Liu
University of Wisconsin-Madison

Abstract

The recent emergence of low-power high-throughput programmable storage platforms–SmartNIC JBOF (just-a-bunch-of-flash)–motivates us to rethink the cluster architecture and system stack for energy-efficient large-scale data-intensive workloads. Unlike conventional systems that use an array of server JBOFs or embedded storage nodes, the introduction of SmartNIC JBOFs has drastically changed the cluster compute, memory, and I/O configurations. Such an extremely imbalanced architecture makes prior system design philosophies and techniques either ineffective or invalid.

This paper presents **LEED**, a distributed, replicated, and persistent key-value store over an array of SmartNIC JBOFs. Our key ideas to tackle the unique challenges induced by a SmartNIC JBOF are: trading excessive I/O bandwidth for scarce SmartNIC core computing cycles and memory capacity; making scheduling decisions as early as possible to streamline the request execution flow. LEED systematically revamps the software stack and proposes techniques across per-SSD, intra-JBOF, and inter-JBOF levels. Our prototyped system based on Broadcom Stingray outperforms existing solutions that use beefy server JBOFs and wimpy embedded storage nodes by 4.2×/3.8× and 17.5×/19.1× in terms of requests per Joule for 256B/1KB key-value objects.

CCS Concepts

• **Information systems** → **Distributed storage**; • **Hardware** → *External storage*; • **Computer systems organization** → Cloud computing;

Keywords

Energy efficiency, Key-value store, Disaggregated storage

ACM Reference Format:
Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. 2023. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3603269.3604880

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission
and/or a fee. Request permissions from permissions@acm.org.
ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 979-8-4007-0236-5/23/09. . . $15.00
https://doi.org/10.1145/3603269.3604880

1 Introduction

Persistent key-value storage systems have become a crucial distributed infrastructure system component, serving as the basis for a variety of data-intensive applications: web indexing [39, 41, 85], e-commerce platforms [38, 50], social networking [37, 89, 90], photo store [36, 88, 91], IoT sensing [63, 64], and more. It is of paramount importance for the clusters that house these workloads to provide both high performance and low power draw, thereby improving the energy and cost efficiency of data center computing facilities.

People have deployed these workloads over two types of platforms. One is a power-hungry server JBOF [3, 6, 18] that is equipped with beefy server-grade processors based on a sophisticated microarchitecture, multi-channel big memory, massive I/O bandwidth, and tens of terabytes of storage capacity. Server JBOF-based persistent key-value stores [24, 25, 62, 66, 70, 79] can indeed deliver millions of requests per second throughput but come at the cost of extremely high power consumption. The other one is an ultra-low power embedded storage node [12, 13, 20], which has been explored by early pioneering systems like FAWN [34]. It uses an array of such wimpy nodes coupling with an efficient software stack, significantly reducing the cluster power draw and improving the overall system energy efficiency.

However, FAWN-like approaches present drawbacks under scaled deployment. One fundamental reason is that each wimpy data node applies a well-balanced CPU-DRAM-IO architectural design, matching its storage capacity and bandwidth with the constrained computing capacity of an embedded CPU. When deploying at larger scales (e.g., 100+GbE), one needs to add a large number of back-end storage nodes, which would consume a similar amount of power (e.g., 400-500W) as a single server platform or even more. Worse yet, the extra system management complexity as well as the prohibitive capital/operational cost of networking fabric (rack switches) and power distribution units make the solution unattractive and prevent it from practical deployment.

Lately, a new type of low-power high-throughput SoC-based (system-on-a-chip) programmable storage platforms (i.e., SmartNIC JBOFs) [2, 5, 8, 9, 11] have emerged in the market. They consist of either an embedded multi-core processor or an FPGA, an array of domain-specific accelerators, a small amount of onboard DRAM (8–16GB), a server-grade NIC (e.g., 100+GbE), a PCIe switch, and a collection of NVMe SSDs. Such a platform, operating similarly to a server JBOF, consumes at least one order of magnitude lower power (e.g., 52.5W for the Broadcom Stingray PS1100R [2]). It uses a slightly upgraded CPU than the one in an embedded storage node, but presents a significant increase in network/storage I/O bandwidth and storage capacity. This not only suggests us a new

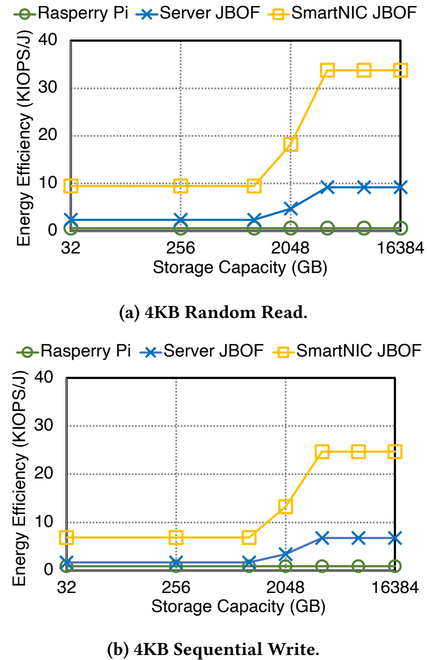
platform to build low-power and high-performance persistent key-value store, but also motivates us to rethink the system architecture for energy-efficient data-intensive computing.

However, translating these intuitive opportunities into energy efficiency gains entails challenges. We start with porting FAWN [34] and other software stacks over an array of SmartNIC JBOFs, and find out that many of the prior proposed techniques become either invalid or inefficient. This is mainly because SmartNIC JBOFs apply an ***extremely imbalanced*** architectural design in terms of compute, memory, and I/O, contradicting the platform assumptions that previous systems based on server JBOFs or embedded storage nodes held before. First, flash density scales much faster than DRAM, resulting in a highly skewed storage hierarchy. This indicates that each key-value object can use much less than one byte of in-memory space for indexing. Second, the network and storage I/O throughput that a core should drive (i.e., computing density) has increased by one or two orders of magnitude, meaning that (1) there is little computing room left for each I/O; (2) any execution waiting or stall (e.g., due to head-of-line blocking) would jeopardize performance significantly. Third, uneven load distribution has emerged as a serious issue that can easily cause system overloading, given that the computing cycles of a SmartNIC JBOF are scarce. This requires careful cluster-level traffic control and judicious I/O scheduling.

This paper presents the design and implementation of **LEED**, a distributed and replicated key-value store that runs over an array of SmartNIC JBOFs to combat these challenges. Our key ideas to tackle the unbalanced nature of a SmartNIC JBOF are: (1) trading excessive I/O bandwidth for scarce SmartNIC core cycles and memory capacity; (2) making scheduling decisions as early as possible to streamline the request execution flow. LEED represents a careful synthesis of conventional networking and system techniques along with novel data structures and algorithms proposed across three different levels, i.e., per-SSD, intra-JBOF, and inter-JBOFs.

- First, LEED designs a specialized key-value store using a circular log data structure and a DRAM/Flash hybrid indexing scheme to remedy the skewed storage hierarchy;
- Second, to maximize the SmartNIC core usage, LEED streamlines key-value command processing and optimizes bookkeeping compactions with prefetching and pipeline parallelism. It further applies a token-based intra-JBOF I/O engine and inter-JBOF request scheduler that adapt client issuing rates to the SSD serving capability;
- Third, to mitigate the I/O imbalance, LEED develops an intra-JBOF data swapping mechanism that temporarily redirects overloaded writes to unloaded SSDs; enhances the inter-JBOF chain replication with the request shipping capability to expand the system read serving throughput;

We prototype LEED using Broadcom Stingray PS1100R boards and compare it with other persistent key-value stores deployed atop two traditional platforms: FAWN [34] over embedded storage nodes (using Raspberry Pi 3 Model B+) and KVell [62] over Intel Xeon-based server JBOFs. Our evaluations using YCSB workloads show that LEED outperforms KVell by 4.2×/3.8× and FAWN by 17.5×/19.1× in terms of requests per Joule for 256B/1KB key-value object cases. We further demonstrate the efficacy of proposed techniques under various controlled experiments and measure system performance under failures.



(a) 4KB Random Read.

(b) 4KB Sequential Write.

Figure 1: Energy efficiency comparison among three platforms when performing persistent I/Os. We increase storage capacity by first maxing out NVMe drives on a node (only for Server/SmartNIC JBOFs) and then adding more nodes.

2 Motivation

This section introduces the SmartNIC JBOF platform and its potential, discusses prior low-power and high-performance data store solutions, and illustrates the challenges of using SmartNIC JBOFs for building a persistent key-value store.

2.1 SmartNIC JBOF: a Promising Platform

SmartNIC-based JBOFs [2, 5, 8, 9] are a new type of programmable storage platform, offering significantly lower power and higher cost efficiency than traditional server-based solutions. Hardware vendors couple a Multicore-SoC or FPGA-based SmartNIC with NVMe drives, connecting them via an onboard or standalone PCIe switch. Take the Broadcom Stingray [2] as an example (which is our prototyped testbed). It encloses a PS1100R SmartNIC, a PCIe carrier board (containing a PCIe switch), up to 4 NVMe SSDs, and a standalone power supply. The SmartNIC has an 8-core ARM A72 CPU running at 3.0GHz, 8GB DDR4-2400 DRAM (along with 16MB L2 cache), FlexSPARX [1] acceleration engines, 100Gb NetXtreme Ethernet NIC, and a PCIe Gen3 root complex controller. The PCIe switch offers ×16 PCIe 3.0 lanes (15.75GB/s theoretical peak bandwidth) and allows either 2 ×8 or 4 ×4 PCIe bifurcation. A Stingray storage box with four Samsung DCT983 960GB SSDs is listed as roughly one-third of a server JBOF with similar I/O configurations.

The SmartNIC JBOF is a promising platform and offers us an opportunity to drastically reduce overall power draw and improve

energy efficiency when running data-intensive workloads. We measure the wall power of a Stingray JBOF via a *Watts Up Pro* meter [21] and find out that its maximum active power (when driving all SSDs) is up to 52.5W, nearly one-fifth to one-fourth of a typical server JBOF. We also consider another storage platform–an ultra-low power embedded node (e.g., PCEngine Alix [20] and Raspberry Pi [12, 13]) used by previous systems [34, 77]. Take the Raspberry Pi 3 Model B+ as an example. Its active power is around 3.9-4.2W but with a much smaller storage capacity (32GB) and lower I/O bandwidth (60-80MB/s). Hence, when building a data store cluster using SmartNIC JBOFs, server JBOFs, or embedded storage nodes, SmartNIC JBOFs would deliver the best energy efficiency in terms of IOPS per Joule regardless of capacity. As shown in Figure 1, considering the 16TB case and system configurations of our testbeds, SmartNIC JBOFs outperform server JBOFs by 4.8×/4.7× and Raspberry Pi nodes by 56.5×/26.4× when performing 4KB random read/sequential writes across the entire capacity, respectively.

