A Kernel-Bypass Data Path for the DistWalk Distributed Workload Emulator

Niccolò Mulè[1][0009-0000-0968-6836], Jacopo Carlon[1][0009-0006-5351-9668], Giuseppe Lettieri[2][0000-0003-1005-7441], and Tommaso Cucinotta[1][0000-0002-0362-0657]

[1] Scuola Superiore Sant'Anna, Pisa, Italy
n.mule1@studenti.unipi.it, jacopo.carlon@santannapisa.it, tommaso.cucinotta@santannapisa.it
[2] University of Pisa, Pisa, Italy
giuseppe.lettieri@unipi.it

Abstract. With the growth of distributed systems in size and complexity, evaluating their end-to-end performance has become increasingly challenging. DistWalk is an open-source distributed workload emulator allowing for estimating the impact of key design decisions on the performance of an application, prior to its development. For performance-critical software, the overhead introduced by the networking stack has led to a broad adoption of kernel-bypass techniques. To enable DistWalk to model such deployments, we integrated a data path based on the Data Plane Development Kit (DPDK), a kernel-bypass framework. The DPDK code path is fully compatible with the existing socket-based implementation, letting both coexist within the same binary and allowing cross-protocol forwarding between DPDK and TCP/UDP nodes in the same topology. Support for multi-threading is provided by leveraging Receive Side Scaling (RSS) to dispatch incoming packets to different worker threads according to their source MAC address. On two servers connected through a 10 Gbps Ethernet link, DPDK achieved a median round-trip latency of 11.2 µs, approximately 2.7 times faster than UDP and 3 times faster than TCP in their best achievable configurations. The new feature is available in the original DistWalk repository.

Keywords: DPDK · DistWalk · Kernel Bypass · Low-latency Networking

1 Introduction

In the past few decades, the emergence of Cloud and Edge Computing has considerably changed the way applications are designed and deployed. Services that were once hosted on a single machine are now spread over geographically distributed sites, each one of which may be composed of multiple data centers and equipped with heterogeneous IT infrastructures [5]. Every component of a distributed application may contribute to

end-to-end performance variability in a way that is non-trivial to model. The end-to-end response time of a request is notably affected by various factors like the needed processing, storage access and networking operations; however, other configuration choices can have a substantial impact on latency and throughput, such as the CPU scheduling policy being used and its parameters [8], along with the disk I/O and network packet scheduling policy and options. Having a means to predict, or estimate, the performance profile achievable by applications *before* they are actually built can be extremely beneficial to make the correct design decisions. A wide range of benchmarking tools is available to measure various aspects of a system's performance, but they are usually narrow-scoped and struggle to capture the complexity of a distributed system. Network utilities like `ping`, `iperf3` and `traceroute` do not consider the aspects relative to computation and storage; CPU and storage bound utilities, such as `stress-ng` and `fio`, neglect the network; and passive monitoring tools can observe a system under load but cannot introduce stress themselves. Therefore, none of them can answer a question that naturally arises when designing distributed applications: *how can I evaluate the impact of the deployment configuration on end-to-end latency before I actually develop my application?* DistWalk [3] is a workload emulator designed to fill this gap by offering the capability to define a distributed workload in terms of a sequence of computing and storage operations to be carried out on a number of nodes, as well as forwarding operations across them. Workload generators and benchmarking suites of this kind are widely used in cloud and HPC contexts, as they allow the validation of design and deployment choices without requiring a full implementation of the application.

Applications where performance is critical incur an upper bound on packet processing speed, due to the Operating System (OS) networking stack, often including many unneeded features. To address this limitation, the industry standard for these workloads is to use techniques that bypass the kernel, either partially or completely. To enable DistWalk to model these deployments, we present a data path for DistWalk based on the Data Plane Development Kit (DPDK) [1], an industry-leading full kernel-bypass framework. As a result, developers can evaluate the performance impact of DPDK for their application, and decide whether its adoption justifies the increased development complexity compared to a socket-based approach.

