Building Blocks for Network-Accelerated Distributed File Systems

Salvatore Di Girolamo, Daniele De Sensi, Konstantin Taranov, Milos Malesevic,
Maciej Besta, Timo Schneider, Severin Kistler, Torsten Hoefler
Dept. of Computer Science, ETH Zürich, 8092 Zürich, Switzerland
{salvatore.digirolamo, daniele.desensi, konstantin.taranov, maciej.besta, timo.schneider, htor}@inf.ethz.ch
{milos.malesevic, kistlers}@student.ethz.ch

***Abstract*—High-performance clusters and datacenters pose increasingly demanding requirements on storage systems. If these systems do not operate at scale, applications are doomed to become I/O bound and waste compute cycles. To accelerate the data path to remote storage nodes, remote direct memory access (RDMA) has been embraced by storage systems to let data flow from the network to storage targets, reducing overall latency and CPU utilization. Yet, this approach still involves CPUs on the data path to enforce storage policies such as authentication, replication, and erasure coding. We show how storage policies can be offloaded to fully programmable SmartNICs, without involving host CPUs. By using PsPIN, an open-hardware SmartNIC, we show latency improvements for writes (up to 2x), data replication (up to 2x), and erasure coding (up to 2x), when compared to respective CPU- and RDMA-based alternatives.**

***Index Terms*—File systems, next generation networking**

I. Introduction

Distributed File Systems (DFSs) play a fundamental role in tackling the growing I/O bottleneck. By decoupling control and data planes, these architectures can be easily managed and scaled out. While there exist a plethora of DFS architectures, it is possible to identify building blocks that are fundamental and typically implemented by all of them. For example, clients must be authenticated and their requests must be validated. If this does not happen, a client can write to any storage location, violating tenant isolation and potentially bringing the file system to an inconsistent state. Additionally, data must be stored resiliently by either replicating it on different storage nodes or storing it together with parity blocks (i.e., erasure-coded). Without resiliency, the failure of a single storage node can compromise the entire file system or large parts of it.

These building blocks, which we call *DFS policies*, are defined by the DFS control plane and enforced in the data plane. For example, file or object metadata store access permissions and whether and how the file or object must be replicated or erasure-coded. Whenever clients access the data, these policies must be enforced: i.e., the client request must be validated and the data must be eventually replicated or erasure-coded.

Until recently, the performance of data plane storage operations has been greatly limited by storage media performance. Hence, factors like network overheads, data copies, and CPU utilization were of negligible importance. This led to the introduction of complex software layers into the storage nodes to implement strategies for easing storage media bottlenecks (e.g., batching, striping), and enforce DFS policies.

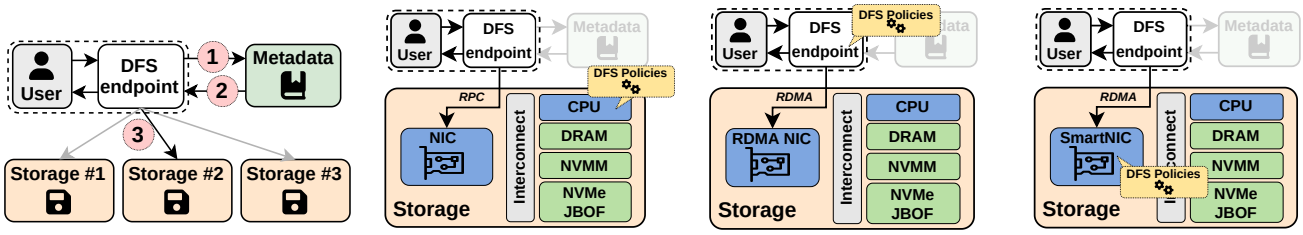
However, while this assumption does not hold at all for in-memory file systems, it must also be revised for persistent storage. With the emergence of dense, byte-addressable non-volatile main memories (NVMMs) [1] and NVMe JBOFs (Just a Bunch of Flash) [2], storage media can ingest data at network speed or faster. Hence, network and software overheads start playing an important role and must be optimized to not become bottlenecks. In this direction, remote direct memory access (RDMA) [3] has been the focus of many DFS optimizations [4]–[8]. RDMA provides low latency and high bandwidth one-sided communications, enabling clients to access memory of remote storage nodes without involving their CPUs.

Unfortunately, RDMA provides negligible compute capabilities, which are insufficient to express custom DFS policies. Consequently, DFSs usually still rely on storage node CPUs to enforce custom policies by using remote procedure calls (RPC) for triggering policy enforcement and RDMA to move data. However, this approach loses the one-sided characteristic of RDMA, which is key for getting low latency and high throughput. To work around this issue and fully embrace RDMA, some DFSs [9]–[11] delegate policy enforcement to clients. However, this can lead to worse performance (e.g., for replication, a client must write to each replica) or require higher trust (e.g., without requiring request validation).