2.2 Prior Solutions

Researchers have built various high-performance and low-power persistent key-value stores over either server JBOF or embedded storage node platforms.

2.2.1 Server JBOF. LevelDB [24] and RocksDB [25] are two widely-used key-value stores based on log-structured merge trees (LSMs). SILT [66] develops compact indexing data structures using partial-key cuckoo hashing and entropy-coded tries to reduce the DRAM footprint when serving small key-value objects. WiscKey [70] designs a new data layout (i.e., separating keys and values) for these LSM-tree-based data stores such that it decouples key sorting and garbage collection phases. PebblesDB [79] proposes the fragmented log-structured merge tree data structure to reduce data rewrites. KVell [62], a recent key-value store for NVMe SSDs, applies a shared-nothing architecture to avoid synchronization overheads and uses efficient device access batching to max out the SSD read/write bandwidth.

Limitations: Server JBOFs are indeed a high-performance platform to build persistent key-value stores. Prior systems have explored different approaches to fully utilize their massive I/O bandwidth. However, as discussed above, its power draw is much higher than a SmartNIC JBOF or embedded node, making it unattractive to explore a low-power design. Further, high power consumption also increases the capital cost for operating enterprise clusters and data centers [27, 32].

2.2.2 Embedded Storage Node. FAWN [34] is the primary system that exploits the use of such nodes for serving persistent key-value requests. It uses an array of wimpy storage nodes to reduce the power draw and achieve high energy efficiency. A FAWN cluster consists of front-end servers and back-end FAWN-KV nodes. The front-end node takes client requests, forwards them to a back-end datastore node based on the key hash, and returns the results back. Each back-end node is divided into a series of non-overlapping virtual partitions for load balancing and failover acceleration. Its software system comprises four components: a log-structured key-value store that harnesses flash access properties, a consistent hashing scheme for efficient item placement and lookup, a chain replication mechanism to achieve per-key strong consistency, and a load adaption scheme via continuously adding/removing back-end nodes.

	Embedded Node	Server JBOF	SmartNIC JBOF
Storage hierarchy skewness: the size ratio between Flash and DRAM	16	64	1024
Computing density (for network): the bandwidth a core should deliver	0.25 GbE	3.2 GbE	12.5 GbE
Computing density (for storage): the IOPS a core should drive	5K	125K	500K
Maximum load: the request demand that a node receives with high prob.	$0.01m + \Theta(\sqrt{0.02m})$	$0.33m + \Theta(\sqrt{0.16m})$	$0.33m + \Theta(\sqrt{0.16m})$

Table 1: Data store node comparison among embedded, server JBOF, and SmartNIC JBOF. We use our testbeds as an example, i.e., Raspberry Pi 3 Model B+, Supermicro 2U server, and Stingray PS1100R. The computing density for storage considers the IOPS of 4KB random read. m is the client request rate. When calculating the maximum load, we assume a 100-node embedded cluster and a 3-node Server/SmartNIC JBOFs.

Limitations: FAWN achieves both low power and high performance at a small and modest scale. However, it presents diminishing returns under scaled deployment and even becomes cost-inefficient. This is because each back-end node presents a well-balanced CPU-DRAM-IO architectural design, which matches storage capacity and throughput with the limited computing capacity of an embedded CPU. To handle a rising demand, one has to increase the number of back-end nodes significantly. This would bring in two issues. First, the aggregated power and cost would outweigh a server-based solution. For example, supporting 100GbE bandwidth requires 1000 low-end 100Mbits PCEngine Alix [20] (used by FAWN) or 100 high-end 1Gbits Raspberry Pi [12, 13] boards, consuming around 3000-5000W and 400-500W, respectively. Meanwhile, the cluster further needs a couple of rack switches for physical connectivity which also consumes power. Second, the infrastructure managing complexity has increased exponentially, including cabling, space planning, packaging, power management, and failure diagnostics. This prevents FAWN from wide adoption and makes it less favorable than server JBOFs under today's I/O requirements.

2.3 Challenges

Even though SmartNIC JBOFs are promising to achieve low power and high performance simultaneously, translating these intuitive opportunities into efficiency gains entails great challenges. Fundamentally, a SmartNIC JBOF is a significantly ***imbalanced computing system*** in terms of compute, memory, and network/storage I/O. In comparison, a server JBOF or embedded node is much more balanced, where the platform couples just the right amount of DRAM, network, and flash with its CPU's computing capability. As a result, the highly-skewed storage hierarchy, limited per-IO computing cycles, and high-probability load imbalance induced by SmartNIC JBOFs make prior techniques either invalid or inefficient.

Challenge #1: data indexing under a highly-skewed storage hierarchy (*C1*). Compared with the other two platforms, the size ratio between Flash and DRAM of a SmartNIC JBOF has increased by two orders of magnitude (Table 1), indicating that a DRAM-only indexing mechanism would become infeasible. For example, the FAWN datastore uses 1GB DRAM to index up to 16GB of persistent data. Even though it takes the constrained memory capacity into consideration, each key-value object still requires 6

bytes in-memory hash index, including a 15-bit key fragment, a valid bit, and a 4-byte pointer to the location of the data log. Suppose a SmartNIC JBOF holds 8TB flash space with at most 8GB DRAM (like our testbed). Using FAWN's approach, indexing 256B objects would require 192GB DRAM (to store the metadata), significantly exceeding the SmartNIC JBOF DRAM capacity. An effective design should use less than half a byte per object. Hence, one should re-think how to design an indexing mechanism that adapts to such a skewed storage hierarchy. This might even require revamping the data store architecture and key-value processing path.

Challenge #2: request execution under high I/O density and small computing headroom (*C2*). The computing density, defined as how much IOs a core should handle, is also increased by two orders of magnitude (Table 1). Computing power is a scarce resource on a SmartNIC JBOF, and the amount of cycles that a core spends on an I/O request is limited. Taking the Stingray as an example, its 3.0GHz 8-core ARM processor has to handle 100Gbps networking and up to 4M NVMe IOPS, whereas an embedded node in FAWN only needs to drive at most 1Gbps and 20K IOPS, respectively. When saturating the I/O bandwidth, a back-of-the-envelop calculation indicates that the maximum tolerable latency for processing a MTU-sized packet and a 4KB I/O is just $0.96\mu s$ and $5\mu s$ on the SmartNIC JBOF, where a FAWN node has much more headroom (i.e., $48\mu s$ and $400\mu s$). The server JBOF resolves this issue by employing high-end manycore server CPUs, making today's storage servers much beefier [28, 31]. Therefore, we should carefully orchestrate I/Os along the datapath between the network interface and NVMe drives to avoid unnecessary execution synchronization or blocking. At the cluster scale, we should also consider harvesting as many cross-node available computing cycles as possible.

Challenge #3: I/O scheduling in case of a propensity of uneven load distribution (*C3*). A JBOF holds a much larger data storage capacity than an embedded node. Given the same key range, a JBOF cluster requires fewer nodes, where each node would observe a much higher maximum load than the embedded case. This can be corroborated by the *balls into bins* problem. Theoretical analyses [35, 78] show that the maximum load per-server will be $m/n + \Theta(\sqrt{mlogn/n})$ with high probability when $m > nlogn$, where m and n refer to the client request rate and the number of storage nodes, respectively. A smaller node quantity will cause a larger maximum load (Table 1). FAWN effectively balances traffic based on key access randomness and fast node adaption. But for server/SmartNIC JBOFs, one would easily experience system overloading, I/O bandwidth over-subscription, and tail latency increase. Researchers have developed a series of overloading control techniques [44, 76, 93]. What makes the SmartNIC JBOF unique and difficult is its limited computing resources can hardly tolerate ephemeral request bursts. This requires us to design a streamlined and controlled request execution path either within or across a JBOF. The problem is further exacerbated in that NVMe drives present a considerable discrepancy in bandwidth between reads and writes.

3 Design and Implementation

This section presents LEED and discusses how we address the above challenges. Our system design goals are:

Challenges	Level	Design Principles	Proposed Techniques
C1	SSD	P1	A data store w/ circular log and hybrid index (§3.2, §3.3)
C2	Intra-JBOF	P2	A streamlined I/O executor (§3.4)
	Inter-JBOF	P2	A fctl-based req. scheduler (§3.5)
C3	Intra-JBOF	P1	A data swapping mechanism (§3.6)
	Inter-JBOF	P2	A CRRS replication protocol (§3.7)

Table 2: A summary of proposed techniques in LEED.

- **Maximize NVMe storage utilization.** A SmartNIC JBOF embodies similar storage density as a server JBOF. To demonstrate its potential for holding key-value objects, LEED should fully use the device capacity with little waste;
- **High performance under arbitrary access distribution.** Storage I/O stall due to skewed load distributions is the culprit causing latency increases. Tail latency has become a pivotal metric for evaluating data-intensive workloads [48, 51, 65]. LEED should maximize the I/O throughput of a SmartNIC JBOF and ameliorate both average and tail latencies;
- **High availability.** The SmartNIC JBOF has a standalone power supply. Upon failures, due to the large storage density, much more keys are affected than in the FAWN scenario. We design LEED to provide small downtime and little performance interference during the cluster change.

3.1 Key Ideas and System Overview

3.1.1 Key Ideas. SmartNIC core is the most scarce computing resource, followed by memory and I/O bandwidth. Given such inherent unbalanced nature, when building LEED over an array of SmartNIC JBOFs, one should ensure a key-value request can be served immediately whenever there are available core cycles in the cluster. A fundamental challenge is that application demands and key access distributions are unpredictable. Hence, we must achieve this goal in a combined reactive and proactive manner. This yields our two design principles. First, we trade excessive storage I/O bandwidth for other SmartNIC computing resources, e.g., core cycles and memory capacity (***P1***). Such a reactive approach would sometimes hurt the per-request latency since the system issues more I/Os, but it leads to increased overall throughput. Second, we strive to make scheduling decisions as early as possible along the data path to streamline the request execution flow (***P2***). This proactive regulation tries to distribute the right amount of request loads to each JBOF and its SmartNIC cores, avoiding unnecessary I/O stalls at any hardware entities.