Contributions. This paper makes the following contributions: (i) a DPDK-based data path for DistWalk that coexists with the socket-based one, enabling cross-protocol forwarding between DPDK-based and TCP/UDP-based nodes within the same topology (Section 4.2); (ii) a Layer-2 multi-

threading scheme that distributes incoming Ethernet frames across different threads by configuring the NIC to hash on the source MAC address (Section 4.3); (iii) a message-persistence mechanism that copies requests out of DPDK buffers into pre-allocated receive buffers, enabling asynchronous command processing without exhausting the memory pools (Section 4.4); (iv) an experimental evaluation assessing the latency gain of the DPDK path over UDP and TCP, as well as the impact of SR-IOV vs *veth* backends and NUMA placement (Section 5). The modifications described in this paper have been integrated in the original DistWalk repository at: https://github.com/tomcucinotta/distwalk/.

2 Background

2.1 DistWalk

DistWalk [3] is an open-source distributed workload emulator written in C. DistWalk features a client-server architecture, where a *client* builds a set of requests (a programmable sequence of commands which are processed in order) based on a user-defined pattern, and sends them to one or more node processes — the servers. The list of issuable commands includes **COMPUTE**, which performs a CPU-bound busy-wait for a specified duration (either deterministic or probabilistic), **LOAD** and **STORE** for I/O operations on storage files; **FORWARD** to simulate a Remote Procedure Call (RPC) on another DistWalk node and model flexible network topologies; and **PSKIP**, which skips the next command with a given probability, e.g., to model cache hits and misses. Each DistWalk server node consists of a pool of *worker* threads, that handle multiple clients by monitoring the file descriptors associated with the clients' connections, and a separate *storage* thread that executes storage commands. A DistWalk client is composed of one or multiple pairs of *sender* and *receiver* threads: the former encodes the requests, timestamps them, and sends them to the first node in the chain at a configured rate; the latter receives the responses from the nodes and computes round-trip latency metrics. DistWalk supports UDP, TCP and TLS, and offers granular control over low-level configuration parameters of both nodes and clients.

2.2 DPDK

Modern Ethernet interfaces can transfer data at rates ranging from 10 up to 100, or even 400, gigabits per second (Gbps). At a rate of 10 Gbps, a link can deliver 14.88 million minimum-size Ethernet frames per second,

which translates to roughly 67 ns to process each packet if the traffic were processed by a single CPU. At 100 Gbps, the time budget would drop below 7 ns. The ever-increasing throughput requirements lead to strict latency constraints, to the point where the overhead introduced by the OS and its networking stack becomes the major bottleneck [6, 13]. Motivated by this, a variety of so-called *kernel-bypass* techniques have been developed to access the network interface controller (NIC) entirely from userspace and therefore reduce the performance mismatch between hardware and software. The Data Plane Development Kit (DPDK) is a programming framework that implements kernel-bypass for NICs on x86, ARM and PowerPC processors. In DPDK, the NIC is unbound from its kernel driver and is instead bound to a minimal kernel module, typically `vfio-pci`, to expose direct device access to userspace. NIC interrupts are disabled and its descriptor rings are continuously monitored by userspace Poll Mode Drivers (PMDs), thus using 100% of a CPU core in order to deliver the lowest possible latency when a new packet is received. Packets are stored in `rte_mbuf` structures, the equivalent of `sk_buff` in Linux. Such mbufs live in pre-allocated memory pools, backed by hugepages to minimize TLB misses. DPDK API functions for the transmission and the reception of packets, such as `rte_eth_rx_burst()` and `rte_eth_tx_burst()`, are designed to move up to a maximum number of packets in batches to further optimize performance by amortizing the function call overhead. In DPDK, the threading model is based on a *run-to-completion* pattern, in which a DPDK thread, referred to as an *lcore*, is pinned to a dedicated core and executes a worker function in an infinite loop, never yielding the CPU. DPDK applications can scale by launching multiple lcores, each one operating on disjoint sets of receive and transmit queue pairs in order to avoid needing synchronization. In order to effectively distribute incoming packets to different queues, DPDK offers an API to program the Receive Side Scaling (RSS) configuration of the NIC.

3 Related Work

Various benchmark suites have been proposed for the evaluation of cloud and distributed applications. DeathStarBench [9] provides a collection of applications based on cloud microservices, e.g., social networks, hotel reservation, e-commerce and media streaming. TailBench [11] is a benchmark suite and evaluation methodology which focuses on latency-critical workloads and consists of nine applications spanning domains such as search engines, speech recognition, in-memory key-value stores and image

classification. ServiBench [14] draws its attention to serverless platforms, specifically. However, these suites mostly rely on pre-configured workloads, leaving the user with little control over configuration parameters of each benchmark. On the other hand, some mono-dimensional workload generators, such as YCSB [7], are widely adopted to stress a specific subsystem, but they only show one aspect of a more complex, and usually multi-tier, deployment. In contrast, DistWalk offers granular control over the behavior of the simulated workload, the multi-hop network topology, transport protocols and low-level configuration parameters of the individual nodes and clients. Moreover, to the best of our knowledge, no existing distributed-system benchmark provides a kernel-bypass data path, which is the gap we address with this work.

Among the most well-known kernel-bypass techniques there are netmap [13], AF_XDP [2] and DPDK [1], each one offering a different degree of kernel intervention in the data path. *netmap* patches the NIC kernel driver to allocate packet buffers in non-pageable kernel memory and maps this region in the virtual address space of the user process. *netmap*, however, requires periodic synchronization between NIC and application via system calls. AF_XDP, instead, leverages the extended Berkeley Packet Filter (eBPF): an eBPF program connected to the XDP hook intercepts packets that meet certain criteria and forwards them to a dedicated socket in userspace. The remaining packets follow their normal path towards the kernel, leaving the NIC available for ordinary traffic. Comparative studies of these three alternatives demonstrate that DPDK typically offers the lowest latencies and the highest throughput [4,12], at the cost of dedicating the NIC and at least one CPU core to the data plane.

4 Design and Implementation

The goal that guided our work was to achieve the lowest possible end-to-end latency between DistWalk nodes, while maintaining compatibility with the existing code path. Since DPDK completely bypasses the kernel, all communication between nodes and clients occurs at Layer 2 of the OSI model. While implementing Layer 3 features in userspace would have been possible, we deemed Layer 2 more interesting, as DPDK applications are typically deployed in high-performance clusters where all hosts can be assumed to be interconnected via Layer 2 switches. Another design choice was to adopt a one-mbuf-per-message approach. Each DistWalk message is encapsulated in a single mbuf and therefore in a single Ethernet frame. This avoids the complexity of reassembling a message that may be

fragmented across multiple frames, at the cost of limiting the size of the message to the Maximum Transmission Unit (MTU), minus the 14-byte Ethernet header. Since DPDK is particularly useful for latency-sensitive workloads, which tend to be characterized by relatively compact message sizes, this constraint has limited impact in practice; lifting it is left to future work (Section 6).

4.1 Packet Format and Addressing

The EtherType field of each frame is set to `0x88B5`, one of the two values IEEE reserves for local experimental use [10]; any incoming frame with a mismatching EtherType is silently dropped. The Ethernet header is followed by a `message_t` structure, identical to the one used in the socket-based code path. This structure contains a field for a request identifier, the size of the message, a status code and an array of one or more `command_t` structures, terminated by an `EOM` record.

On the command line, the DPDK bind address is specified in a URI-like scheme of the form `dpdk://`, accepting either a network interface name (e.g., `dpdk://veth0`, a virtual interface used with an `AF_PACKET` socket) or the PCI address of a physical or virtual device (e.g., `dpdk://0000:04:00.0`).