3.1.2 LEED Architecture. Figure 2-a depicts a LEED cluster, consisting of clients and back-end SmartNIC JBOFs, connected via ToR Ethernet switches. Clients issue requests to back-end storage nodes via a co-located front-end library. A SmartNIC JBOF (Figure 2-c) holds several NVMe SSDs, where each has multiple partitions or virtual nodes (similar to FAWN [34]). A (virtual) storage node runs a LEED data store and is responsible for a specific key range. LEED divides the whole key space into different partitions and uses consistent hashing to decide the subspace-node mapping.

Table 2 summarizes our proposed core techniques and associated challenges. LEED uses the RDMA networking stack. Within a JBOF, the SmartNIC core (Figure 2-b) schedules requests among NVMe drives based on their bandwidth availability; redirects overloaded writes to an unloaded SSD. Cross JBOFs, LEED applies an

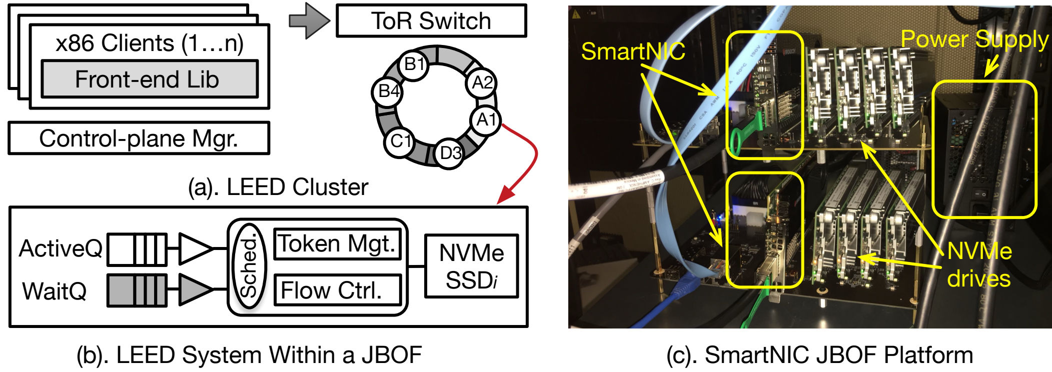


Figure 2: An Overview of LEED system architecture.

end-to-end flow control to regulate incoming traffic loads; handles imbalanced reads using an optimized chain replication protocol. In tandem, these techniques aim to fully reap available computing cycles in the cluster to serve key-value requests, maximize bandwidth utilization, and elude I/O blocking due to over-subscription. Our control-plane manager is backed by the etcd service [4], running over a quorum group of nodes. It maintains the mapping between data store partitions and virtual nodes, monitors the SmartNIC JBOF health and detects failures via periodic heartbeat messages, performs membership management when a node joins/leaves, and configures the replication factor based on workload requirements.

3.2 LEED Data Store

We first consider designing a compacted in-memory index scheme to tackle the skewed storage hierarchy. However, as discussed before (§2.3), for 256B or even smaller key-value objects, addressing the full SmartNIC JBOF flash space under its DRAM capacity constraint indicates that one could use at most 1/4 byte in-memory index to locate a KV pair, significantly exceeding the capability of today's memory-only indexing approaches (e.g., FAWN [34] taking 6 bytes, SkimpyStash [49] requiring 1 byte in the best case, SILT [66] using 0.7 bytes). Worse yet, the size ratio between an NVMe drive and the SmartNIC SoC memory would increase exponentially as QLC/PLC devices become available [26, 30]. Therefore, we propose a new key-value store that uses a specialized data structure based on a DRAM-Flash hybrid indexing scheme. Our design, driven by the first principle (***P1***), sacrifices storage I/O bandwidth, meaning that a key-value request would require 2+ NVMe accesses: one for retrieving metadata index and one for reading/writing data objects.

3.2.1 Data Structure. Our data store revolves around a key data structure–**circular buffer or circular log**. It is a fixed-sized continuous space on the SSD, where its head/tail points to the beginning/end of the used region. A circular log supports three types of operations: *read* from a specified offset within the valid range; *append* to the end of the log (which increments the tail pointer); *compact*, reclaiming fragmented and outdated data entries and generating space to serve future writes. As in-place updates are inefficient on an SSD, overwritten data entries are appended to the log tail whose previous value becomes invalid. Compaction is a central operation that ensures the SSD capacity is fully utilized. It is a batched operation and is triggered when the gap between the tail and head has reached a threshold. We choose this data structure because (1) it leverages the fast random read and sequential write characteristics of NVMe drives [7, 14, 15]; (2) it consumes fewer CPU cycles on reads/writes, unlike the sorting or synchronization phase in an LSM-based or B tree-based implementation.

3.2.2 Data Layout. We separate the keys and values and use different circular logs to store them (Figure 3-a). Similar ideas have been explored in the WiscKey [70] and TerarkDB [19]. In addition to reducing the write implication, this also minimizes the memory footprint and saves the NVMe bandwidth during the item lookup phase. We co-locate keys and their indexes and place them into a circular *key log*. The whole key space of a (virtual) node consists of multiple segments, where each includes up to M chained overflow buckets. A bucket then contains N keys as well as metadata fields (described below), whose size is limited to the SSD block size (e.g., 512B/4KB). Each bucket is appended to the key log. The data structure of a segment is changed to an array of buckets when writing to the SSD. Note that each segment only belongs to one data store and can be accessed by a single core. We simply use one lock bit in the segment table for concurrency control. Values are directly appended to a circular *value log*. Hence, LEED divides one SSD partition into a key and a value log and persistently saves their partition addresses.

3.2.3 Metadata and In-memory Index There are three types of metadata in our KV store. First, a segment index contains K-bits to indicate the chain length and 4B offset pointing to the location of the key log. Second, a bucket consists of a 4B bucket index for key hash matching, K-bits to capture the chain length, K-bits to indicate the position of this bucket within the chain (segment), 4B head/tail fields (used for recovery), and a sequence of key items. Third, a key item encompasses four fields: key, key length, value length, and value offset (pointing to the value log). We only place segment indexes in the SmartNIC DRAM and organize them via a hashtable (called **SegTbl**). This allows us to drastically support large NVMe capacity with little memory usage.

3.3 Command Processing

Our data store supports three basic operations (Figure 3-b). A *GET* firstly looks up the SegTbl (in SmartNIC's DRAM) based on the key hash to locate the data segment position, fetches the segment from the key log (SSD) to memory, traverses the chain to find the key item, and reads the value based on the offset from the value log (SSD). A *PUT* also fetches the data segment via the SegTbl and locates the

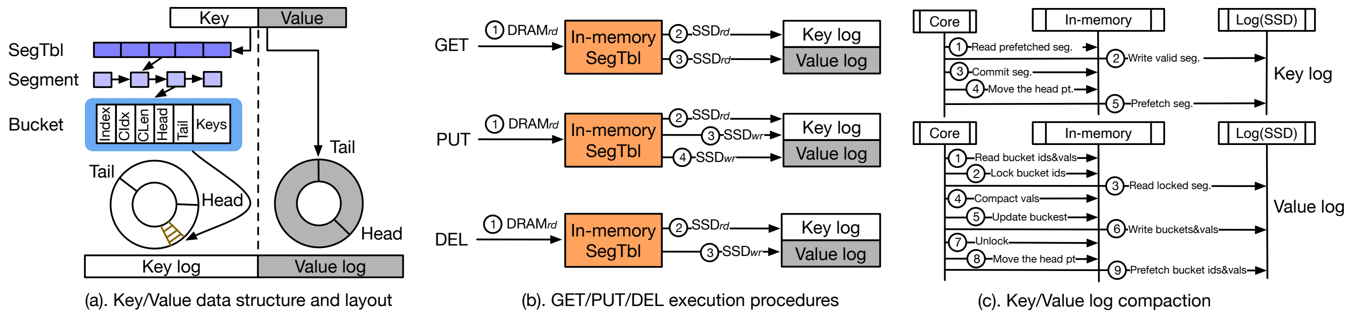


Figure 3: LEED data store architecture and GET/PUT/DEL/Compact operation procedure.

bucket for insertion. It then appends the updated bucket and value to the key log and value log, respectively. PUTs move the tail pointer of both circular logs and no in-place updates happen. We also parallelize value log write and key log operations for latency saving. A *DEL* performs similarly as a PUT but only manipulates the key log, updating the corresponding value length field to zero as a deletion marker. Overall, the GET/PUT/DEL triggers 2/3/2 NVMe accesses. Since NVMe reads/writes take tens of microseconds, instead of busy polling the I/O completion word through PCIe that wastes core cycles (principle ***P1***), we develop an event-based asynchronous execution framework. Each store maintains a waiting event queue, where an event describes the segment index command, as well as execution status for an operation. We then use a per-command state machine to track their running phases.

3.3.1 Compaction and Optimization. LEED reclaims outdated entries using compactions. The operation is heavyweight, not only consuming computing and storage I/O bandwidth but also stalling the ongoing request execution when it manipulates the same bucket. PUTs would be served slowly if the new log entry generation speed cannot catch up with the PUT rate. LEED introduces two optimizations: (1) prefetching future segments. When executing the Nth compaction, LEED prefetches compacted segments for the $N + 1$th one such that it can obviate loading data from an SSD; (2) dividing one compaction into S sub-compactions for parallel execution and co-scheduling them concurrently, where each sub-compaction can prefetch data logs concurrently.

Figure 3-c depicts the compaction process. For the key log, after task splitting, sub-compactions start concurrently in the following steps: reading and verifying prefetched segments, appending valid buckets to the key log, updating the offsets of the segments that are still valid in SegTbl, and moving the head pointer of the key log. Prefetching happens in the background. The value log compaction is more complicated since it might change the value offset and update the key log. Similarly, multiple parallel sub-compactions work as follows. First, it reads the prefetched bucket IDs (which are stored in a small dedicated circular log), prefetches values, and locks related segments in SegTbl. Second, it fetches the buckets from the key log, collects all valid values, and updates their corresponding buckets. Third, it writes the compacted values and updated buckets to their corresponding logs, unlocks all the buckets, and moves the head pointer of the value log. Our log structure ensures that the old value is still valid before committing. The key log compaction process would skip the segment when it is locked by PUT/DEL/value log compaction operations. Value log compactions use locks to ensure consistency.

3.4 Intra-JBOF I/O Execution

When a request enters a JBOF, the SmartNIC core fetches packets from the RDMA stack and delivers them to the data store engine. There are two general design strategies: one is employing a dedicated dispatcher that buffers incoming requests and distributes them in a load-aware fashion, which would waste SmartNIC cores and incur head-of-line blocking; the other one is using a designated load-agnostic processing pipeline and compensating it with either admission control or work stealing/offloading mechanisms. LEED takes the latter one (following the principle ***P2***) because it operates in a DRAM constraint (or low memory footprint) mode and tries to maximize core utilization for key-value processing. We discuss how to integrate load awareness in §3.5.