In the socket-based code path, the target address of a forward operation is stored in the `fwd_opts_t` structure as a 4-byte IPv4 address plus a 2-byte port number (6 bytes in total). For DPDK-based forwarding, there are no IPs or ports; the target address is instead a 6-byte MAC address. These two representations are placed in a C `union` within the same `fwd_opts_t` structure. As both representations are 6 bytes long, a `memcmp` operation will work regardless of the protocol in use.

4.2 Coexistence with the Socket-Based Code Path

On the node, the worker thread's main loop handles both the DPDK and socket file descriptors. In a single iteration of the loop, the worker drains the DPDK receive queue using `rte_eth_rx_burst()`, processes the received packets, flushes the DPDK transmit queue with `rte_eth_tx_burst()` and then handles socket events using non-blocking I/O multiplexing functions. The node can therefore bind to a DPDK interface and one or more TCP or UDP ports and is able, if requested, to perform cross-protocol forwarding. For instance, a client may send a request using DPDK to node A, asking it to FORWARD to node B via TCP. Node A opens a TCP connection with node B and, upon receiving the response, copies it into a DPDK mbuf and sends it back to the originating client (Figure 1). This functionality can

be particularly useful to model scenarios in which nodes must fall back to TCP or UDP to reach hosts that are outside the local Layer 2 domain.

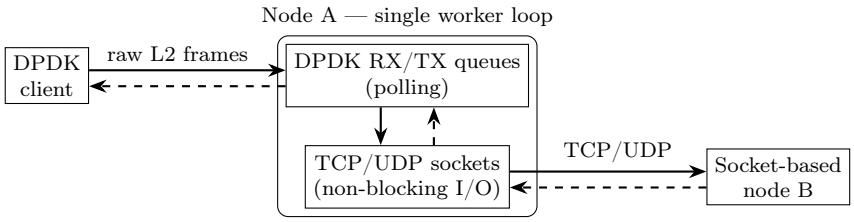


Fig. 1. Cross-protocol forwarding between the DPDK and the socket-based data paths, both served by the same worker loop. Solid arrows denote the request path, dashed arrows the reply.

4.3 Multi-Threading via RSS

A DistWalk node can be configured to use multiple worker threads. With sockets, the operating system is responsible for distributing the incoming connections to each thread. With DPDK, however, the kernel is bypassed, and each worker thread polls its own dedicated receive queue to check for incoming packets. For the hardware to place packets in separate queues, RSS must be configured on the NIC during the initialization of DPDK. At the hardware level, the NIC computes a hash function on a number of fields within each packet, and uses the least significant bits of the resulting hash value to index an indirection table that maps each packet to one of the available queues. RSS typically hashes Layer 3 fields, such as the source IP address. However, packets that DistWalk sends via DPDK are raw Ethernet frames, so RSS must be configured in two steps. Firstly, during the setup of the Ethernet port, the `rss_hf` field, a bitmask that specifies the type of packet fields to be hashed, is set to `RTE_ETH_RSS_L2_PAYLOAD` masked against `flow_type_rss_offloads`, which reports which RSS hash types are supported by the underlying device. Secondly, the bytes of the Ethernet payload are selected for hashing using the vendor-specific function `rte_pmd_i40e_inset_set()`, provided by the corresponding PMD. This function instructs the NIC to extract the headers of received packets and place them into a field vector of sixty-four 16-bit words, of which words 3–5 correspond to the source MAC address. As a result, packets from different clients, which have different source MAC addresses, hash to different RSS values and are distributed to different queues.

4.4 Asynchronous Commands and Message Persistence

In DistWalk, commands of type COMPUTE are executed synchronously. Other commands may stall the worker thread much longer and are therefore handled asynchronously. STORE and LOAD operations are assigned to the dedicated storage thread, which notifies their completion via a pipe. Similarly, FORWARD operations require waiting for a reply from the node downstream. In both of these cases, however, the worker thread can continue to handle incoming requests. For TCP and UDP, this does not raise any concern, as the message is persisted in large receive buffers. In DPDK mode, instead, the message lives in an mbuf allocated from a fixed-size mempool. Returning the mbuf to the mempool right away would cause the worker to lose the commands yet to be executed in the message, while holding onto the mbuf until the operation completes poses the risk of exhausting the mempool if the node is under heavy load. The solution we found is to copy out the message into one of DistWalk's pre-allocated receive buffers, before the mbuf is returned to the mempool.

5 Evaluation

5.1 Experimental Configuration

All the experiments were carried out on two twin servers, each equipped with two Xeon E5-2640 v4 processors (10 physical cores per socket, 2.40 GHz, Turbo Boost up to 3.40 GHz and Hyper-Threading enabled). The two sockets in each server are organized in a Non-Uniform Memory Access (NUMA) configuration. Both servers have a dual-port Intel X710 10 Gigabit SFP+ NIC with NUMA affinity to node 0. The two hosts' NICs are interconnected via a 10 Gigabit SFP+ Ethernet cable. All DPDK processes are pinned to cores on the NUMA node local to the NIC, to minimize latency. The servers run Ubuntu 24.04.3 with Linux kernel 6.8.0 and use DPDK version 25.03, compiled from source. In most experiments, the NIC is unbound from its `i40e` driver and bound to `vfio-pci` for DPDK use.

In order to obtain reproducible results, the kernel boot parameter `isolcpus` was used to exclude the cores assigned to DPDK from the Linux scheduler. The CPU governor is set to `performance` mode and frequency is locked via `intel_pstate`; Turbo Boost is disabled and the hardware energy-performance bias is set to favour performance over power saving. All DistWalk processes are launched with the `SCHED_FIFO` scheduling policy and maximum priority. `sched_rt_runtime_us` is set to -1 to prevent the

kernel from imposing a 95% CPU limit on real-time tasks. Since Hyper-Threading is enabled, only one logical CPU of each physical core is assigned to worker threads, while its sibling is left idle to avoid contention. For TCP and UDP experiments, the NIC receive-queue interrupt is pinned to the core running the DistWalk process. Unless otherwise specified, each experiment is composed of 10 000 requests sent at a 2 ms period.

5.2 Round-Trip Latency: DPDK vs UDP vs TCP

The first experiment we present compares the round-trip latency obtained from DPDK, UDP and TCP. This experiment represents a pure latency test, meaning that no computation or storage operations were required, thus exclusively reflecting the transport overhead. For TCP and UDP we picked the best achievable configuration, obtained with the `--wait-spin` DistWalk flag, which makes I/O multiplexing calls non-blocking, effectively a zero-timeout polling loop preventing the core from entering idle states between packets. Figure 2 shows the latency distribution for each one

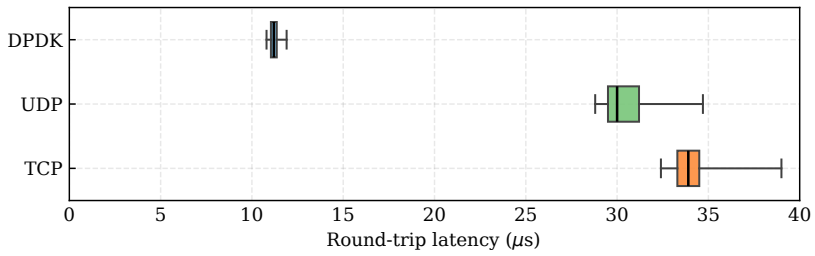


Fig. 2. Per-request round-trip latency distribution for DPDK, UDP, and TCP. Boxes show the median and the first/third quartiles; whiskers the 1st and 99th percentiles.

of the three configurations. DPDK achieves a median latency of 11.2 µs, roughly 2.7 times faster than UDP (30 µs) and 3 times faster than TCP (33.9 µs).