Our intra-JBOF I/O execution engine works as follows. First, we partition the computing resources and create a static mapping between cores and I/O devices to avoid execution contention. For example, our prototype uses cores 0–3 to handle NVMe drives 0–3 and cores 4–6 to poll the 100GbE RDMA receiving queue, respectively. The rest unallocated one (core 7) is mainly responsible for control-plane tasks instead of data-plane ones. Our design centers around NVMe I/O processing and ensures that a storage I/O can be served immediately when a core is available. Second, we use a lockless concurrent queue everywhere in the system (e.g., the NIC/SSD ring buffer) for inter-core communication. The basic queueing discipline is first-come first-serve (FCFS).

Third, since the SSD available bandwidth and per-storage IO execution cost is varied unpredictably (based on SSD internal conditions and workload profiles) [33, 42, 45, 52, 73], we need an adaptive mechanism to limit the number of I/Os issued to an SSD to elude I/O stalls. LEED equips each SSD partition (Figure 2-b) with (1) an *active queue*, maintaining ongoing data store commands that wait for completion signals; (2) a *waiting queue*, storing runnable requests received from clients. The size of an active queue represents the SSD's current I/O serving capability. We then translate its queue capacity to N tokens using the measured per-IO latency following the prior work [61, 73, 82]. The token quantity of command, representing its execution cost, is empirically decided offline. When a request is retired from the active queue, the scheduler picks the next command for execution if its token requirement is satisfied. The waiting queue captures the SSD overloading status. Both queues

Algorithm 1 Load-aware scheduling based on flow-control.

```
1:  procedure REQ_SCHED()
2:      while true do
3:          for T in AllTenants do
4:              req = T.req_queue.dequeue()
5:              if req.token ≤ MappedSSDs(req.target).tokens then
6:                  MappedSSDs(req.target).tokens -= req.token
7:                  rpc_send(req); Break
8:              else
9:                  if OutReqs(req.target) ≥ 1 then
10:                     T.req_queue.enqueue(req)
11:                 else
12:                     MappedSSDs(req.target).tokens = 0
13:                     rpc_send(req); Break
14:         if timeout is true then
15:             Break
```

are shallow, whose size hinges on the SSD queue depth and I/O read/write rate.

3.5 Inter-JBOF Scheduler based on Flow Control

Our intra-JBOF I/O engine alone is inadequate to address system overloading. Ideally, a key-value request should be issued from a client only if its target JBOF has a serving capacity. Otherwise, one would observe tail latency increases. Driven by our design principle (***P2***), we aim to develop such a scheduler at the cluster level that can make a judicious decision before submitting requests to a SmartNIC JBOF. LEED envisions this via an end-to-end flow control mechanism that informs clients of the data store's serving availability.

Our design is inspired by prior work on system overloading control [44, 87, 92]. Each back-end SSD allocates available tokens based on its waiting queue among co-located tenants in a weighted fashion and distributes them via a piggyback response. LEED then builds a load-aware scheduler using this information. For an upcoming scheduling round, the front-end traverses each active client in a round-robin manner, picks the next request, and submits it only if (1) its target SSD offers enough tokens (Alg1 L5–7); or (2) there are no outstanding commands being issued (Alg1 L9–13). This behaves similarly to the Nagle's algorithm [10, 74, 75]. We update the token amount after submitting a request successfully, or when receiving a response that piggybacks the allocated available token amount from the target SSD.

Our cross-node communications use RPCs [53, 58, 59] and exploit a hybrid use of one-sided/two-sided RDMA verbs. Akin to the NVMe-over-RDMA implementation, the sender takes advantage of a two-sided RDMA SEND verb to pass the command, while the receiver uses the one-sided RDMA WRITE because we pre-allocate the response memory when preparing the request. This streamlines the I/O processing and enables efficient memory management at the sender. Further, we use the 32-bit immediate (IMM) field to directly write into the remote node's completion queue to identify the corresponding request, reducing RDMA WRITEs messages.

3.6 Intra-JBOF Write Imbalance Handling

A SmartNIC JBOF holds multiple NVMe drives, where each has one or several partitions. When one partition receives more requests than its serving capacity, its waiting queue would become full (§3.4), where no available tokens are generated and backpropagated, finally stalling clients. However, given the uneven load distribution (§2.3), other SSDs either within or across the JBOF, might still have available bandwidth. Motivated by this, LEED introduces techniques to outsource requests to other available SSDs at intra-/inter-JBOF levels. For write imbalance, we develop a data swap mechanism to temporarily redirect PUT requests to other co-located SSDs. This technique follows our first design principle (***P1***), trading storage I/O bandwidth for computing cycles. Note that read imbalance can hardly be handled within a JBOF because the key-value pair location has been decided.

The SmartNIC core continuously detects if its mapped NVMe drive is over-subscribed. Upon overloading, LEED redirects overloaded PUT requests to other available SSDs that are not the home of the current data store. One can simply view it as if we create a swapping data region from another SSD to absorb burst writes. The swapping region will then be merged back during future compactions (when the home SSD has available bandwidth). Thanks to our flexible circular log data structure, LEED only makes two unobtrusive changes to enable this technique. First, it extends the metadata field of each log entry with an SSD identifier so that one can locate the correct key log position. Second, it makes the scheduler route requests from one SSD's waiting queue to another one's active queue (when the queueing occupancy gap is larger than a threshold). All subsequent GETs to this region will then be redirected before swapping back. When there are multiple temporary store candidates, LEED chooses the one with the most available bandwidth.

3.7 Cross-JBOF Read Imbalance Handling: CRRS

Similar to prior works [34, 56, 60], LEED uses the chain replication [80] to achieve per-key strong consistency. PUTs/DELs are sent to the head of a chain and then propagated to each replica along with its successors in the ring space. GETs are directly served by the tail. Assuming a replication factor R, each (virtual) node is part of R different chains: it is the "head" for one chain, a "mid" node in $R-2$ chains, and the "tail" for one. Write imbalance cannot be addressed because all replicas would receive the request. But read imbalance is possible if we enable more replicas to serve GETs.

Inspired by prior efforts [83] and following our second design principle (***P2***), we propose an optimized chain replication scheme that tailors to our setting, called CRRS (*Chain Replication with Request Shipping*). Our goal is to allow more replicas to serve GETs without hurting the consistency guarantee. LEED augments every data store with a hash map to mark all dirty keys. PUTs/DELs are performed in the same way as before with slightly more operations on the attribute (Figure 4). Upon receiving a new object update, if a node is not the tail, it will set the dirty bit and forward the request to the next replica as the default protocol. The tail then clears the dirty bit (commitment point), executes the command, replies back to the front-end, and sends an acknowledgment backward through the chain to notify data commitment. A replica will then unset the dirty bit when receiving the acknowledgment.

GET processing differs from the original chain replication in that requests can be served by more replicas (not only the tail). This is because: when receiving a GET operation, a replica can use the dirty bit to decide whether it has the latest data copy. If its dirty bit is clear,

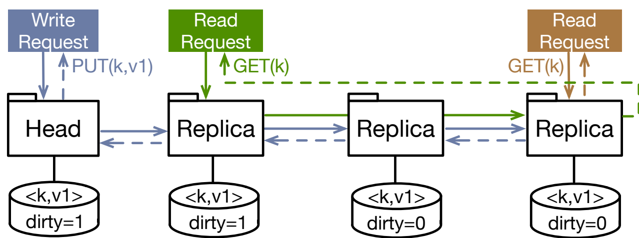


Figure 4: Request execution under CRRS. Writes (blue) are propagated across the chain. Reads to the tail (brown) are clean. Reads to a replica with a dirty bit (green) are shipped. Dotted lines are acknowledgments or client responses.

the replica executes the command and sends back the fetched object. Otherwise, LEED will ship this request directly to the tail node since it always commits the latest write and holds the latest value. LEED does not violate the consistency model because all operations are serialized by the tail during this read/write interleaving phase. Another design option is to ask the intermediate node to issue a version query message (similar to CRAQ [83]) to implicitly serialize command processing. We find that this approach generates more internal traffic across JBOFs and perturbs the traffic pattern. With this, LEED allows more replicas to handle reads and chooses the target data store with the maximum amount of available tokens, mitigating the read imbalance issue. CRRS complicates our failure handling procedure and we will discuss it next (§3.8).

3.8 Node Join, Leave, and Failure Handling

LEED handles cluster changes and unplanned failures. To facilitate node join/leave, we propose a new primitive called *COPY* using *PUT* and *GET* operations. A *COPY*, interacting with DRAM/SSDs, locks the source data segments in the SegTbl, reads corresponding key-value items from key/value logs, and then copies them to the destination SSD via *PUT*s. The lock is released when the process is finished. The *COPY* is mutually exclusive with *PUT*s/*DEL*s, and thereby, copied key-value pairs are immutable during its execution. Further, we introduce three node states (i.e., *JOINING*, *RUNNING*, *LEAVING)* to capture different execution phases.

3.8.1 Node Join and Leave. A JBOF has multiple virtual nodes with different key ranges. When a JBOF joins, each virtual node splits the key range of a chosen partition into two and divides the data store by inserting a new virtual node. It receives a new virtual ID whose state is set to *JOINING* and takes the selected key range. Upon receiving membership updates from the control plane, tails of *R* hash chains start to issue *COPY* to the newly joining virtual node and move the stipulated data range. During this phase, incoming *PUT*s and *DEL*s might be forwarded to the new virtual node depending on if their keys are copied. After all the tails finish copying and notify the control plane, the newly joined node changes to *RUNNING*. Finally, clients can start issuing requests to this node. All the tails stop forwarding requests and free the data ranges that they are no longer responsible for.

Since all the nodes in the system receive membership updates from the control plane asynchronously, a request may traverse the hash chain with different views. For instance, a *PUT*/*DEL* might skip the newly joined node, while a GET would read from the former tail that contains outdated values. Thus, to ensure data consistency, an operation can be performed if and only if all the nodes participating in the operation have the same view. We achieve this by adding a hop counter in a request, which increases by one when the request is being forwarded. Upon receiving a request, a node can fetch a local copy from its hash ring first, and then compare the hop counter to verify if the request has entered the correct position along the chain. If a violation happens, a NACK would be issued back to the client for retrying.