5.3 SR-IOV Virtual Functions vs Virtual Ethernet Pairs

The DPDK data path supports two testing configurations: SR-IOV *Virtual Functions* (VFs), which can independently be bound to `vfio-pci` to access dedicated hardware directly, or *virtual Ethernet* pairs (*veth*), which use an `af_packet` PMD to send and receive packets via a Linux bridge. In this

experiment, we compare the performance of these two testing setups on a single machine. The VF configuration achieves a median latency of 10.9 µs, about 12% faster than *veth* at 12.4 µs (Figure 3). This result was to be expected, as in the former case traffic is handled by the internal Layer 2 switch of the NIC, while for *veth* it must traverse the virtual bridge in the kernel. The *veth* configuration, however, remains useful to test the DPDK code path on hosts lacking a NIC with either SR-IOV or DPDK support.

5.4 Impact of NUMA Affinity

Figure 4 illustrates the performance impact of NUMA placement, evaluated by running each of the four combinations of node and client across the two NUMA nodes. Both node and client are executed on the same machine by binding each process to a separate SR-IOV Virtual Function. When both processes are running on NUMA node 0 (local to the NIC), the measured median latency is lowest at 10.9 µs. Moving only one process to NUMA node 1 raises the median value to 11.3–11.4 µs, while moving both processes to NUMA node 1 increases it to 11.7 µs (a 7% performance degradation compared to the best case). This can be explained by the NIC's affinity to NUMA node 0, causing processes that run on NUMA node 1 to pay a latency cost when traversing the QuickPath Interconnect.

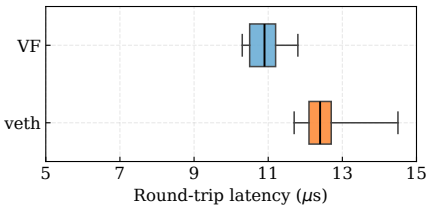


Fig. 3. Round-trip latency for the VF (vfio-pci) and *veth* (af_packet) configurations. Boxplot conventions as in Figure 2.

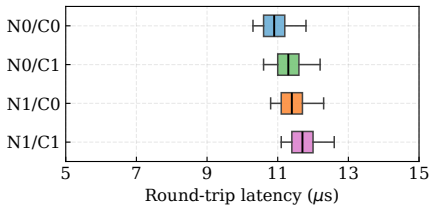


Fig. 4. Impact of NUMA placement: Nx/Cy denotes the node process running on NUMA node x and the client on node y. The NIC has affinity to NUMA node 0. Conventions as in Figure 2.

5.5 Multi-Queue Scaling with RSS

This qualitative experiment verifies the ability of DistWalk nodes to scale to multiple worker threads when in DPDK mode. Two clients, run on the

same machine, are bound to separate VFs with different MAC addresses and are configured to target the same node, executed on the other host. The compute time was set to 1 500 µs to ensure that the execution times of both clients were long enough so that their requests overlap. The transmission period was set to 2 ms, thus causing, on the node side, an aggregate arrival rate of one packet per millisecond. Given that each packet requires a 1 500 µs service time, latencies grow unbounded if the node is single-threaded. By running the node with two worker threads, which enables RSS to assign each packet to a specific queue, we expect latencies to remain steady. We repeat this experiment 30 times, with 1 000 packets per run, reporting the mean and its 95% confidence interval. The results display a 1 514 µs mean latency for both clients, which is the compute time plus approximately 14 µs of transport overhead, indicating that the two queues are operating independently of one another.

Discussion. The estimated 20 µs latency reduction per request that DPDK can offer over the socket-based data path (see Figure 2) may be decisive for applications where request latency is a major concern. However, the benefit becomes marginal when computation or storage operations dominate the response times. In practice, the intended workload can be emulated both with and without DPDK, so that the actual latency improvement can be estimated for a specific application, even prior to its development.

6 Conclusion

In this paper, we presented a data path for DistWalk built upon DPDK and coexisting with the current socket-based code path. By doing so, we extended DistWalk's capabilities allowing it to model applications that adopt kernel-bypass techniques, while still maintaining compatibility with TCP, UDP and TLS. In the experimental evaluation, DPDK proved to provide considerable benefits when performance counts, displaying round-trip latency metrics that are 2.7 and 3 times faster than UDP and TCP, respectively. During testing, we also assessed the performance impact of two on-host configurations (SR-IOV Virtual Functions vs virtual Ethernet pairs) and NUMA node placement, and verified that DistWalk in DPDK mode scales correctly with RSS.

Future work includes (i) extending multi-threading support to the client side, which is currently limited to a single pair of threads; (ii) generalizing the support for RSS, currently restricted to Intel i40e; (iii) lifting the single-frame limit by fragmenting messages or by layering a userspace L3/transport stack over the DPDK data path; (iv) evaluating the

performance of the data path at higher link rates, such as 100 and 400 Gbps; (v) integrating other kernel-bypass frameworks, such as *netmap* [13] or AF_XDP [2], to expand the range of scenarios that DistWalk can model.

References

1. About DPDK, https://www.dpdk.org/about/, last accessed: May 2026
2. AF_XDP, https://www.kernel.org/doc/html/next/networking/af_xdp.html, Linux kernel documentation, last accessed: May 2026
3. Andreoli, R., Cucinotta, T.: DistWalk: A distributed workload emulator. In: 2025 IEEE 25th International Symposium on Cluster, Cloud and Internet Computing (CCGrid). pp. 1–10 (2025). https://doi.org/10.1109/CCGRID64434.2025.00042
4. Ara, G., Lai, L., Cucinotta, T., Abeni, L., Vitucci, C.: A framework for comparative evaluation of high-performance virtualized networking mechanisms. In: Int. Conf. on Cloud Computing and Services Science. pp. 59–83. Springer (2020)
5. Buyya, R., Vecchiola, C., Selvi, S.: Mastering Cloud Computing: Foundations and Applications Programming. Morgan Kaufmann (2013)
6. Cai, Q., Chaudhary, S., Vuppalapati, M., Hwang, J., Agarwal, R.: Understanding host network stack overheads. In: Proceedings of the 2021 ACM SIGCOMM Conference. pp. 470–484 (2021). https://doi.org/10.1145/3452296.3472888
7. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proc. of the 1st ACM Symposium on Cloud Computing (SoCC). pp. 143–154. ACM (2010). https://doi.org/10.1145/1807128.1807152
8. Cucinotta, T., Abeni, L., Marinoni, M., Mancini, R., Vitucci, C.: Strong temporal isolation among containers in OpenStack for NFV services. IEEE Transactions on Cloud Computing **11**(1), 763–778 (2023). https://doi.org/10.1109/TCC.2021.3116183
9. Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., et al.: An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In: Proc. of the 24th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 3–18. ACM (2019). https://doi.org/10.1145/3297858.3304013
10. IEEE: IEEE standard for local and metropolitan area networks: Overview and architecture (2014), section 9.2, EtherType allocations
11. Kasture, H., Sanchez, D.: TailBench: A benchmark suite and evaluation methodology for latency-critical applications. In: Proc. of the IEEE Int. Symposium on Workload Characterization (IISWC). pp. 1–10. IEEE (2016). https://doi.org/10.1109/IISWC.2016.7581261
12. Lettieri, G., Maffione, V., Rizzo, L.: A survey of fast packet I/O technologies for network function virtualization. In: High Performance Computing. pp. 579–590. Springer (2017). https://doi.org/10.1007/978-3-319-67630-2_40
13. Rizzo, L.: netmap: A novel framework for fast packet I/O. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). pp. 101–112. USENIX Association (2012)
14. Scheuner, J., Eismann, S., Talluri, S., van Eyk, E., Abad, C.L., Leitner, P., Iosup, A.: Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications (2022). https://doi.org/10.48550/arXiv.2205.07696