Node leave works similarly to the joining process except that tail nodes are responsible for copying data ranges to the next node down the hash ring. When a (virtual) node leaves on demand or involuntarily, the state of the virtual node in the control-plane manager changes to *LEAVING* after a heartbeat timeout. The number of replicas of this virtual node would become *R-1* temporarily and change back to *R* after the leaving process completes. Clients stop issuing requests to this virtual node immediately upon receiving the state update. The tails begin to copy to the next node. After the data copy is completed from all tails, the virtual node is permanently deleted from the control plane.

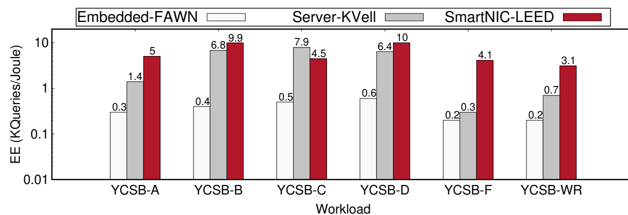
3.8.2 Failure Handling. LEED assumes that each node operates under the fail-stop mode and node-node communications use a reliable transport protocol. We use the heartbeat message for failure detection and trigger a node leave if a JBOF is inactive for a certain period. Next, we discuss how a failed node interacts with our CRRS chain replication.

- **Failed head:** Reads are still served by the rest of a chain. For ongoing writes, the dirty bit of associated objects is set. As these requests are finally committed at the tail, the dirty bit will be cleaned when receiving the response. The former second node along the chain keeps sending responses to the failed head until it becomes the head and starts to serve requests. During the joining process, this node will discard all uncommitted changes and data, and obtain the latest and consistent copy from the tail node;
- **Failed mid-node:** Writes cannot be propagated until the control plane updates the neighbor. After that, dirty bits are cleared by the tail. Similarly, the failed mid-node will receive a data copy during rejoining. Reads are unaffected;
- **Failed tail:** For committed writes, dirty bits are cleared for the rest of a chain. Reads to these objects are handled by other replicas. However, for uncommitted writes, an unsuccessful acknowledgment is back-propagated to clients, rolling back to the previous old value and clearing the dirty bit simultaneously. Reads will return the old value. A subtle scenario–the write commitment is invisible to other nodes–would happen, e.g., due to an unexpected power failure. In this case, the penultimate node of a chain will keep the dirty bit until it becomes the tail, which then commits the write and propagates the response.

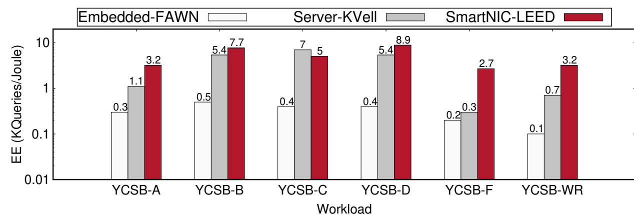
4 Evaluation

4.1 Experimental Methodology

Testbed setup. Our testbed comprises an RDMA-capable rack with x86 servers and Stingray PS1100R storage nodes, connected to a 100Gbps Arista 716032-CQ switch. The **Server JBOF** has two Intel Xeon Gold 5218 processors, 96GB memory, and a 100Gbps dual-port Mellanox ConnectX-5 NIC. The **SmartNIC JBOF** configuration is



(a) 256B Key-Value object



(b) 1KB Key-Value Object

Figure 5: Energy efficiency (Queries per Joule) comparison among three platforms: FAWN over embedded storage nodes, KVell over server JBOFs, and LEED over SmartNIC JBOFs under 6 YCSB workloads. y-axis is log scale.

described in §2.1. We equip both with 4–8 Samsung DCT983 960GB NVMe SSDs. We run Ubuntu 20.04 on Intel machines and Ubuntu 16.04 on Stingray nodes. The **embedded node** is a Raspberry Pi 3 Model B+, consisting of one 1.4GHz 64-bit quad-core ARM Cortex A53 processor, 1GB DRAM, 1 Gigabit Ethernet over USB 2.0, and a 32GB Sandisk drive. Its idle power is 3.6W. Our emulated FAWN cluster uses a Dell 1GbE switch and a PowerEdge R7525 as clients. We use a *Watts Up Pro* meter and a *HOBO Plug Load Data Logger* to measure the power of Server/SmartNIC JBOF and FAWN cluster, respectively.

Implementation details. We develop LEED based on the SPDK reactor framework [17] (version 19.10). Polling on the SmartNIC JBOF incurs only a little power consumption. For example, we observed 7.5W higher power with eight polled cores than the idle case (45W). Each SSD could hold 32 virtual partitions. We empirically determine the number of tokens for GET/PUT/DEL. The ETCD (used by our control-plane) version is 3.5. We have released the source code of LEED on our GitHub repository (https://github.com/netlab-wisconsin/LEED).

Comparison schemes and workloads. We compare LEED with two other platforms: (1) server JBOFs that run a recently developed fast persistent key-value store (i.e., KVell [62]) as the backend; (2) embedded storage nodes that run the FAWN [34] storage stack. Instead of using the deprecated PCEngine Alix board, we choose Raspberry PI nodes with similar power consumption but better performance. Our workload generator is based on the YCSB [46]. Similar to prior studies [34, 36, 37, 49, 88], we mainly consider small key-value objects (i.e., 256B and 1KB). Our experiments use both random and Zipf [22] distributions with different skewness factors. Each data store holds around 1.6 billion objects during the test. We disable front-end caching and focus on measuring the persistent key-value processing performance.

4.2 LEED v.s. Other Data Stores on SmartNIC JBOF

We ported FAWN [34] and KVell [62] which were originally designed for embedded and server storage nodes onto a SmartNIC JBOF and compared with LEED (Table 3). We use a hybrid DRAM/Flash indexing scheme to address the skewed storage hierarchy and can support nearly the entire 4×960GB NVMe space. It incurs some storage overheads due to key/value logs (less than 5%). Regarding the other two, FAWN uses a log-structured design and requires 6 bytes in-memory hash index for each object. Given the limited DRAM capacity, it can only use 7.7%/24.1% of the whole capacity for 256B/1KB objects. KVell requires an in-memory B-tree index,

	FAWN-JBOF		KVell-JBOF		LEED	
	1KB	**256B**	**1KB**	**256B**	**1KB**	**256B**
Max. Capacity	24.1%	7.7%	2.6%	0.9%	97.3%	95.4%
RND RD Lat. (us)	54.0	65.4	445.0	416.0	133.1	116.5
RND WR Lat. (us)	44.8	61.4	810.0	764.0	84.0	83.9
RND RD Thr. (KQPS)	74.0	61.2	289.1	299.9	855.9	860.0
RND WR Thr. (KQPS)	88.4	64.8	156.1	160.7	608.6	576.7

Table 3: Single-node performance comparison among FAWN-JBOF, KVell-JBOF, and LEED. FAWN/KVell-JBOF is measured under their maximum capacity. RND=random.

partial in-memory free lists, and a page cache, yielding even less used space, i.e., 33GB/100GB for 256B/1KB objects.

Regarding latency, FAWN-JBOF performs the best as it only requires one SSD access per request. LEED needs 2+ NVMe accesses to process each command (§3.3), whose read/write latencies are nearly double the ones of FAWN-JBOF, i.e., $116.5\mu/83.9\mu$ and $133.1\mu/84.0\mu$ for 256B and 1KB object, respectively. We further break down the latency of each individual command to demonstrate this (Figure 11 in Appendix). Even though PUT issues one more SSD access compared with the other two, LEED overlaps the first two accesses and only adds $10.5\mu s$ more latencies. KVell-JBOF performs the worse, even though it also needs 1 SSD access. For example, its 1KB/256B random read takes $445\mu s/416\mu s$, 3.3X/3.6X slower than the LEED case. This is mainly because its B-tree indexing is computation-heavy and its performance is limited by the SmartNIC processor.

In terms of throughput, LEED outperforms the other two significantly (which also yields the best energy efficiency in terms of requests per Joule) because it supports the entire flash capacity, harnessing all the I/O parallelism of the PCIe lanes and NVMe internals. For 256B random read/write, LEED achieves 2.9X/3.6X and 14.1X/8.9X higher throughput than KVell-JBOF and FAWN-JBOF, respectively. The 1KB case is similar.

We further look into the FAWN-JBOF and find out that its log compaction process can be further improved. In particular, LEED applies execution parallelism by dividing one compaction into multiple independent sub-compactions and running them in parallel (i.e., intra-parallelism). On average across three workloads, we observe 1.9× throughput improvement when using 8 threads (Figure 13a in Appendix). More threads are helpful when invalid entries are distributed non-uniformly since it uses more core cycles for chaining. Similarly, co-scheduling multiple compactions (i.e., inter-parallelism) also improves the throughput by 17.9% (Figure 13b in Appendix).

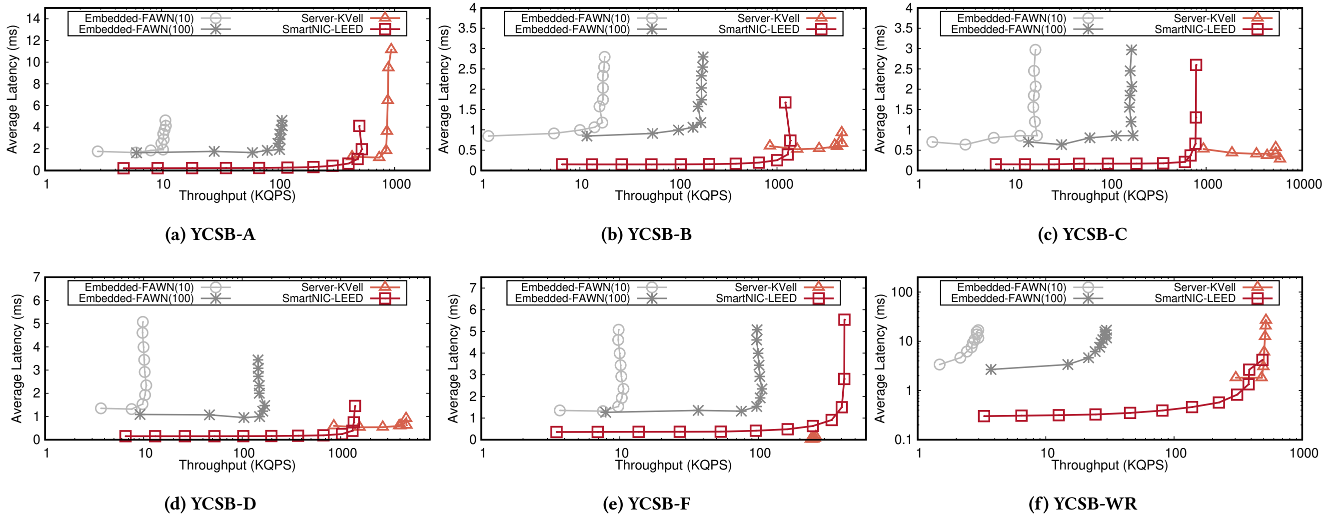


Figure 6: Latency v.s. throughput for 6 YCSB workloads under the 1KB case. FAWN(10) and FAWN(100) represent the number of nodes in the cluster is 10 and 100, respectively. X-axis is log scale for all scenarios. The y-axis of YCSB-WR is log scale.

4.3 Power and Energy Efficiency: Three Platforms

We compare the power and energy efficiency of three persistent key-value systems: FAWN over embedded storage nodes (Embedded-FAWN), KVell over server JBOFs (Server-KVell), and LEED over SmartNIC JBOFs (SmartNIC-LEED). The replication factor of each data store is 3. We run the YCSB workloads with the default skewness. When maxing out their storage utilization, on average, Embedded-FAWN, Server-KVell, and SmartNIC JBOFs consume 42.0W, 756.0W, and 157.5W, respectively. This translates to 0.13W, 0.065W, and 0.014W per Gigabyte of key-value pairs across these three platforms. The Raspberry Pi cluster has the least power draw, but the system is unable to hold large-capacity NVMe drives. Server JBOFs consume the most power, while each socket can support up to 48 PCIe Gen3 lanes, leading to high-density storage nodes. SmartNIC JBOFs instead combine the benefits of these two.

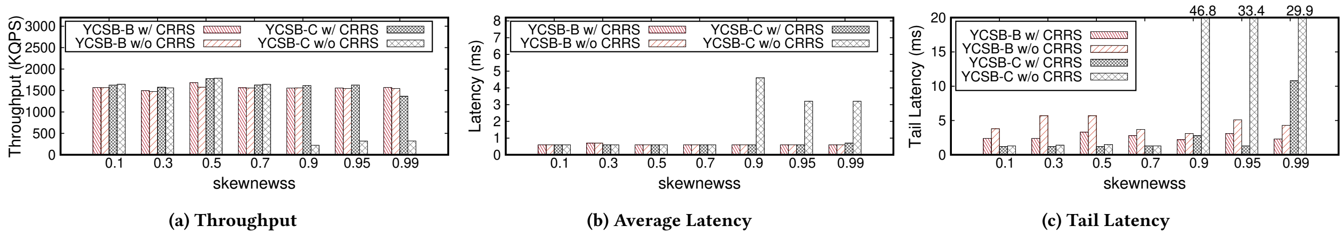
In terms of energy efficiency, on average across 6 workloads (Figure 5), SmartNIC-LEED outperforms Server-KVell by 4.2×/3.8× for the 256B/1KB cases. This corroborates our motivation (Figure 1). LEED can drive most of the NVMe bandwidth using constrained CPU and memory. It further achieves 17.5×/19.1× higher energy efficiency for 256B/1KB objects than the Embedded-FAWN as a Pi node is bounded by the I/O throughput of its SanDisk. There are some high-end embedded boards (such as HiKey970 [29]) that offer slightly improved I/O performance (UFS 2.1 standard, 2-3× better bandwidth than Pi), but still much worse than SmartNIC JBOFs. We observe that Server-KVell achieves higher (e.g., 7 v.s. 5 KQueries/Joule for 1KB object) energy efficiency than the SmartNIC-LEED case when running YCSB-C (read-only). This is mainly because KVell achieves higher throughput than LEED by employing a parallelized shared-nothing I/O and an in-memory sorted indexing design, which operates best under the read-only mode. Instead, each read in LEED requires two NVMe accesses to read the key/value log.

4.4 Latency and Throughput: Three Platforms

We measure the latency-throughput behavior by varying the client request issuing rate for six workloads. In addition to the setups in §4.3, we also consider an artificial 100-node FAWN cluster to match the networking bandwidth as the JBOF ones. We assume an ideal linear scaling case where the throughput of FAWN(100) achieves 10× IOPS as FAWN(10), without latency increases. As shown in Figure 6, in the case of 1KB, Server-KVell achieves the best throughput, outperforming SmartNIC-LEED and Embedded-FAWN(100) by 2.9× and 22.2× on average across six workloads, respectively. When approaching the maximum throughput, SmartNIC-LEED delivers lower average latencies. For example, compared with Embedded-FAWN(100) and Server-KVell, it reduces their average latencies by 47.9% and 28.5% on average across all workloads, respectively. This is mainly due to the intra-/inter-JBOF techniques that tackle the unbalanced loads and control the overloading. LEED uses a data swapping mechanism and an optimized replication scheme to improve the read/write throughput. The end-to-end storage flow control allows LEED to choose a less loaded replica and throttles the client when obtaining no tokens, yielding latency savings. The 256B case performs similarly (Figure 14 in Appendix).

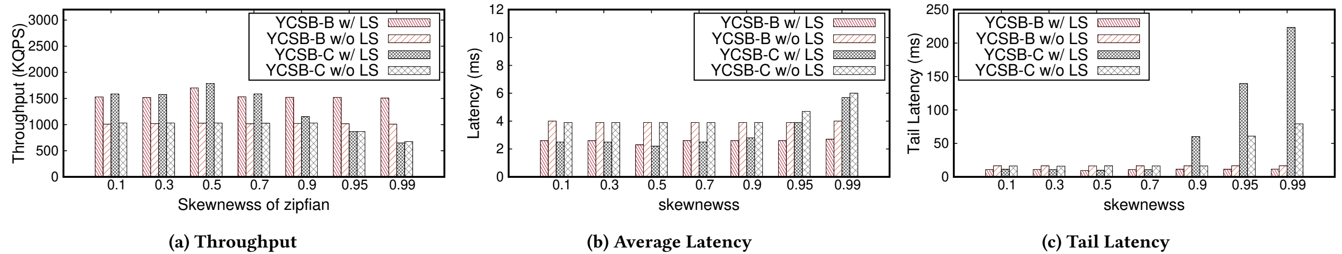
4.5 LEED Achieves Load-aware Scheduling

LEED schedules key-value requests in a load-aware fashion via a token-based streamlined intra-JBOF I/O engine and a flow control-based inter-JBOF scheduler. It exposes the serving capability of each SSD partition and propagates it to the front-end. A request can only be issued and executed when the target SSD provides enough tokens. In the case of YCSB-B (Figure 8), it improves 52.2% throughput, and reduces average/99.9th latencies by 34.4%/33.7%. We observe similar benefits for YCSB-C. However, LEED requires at least one round-trip to backpropagate the token information to the client. When a severe incast scenario happens (e.g., the YCSB-C case with zipf skewness equals to 0.95/0.99), one would still observe the active and waiting queue quickly built up, jeopardizing both average and tail latencies.



(a) Throughput (b) Average Latency (c) Tail Latency

Figure 7: LEED handles the read imbalance at the inter-JBOF level using the CRRS replication. We compare the application performance between w/ and w/o CRRS support. We choose YCSB-B and YCSB-C, and vary the Zipf skewness.



(a) Throughput (b) Average Latency (c) Tail Latency

Figure 8: LEED couples the intra-JBOF and inter-JBOF techniques to achieve load-aware scheduling. We compare the application throughput and latency between enabled and disabled scenarios. We choose YCSB-B and YCSB-C, and vary the Zipf skewness.

4.6 Imbalanced Load Handling in LEED

Write Imbalance. LEED tackles the issue within a JBOF via the data swapping mechanism. It chooses an unloaded SSD to absorb overloaded writes. The higher the skewness is, the more benefits one would observe (Figure 10). Under 0.99 skewness, our mechanism yields 15.4%/17.2% higher throughput for the 256B/1KB cases. Data sapping also ameliorates tail latency because requests could be served by another SSD without waiting in the queue. On average across all skewed scenarios, we obtain 28.6%/32.1% avg./99.9th latency savings.

Read Imbalance. LEED handles the read imbalance at the cluster level via CRRS. Instead of forwarding requests to the tail, it allows each replica to serve reads when the dirty bit of an object is unset. When the request load is evenly distributed and the replica is not overloaded, CRRS has little effect. However, when the client demand exceeds the serving capability of the tail, with CRRS, one can forward traffic to other replicas and obtain considerable performance improvement. For example, for the YCSB-C (Figure 7), when the Zipf skewness is 0.9, 0.95, 0.99, CRRS helps improve the throughput by 7.3×, 5.1×, and 4.2×, reduce the average/99.9th latencies by 86.6%/94.0%, 80.8%/96.0%, and 76.4%/63.7%, respectively.

4.7 Performance under Node Failures

We set up a 3-nodes LEED cluster (w/ three replicas) and measure the throughput when running YCSB-A/YCSB-B workloads (value size is 1KB). We start the joining/leaving process when the cluster becomes stable. As shown in Figure 9, we observe 49.1%/15.9% throughput drop after join started and 66.0%/43.9% after leave started for the YCSB-A and the YCSB-B, respectively. This is mainly because of the *COPY*-induced extra cost. The leave process experiences more degradation as the node receives *COPY* writes and also serves ongoing requests. Our control-plane manager issues membership updates to all the nodes when a node joined the cluster or started to leave or finished the leaving process. A node might actively reject client requests when they observe inconsistent views, adding up to 29.7% extra degradation within a few seconds at the end of the joining process (i.e., YCSB-B).

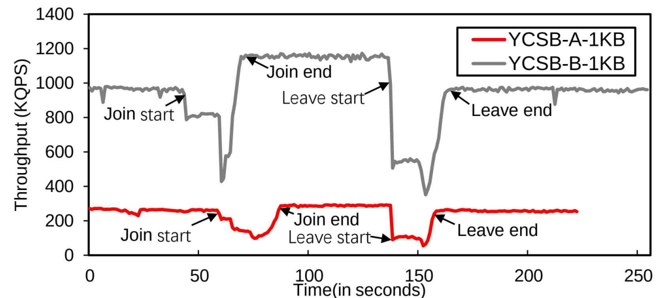
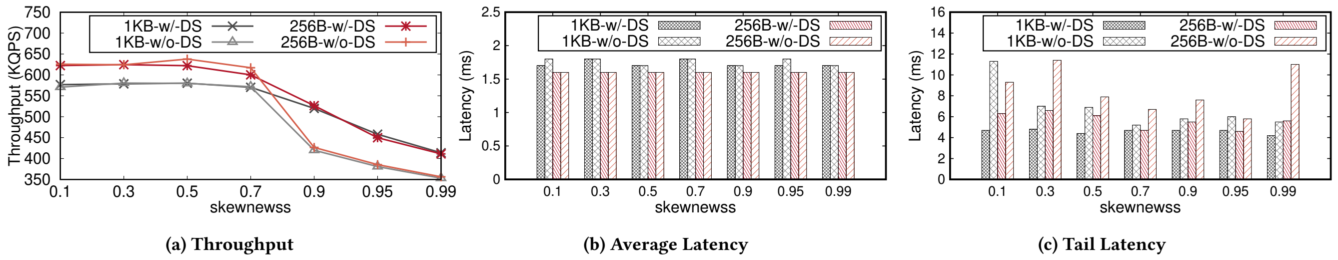


Figure 9: Throughput variation during a node join/leave.

4.8 Discussion

Based on our development experience in building LEED, we'd like to discuss some design trade-offs, limitations of existing SmartNIC JBOFs, and potential improvements for next-generation platforms.

First, LEED is a throughput-oriented system that tries to execute a storage I/O whenever possible. Without sacrificing the NVMe bandwidth, one would observe some CPU stalls (§ 3.6) or memory constraints (§3.2). Second, the earliest scheduling decision can be made when a key-value request is issued from the client. This is useful for our inter-JBOF execution (§3.5), where we employ an end-to-end flow control. But it helps little for the intra-JBOF execution because of the fast-changing environment (e.g., computing availability and storage bandwidth). In this case, we instruct the request scheduling when it arrives at the target JBOF. Third, our Stingray



(a) Throughput

(b) Average Latency

(c) Tail Latency

Figure 10: LEED employs an intra-JBOF data swapping mechanism to handle temporarily imbalanced writes. We consider a write-only workload under Zipf distribution varying the skewness and measure the application throughput and latency.

node encloses 4 Samsung DCT983 SSDs, where each provides up to 400K 4KB IOPS. The data store of LEED consumes at least 2 SSD accesses for each command, translating to up to 200KIOPS per SSD and 800K per JBOF. Our microbenchmarks demonstrate that we can fully use the storage bandwidth. However, to saturate the network bandwidth, especially for small key-value objects, one can either reduce the number of SSD access for each command or increase the platform throughput. Given the limited DRAM capacity, we think the latter would be more practical. Hence, we believe future SmartNIC JBOFs should be equipped with a big PCIe switch (with more lanes) or upgraded to PCIe Gen4. Fourth, in addition to the DRAM capacity, we find that another system limitation is onboard memory bandwidth (which is 4390MB/s on Stingray), which bounds the max number of concurrent operations (such as data segment table manipulation, concurrent queue access) and hurt the throughput. We expect next-generation devices would further increase the bandwidth or even use HBM as the Silicom N5010 card [16]. Finally, we find out that our system still has some available CPU cycles (even running at a modest load), where we can use it for other optimizations. For example, we can further increase the entry size of the segment table to further reduce the in-memory metadata. The trade-off here is that each look-up phase might need more probing cycles. So with better SmartNIC execution engines, one can explore more such system optimizations. In sum, SmartNIC JBOFs deliver extremely high I/O throughput with ultra-low, allowing us to explore a low-power and high-performance, high energy-efficient solutions, where Xeon-based or FAWN-like systems could hardly achieve.

5 Related Work

Energy-efficient computing. Energy efficiency has become a key factor driving the design and implementation of data center systems. Researchers have proposed various specialized hardware architectures and software optimizations. FAWN [34] applies a log-structured based key-value store over an array of winpy nodes. Gordon [40] develops a low-power storage array using embedded processors and programmable SSDs. Lake [84] is a power-efficient key-value store built with NetFPGA and conflates multiple cache layers with processing cores. E3 [68] exploits the use of SmartNICs for energy-efficient executions of microservice-based applications. Andrew et al. [47] harness cheap single-board computers (e.g., Raspberry Pi 3B+) to execute in-memory OLAP workloads for cost and energy savings. People also strive to improve the energy efficiency of an external sort benchmark (i.e., JouleSort [81]) for years [54, 57, 69, 71, 77]. We are exploiting the use of SmartNIC JBOFs for energy-efficient persistent key-value processing.

Key-value caching service. Researchers have built many optimized flash-base cache systems. Honeycomb [67] stores a B-Tree in host memory and executes SCAN and GET on an FPGA-based SmartNIC to improve the throughput. SkimpyStash [49] moves the key-value pointers from the RAM to the Flash and resolves hash table collisions using linear chaining. HiKV [86] builds the persistence hash index in NVM and B+-Tree index in DRAM to avoid long NVM writes for maintaining consistency. Flashshield [55] leverages the host DRAM as a filter to minimize data rewrites., requiring 20 bits per object for indexing. CacheLib [37] reduces the memory footprint for object indexing via a set-associative design. Kangaroo [72] proposes a hybrid caching architecture that combines a small log-structured cache with a large set-associate one. Our work targets a special setting where (1) the DRAM/Flash storage hierarchy is extremely skewed; (2) each I/O has limited computation availability.

Cluster-level overloading control. LEED benefits from prior work on system overload prevention. For example, Huamin Chen et al. [43] conducts capacity planning and request scheduling for web servers by analyzing the dependency among session-based requests. CoDel [76] regulates the queueing delay to elude network overloading. Doorman [23] requires manually configuring the client and server capacity before deployment. DAGOR [93] detects the microservice loading status in realtime and adjusts its load-shedding thresholds adaptively. Breakwater [44] proposes an overload control scheme based on credits and server-side admission control. LEED captures the NVMe bandwidth availability and translates it into tokens.

6 Conclusion

This paper revisits the design and implementation of a low-power high-performance distributed key-value store using emerging SmartNIC JBOFs. We find that conventional system design philosophies and techniques become either invalid or ineffective due to the imbalanced architectural design of a SmartNIC JBOF. We propose new techniques at the SSD/intra-JBOF/inter-JBOF levels to address the unique challenges. Our prototyped system on the Boradcom Stingray demonstrates energy efficiency improvements over prior solutions. This work does not raise any ethical issues.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Brad Campbell, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199 and CNS-2212192. Zerui Guo, Yuebin Bai, and Ming Liu are the corresponding authors.

References

[1] 2021. Broadcom FlexSPARX acceleration subsystem. https://docs.broadcom.com/doc/1211168571391. (2021).

[2] 2021. Broadcom Stingray PS1100R SmartNIC. https://www.broadcom.com/products/storage/ethernet-storage-adapters-ics/ps1100r. (2021).

[3] 2021. Dell PowerEdge R750. https://www.dell.com/en-us/work/shop/productdetailstxn/poweredge-r750. (2021).

[4] 2021. ETCD. https://etcd.io. (2021).

[5] 2021. Fungible FS1600 Box. https://www.fungible.com/product/nvme-over-tcp-fungible-storage-cluster/. (2021).

[6] 2021. HP Nimble Storage. https://www.hpe.com/us/en/storage/nimble.html. (2021).

[7] 2021. Intel Optane SSD 800P Series. https://www.intel.com/content/www/us/en/products/sku/125298/intel-optane-ssd-800p-series-118gb-m-2-80mm-pcie-3-0-3d-xpoint/specifications.html. (2021).

[8] 2021. K200 DPU-based Smart Storage Acceleration Card. https://www.kalrayinc.com/products/k200-lptm-card. (2021).

[9] 2021. Lightbits SuperSSD. https://www.lightbitslabs.com/ty-product-brief-superssd/. (2021).

[10] 2021. Nagle's algorithm. https://en.wikipedia.org/wiki/Nagle%27s_algorithm. (2021).

[11] 2021. NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/. (2021).

[12] 2021. Raspberry Pi 3 Model B+. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/. (2021).

[13] 2021. Raspberry Pi 4 Model B. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. (2021).

[14] 2021. Samsung DCT983 NVMe SSD. https://www.samsung.com/semiconductor/minisite/ssd/product/data-center/983dct/. (2021).

[15] 2021. Samsung PM1733 NVMe SSD. https://samsungsemiconductor-us.com/app/uploads/2021/02/PM1733_U2_Product_Brief.pdf. (2021).

[16] 2021. Silicom N5010 SmartNIC. https://www.silicom-usa.com/pr/fpga-based-cards/100-gigabit-fpga-cards/silicom-fpga-smartnic-n5010_series/. (2021).

[17] 2021. SPDK Event Framework. https://spdk.io/doc/event.html. (2021).

[18] 2021. Supermicro All-Flash NVMe. https://www.supermicro.com/en/products/nvme. (2021).

[19] 2021. Terarkdb. https://bytedance.feishu.cn/docs/doccnZmYFqHBm06BbvYgjsHHcKc. (2021).

[20] 2021. The PC Engines ALIX Series of System Boards. https://www.pcengines.ch/alix.htm. (2021).

[21] 2021. Watts up? PRO. http://www.idlboise.com/sites/default/files/WattsUp_Pro_ES.pdf. (2021).

[22] 2021. Zipfian distribution. https://en.wikipedia.org/wiki/Zipf%27s_law. (2021).

[23] 2022. Doorman: Global Distributed Client Side Rate Limiting. https://github.com/youtube/doorman. (2022).

[24] 2022. LevelDB. https://github.com/google/leveldb. (2022).

[25] 2022. RocksDB. http://rocksdb.org. (2022).

[26] 2023. BarraCuda QLC SSDs. https://www.seagate.com/products/hard-drives/barracuda-qlc-ssd/. (2023).

[27] 2023. Data Centers and Servers. https://www.energy.gov/eere/buildings/data-centers-and-servers. (2023).

[28] 2023. Dell Storage Server Solutions. https://www.dell.com/en-us/dt/storage/powervault-me5.htm?ref=cptl_servers-storage-networking-tile1_cta_primary_storage#tab0=0. (2023).

[29] 2023. HiKey 970 Development Kit. https://www.96boards.org/product/hikey970/. (2023).

[30] 2023. Solidigm Introduces Industry-First PLC NAND for Higher Storage Densities. https://www.tomshardware.com/news/solidigm-plc-nand-ssd. (2023).

[31] 2023. Supermicro Storage Server Solutions. https://www.supermicro.com/en/products/storage. (2023).

[32] 2023. Understanding Data Center Energy Consumption. https://cc-techgroup.com/data-center-energy-consumption. (2023).

[33] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference*.

[34] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*.

[35] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. 1994. Balanced allocations. In *Proceedings of the twenty-sixth annual ACM symposium on theory of computing*.

[36] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[37] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The CacheLib caching engine: Design and experiences at scale. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*.

[38] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*.

[39] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}: Facebook's distributed data store for the social graph. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 49–60.

[40] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. 2009. Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[41] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[42] Feng Chen, David A Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Performance Evaluation Review* 37, 1 (2009), 181–192.

[43] Huamin Chen and P. Mohapatra. 2002. Session-based overload control in QoS-aware Web servers. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 2.

[44] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for us-Scale RPCs with Breakwater. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*.

[45] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. 2009. A survey of flash translation layer. *Journal of Systems Architecture* 55, 5-6 (2009), 332–343.

[46] 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*.

[47] Andrew Crotty, Alex Galakatos, Connor Luckett, and Ugur Cetintemel. 2021. The Case for In-Memory OLAP on "Wimpy" Nodes. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*.

[48] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[49] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-Based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*.

[50] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*.

[51] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's law for tail latency. *Commun. ACM* 61, 8 (2018), 65–72.

[52] Peter Desnoyers. 2012. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*.

[53] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*.

[54] Andreas Ebert. 2013. NTOSort. *sortbenchmark. org* (2013).

[55] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[56] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A Distributed, Searchable Key-Value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*.

[57] Sang-Woo Jun, Shuotao Xu, et al. [n. d.]. Terabyte sort on FPGA-accelerated flash storage. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[58] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[59] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*.

[60] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*.

[61] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[62] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*.

[63] Conglong Li, David G Andersen, Qiang Fu, Sameh Elnikety, and Yuxiong He. 2017. Workload analysis and caching strategies for search advertising systems. In *Proceedings of the 2017 Symposium on Cloud Computing*. 170–180.

[64] Conglong Li, David G Andersen, Qiang Fu, Sameh Elnikety, and Yuxiong He. 2018. Better caching in search advertising systems with rapid refresh predictions. In *Proceedings of the 2018 World Wide Web Conference*.

[65] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*.

[66] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*.

[67] Junyi Liu, Aleksandar Dragojevic, Shane Flemming, Antonios Katsarakis, Dario Korolija, Igor Zablotchi, Ho-cheung Ng, Anuj Kalia, and Miguel Castro. 2023. Honeycomb: ordered key-value store acceleration on an FPGA-based SmartNIC. *arXiv preprint arXiv:2303.14259* (2023).

[68] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.

[69] Ming Liu, Kaiyuan Zhang, Simon Peter, and Arvind Krishnamurthy. [n. d.]. TaichiSort: Energy-efficient Sorting of 1TB with NVMe and Coffee Lake. ([n. d.]).

[70] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*.

[71] Zaid Mahmoud. 2019. KioxiaSort: Sorting 1TB by 89K Joules. (2019).

[72] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*.

[73] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.

[74] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. 2000. Application Performance Pitfalls and TCP's Nagle Algorithm. *SIGMETRICS Perform. Eval. Rev.* 27, 4 (mar 2000), 36–44.

[75] J. C. Mogul and G. Minshall. 2001. Rethinking the TCP Nagle Algorithm. *SIGCOMM Comput. Commun. Rev.* 31, 1 (jan 2001), 6–20.

[76] Kathleen Nichols and Van Jacobson. 2012. Controlling queue delay. *Commun. ACM* 55, 7 (2012), 42–50.

[77] Padmanabhan Pillai, Michael Kaminsky, Michael A Kozuch, and David G Andersen. 2012. FAWNSort: Energy-efficient Sorting of 10GB, 100GB, and 1TB. *Intel Labs, Carnegie Mellon University* (2012).

[78] Martin Raab and Angelika Steger. 1998. "Balls into bins"—A simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*.

[79] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*.

[80] Robbert Van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*.

[81] Suzanne Rivoire, Mehul A Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. 2007. JouleSort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*.

[82] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*.

[83] Jeff Terrace and Michael J Freedman. 2009. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *USENIX Annual Technical Conference*.

[84] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. 2018. Lake: An energy efficient, low latency, accelerated key-value store. *arXiv preprint arXiv:1805.11344* (2018).

[85] Rui Wang, Christopher Conrad, and Sam Shah. 2013. Using set cover to optimize a large-scale low latency distributed graph. In *5th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 13)*.

[86] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. {HiKV}: a hybrid index {Key-Value} store for {DRAM-NVM} memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 349–362.

[87] Jiali Xing, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C. Lee. 2021. Charon: A Framework for Microservice Overload Control. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*.

[88] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. 2019. VStore: A Data Store for Analytics on Large Videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*.

[89] Juncheng Yang, Yao Yue, and KV Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*.

[90] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.

[91] Lujia Yin, Li Wang, Yiming Zhang, and Yuxing Peng. [n. d.]. {MapperX}: Adaptive Metadata Maintenance for Fast Crash Recovery of {DM-Cache} Based Hybrid Storage Devices. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[92] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. 2022. Aequitas: Admission Control for Performance-Critical RPCs in Datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*.

[93] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing*.

Appendices are supporting material that has not been peer-reviewed.

Appendix A Additional Evaluations

This section presents more evaluation results about different aspects of LEED. We use the same setup as described in §4.1.

A.1 LEED Data Store

A.1.1 Latency breakdown. Figure 11 provides the latency breakdown of GET/PUT/DEL operations. Apparently, SSD accesses dominate the request execution, i.e., 97.4% and 97.6% for the 256 and 1KB cases on average across three commands. Even though PUT issues one more NVMe access compared with the other two, it only adds 10.5μ more latencies since the first two SSD reads are overlapping.

A.1.2 Throughput Varying with Read/Write Ratio. We measure the single node throughput by varying the GET/PUT ratio. As shown in Figure 12, LEED throughput drops by 3.3% and 2.9% on average when adding 10% PUT for the 256B/1KB cases. FAWN (running over Raspberry PI) behaves differently since its data store uses a log-structured design where PUT runs faster than GET.

A.1.3 Compaction Optimization Results Figures 13a and 13b present the impact of parallelism on compaction. We consider three workloads: random write only (WR-ONLY), random mixed 50/50 read/write (MIX-50), mixed 50/50 read/write under Zip distribution (MIX-50-Zip) where skewness is 0.99.

A.2 LEED Cluster

A.2.1 Latency and Throughput under 256B Object Figure 14 presents the latency v.s. throughput of three platforms running with six YCSB workloads and 256B objects. The results are similar as the 1KB case.

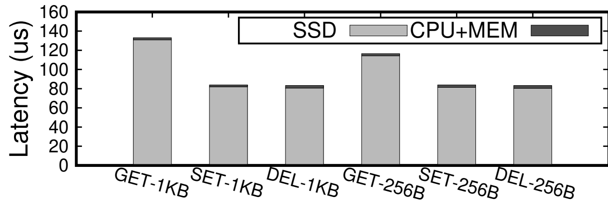


Figure 11: GET/PUT/DEL command latency breakdown for 1KB/256B cases. CPU and memory parts are combined.

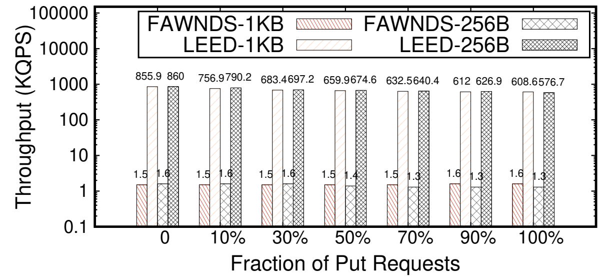
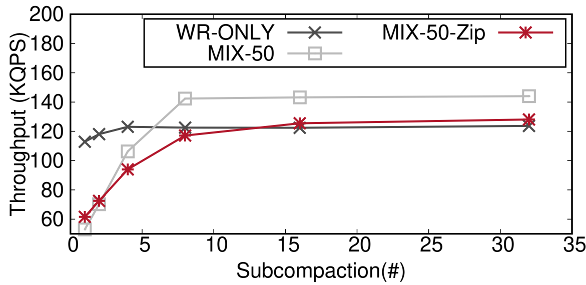
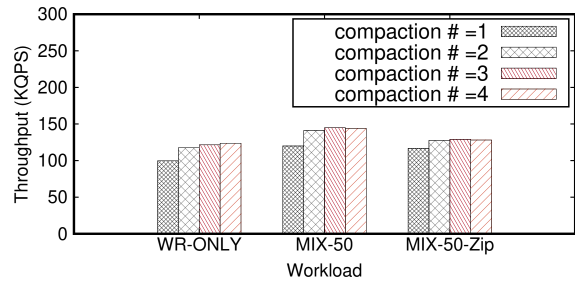


Figure 12: Throughput varying with the PUT percentage for both 256B/1KB under FAWN/LEED. y-axis is log scale.

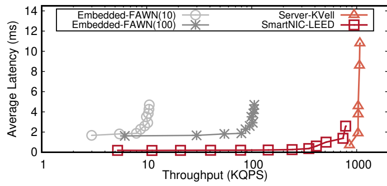


(a) Intra Parallelism

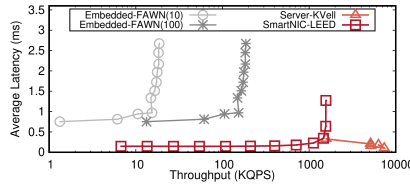


(b) Inter Parallelism

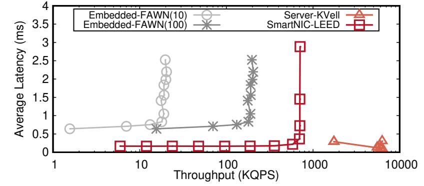
Figure 13: The impact of execution parallelism on compaction performance.



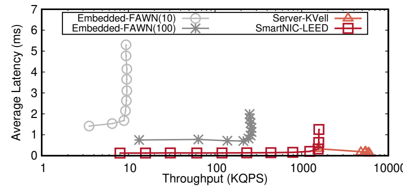
(a) YCSB-A



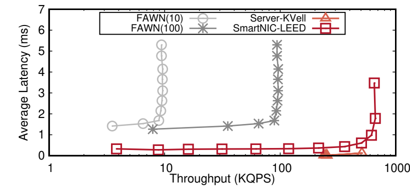
(b) YCSB-B



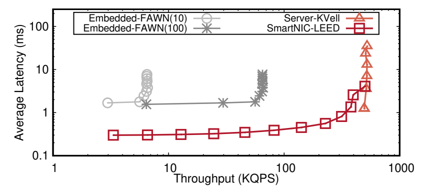
(c) YCSB-C



(d) YCSB-D



(e) YCSB-F



(f) YCSB-WR

Figure 14: Latency v.s. throughput for 6 YCSB workloads under the 256B case. FAWN(10) and FAWN(100) represent the number of nodes in the cluster is 10 and 100, respectively. X-axis is log scale for all scenarios. The y-axis of YCSB-WR is log scale.