We show how DFS policy enforcement can be offloaded to SmartNICs (smart network interface cards). When offloaded, these policies are enforced directly on the SmartNIC, without involving the storage node CPU. Additionally, we show how packet-level processing can accelerate the execution of these policies, even when compared to the case where they are enforced by the CPU of the storage nodes. *This work aims at filling the gap between full-RDMA DFSs, which provide best performance for raw writes but do not support custom DFS policies, and CPU-based DFSs, that allow expressing any DFS policy at the cost of additional overheads (e.g., PCIe latency).* All in all, we introduce the following contributions:

- Show how SmartNICs enable high-performance implementation of DFS-specific protocols;
- Demonstrate how fully programmable SmartNICs with packet-level processing capabilities can accelerate multi-node communications required for, e.g., data replication;
- Apply packet-processing techniques to offload complex tasks such as erasure coding. Not only this approach results better than proprietary, firmware-based solutions but also

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE



(a) (b) (c) (d)

Fig. 1. (a) DFS operational model assumed in this paper; (b) DFS policies implemented in storage nodes CPU; (c) with RDMA; and (d) with SmartNICs.

improves the time-to-market of new strategies as they are not constrained anymore by vendor-dependent deployments.

- We study the benefits of network acceleration for DFSs on an open-hardware, fully programmable SmartNIC. The entire toolchain, comprised of cycle-accurate simulations, is available online, easing reproducibility and enabling quick investigation of new hardware features for future research.

We envision that the approach taken in this work to offload DFS building blocks can be followed for designing next-generation DFSs, allowing them to benefit from RDMA performance while effectively enforcing storage policies.

II. Distributed File Systems and SmartNICs

By decoupling control and data planes, DFSs can distribute I/O accesses and abstract technology- and vendor-specific storage backends, providing an independent and standard interface to the clients while easing the management of conventional storage systems. Commonly, DFSs are composed of three main services: management, metadata, and storage. Activities such as authentication and monitoring are carried out by the management service. Control plane tasks are performed by the metadata service that indexes files/objects and references the actual data stored by the storage service.

Figure 1(a) shows a typical DFS workflow. A user is interfaced with a DFS endpoint that can be either a library, a kernel module, or a separate network node. In the following, we do not make a distinction between users and DFS endpoints, referring to both of them as *client*. The client first authenticates through the DFS endpoint with the management service (not shown in the figure). Then, to access file or object data, it queries the metadata service (1) to retrieve the file layout (2). A file layout describes the regions (e.g., objects or blocks) composing a file and address information (e.g., on which storage node a region is stored and at which storage address). This information allows the client to communicate directly with the storage node for accessing the data (3).

A. DFS policies

A DFS policy is a set of actions defined by a distributed file system to be enforced when clients access data. These policies are defined in the control plane (i.e., management and metadata service) and enforced in the data plane (i.e., storage nodes). Normally, overheads introduced by the enforcement of these policies are factored in the data access (e.g., read or write) latencies. We investigate how DFS policies can be enforced directly by the network interfaces of the storage nodes. By enforcing them on the NIC, on per-packet basis, we can improve the overall latency of write operations besides lowering CPU usage on the storage nodes. While each DFS can define its own custom policies, we identify three classes of policies of general interest for DFSs: protocol, data movement, and data processing policies. In this paper, we select one representative policy for each class:

Protocol: client request authentication. Data access requests issues by clients to storage nodes must be validated. This validation can avoid the case of a malfunctioning or malicious application acting as a client and bringing the DFS to an inconsistent state. With this policy, a client must first obtain a ticket or capability from the metadata node and then use it to make requests to the storage nodes. Storage nodes can verify the legitimacy of the request through the capability.

Data movement: replication. Data replication improves reliability and fault tolerance capability of DFSs. The idea is to replicate the data on k storage nodes, where k can be a global, per-pool, or per-file parameter. The metadata nodes store the replication information, and the replication strategy is triggered whenever new data is stored.

Data processing: erasure coding. Data replication is characterized by high storage costs, which are linear in the replication factor. With erasure coding (EC), data is split into k chunks and stored together with m parity chunks. In case of failure, missing chunks can be recovered by using the remaining ones.

Figures 1(b-d) sketch different DFS architectures for the storage nodes. In a CPU-centric storage node architecture (Figure 1b), the client issues remote procedure calls (RPC) to the storage node, which trigger software components on the CPU of the storage node enforcing DFS policies. Figure 1c shows an RDMA-centric approach, where the CPU of the storage node is bypassed. With this approach, the clients must enforce DFS policies, as RDMA does not expose a programmable interface. Finally, Figure 1d, shows a SmartNIC-centric approach, which we explore in this paper, where DFS policies are implemented directly in the network interface of the storage nodes.

B. Network acceleration

As our goal is to offload the enforcement of these policies to the network, we now identify a set of principles that an in-network compute solution should provide in order to enable effective DFS policy offloading.

One-sided requests. Storage nodes should handle data accesses and relative policy enforcement without involving the host CPU of the storage nodes and without requiring long-

lived per-client data structures. For example, if storage nodes actively participate in the data replication process (e.g., as they are arranged in a tree or ring virtual topology), the forwarding to next replicas should not involve the host CPU nor require to maintain long-term information about the virtual topology. While RDMA-based solutions allow one-sided data access, they are not designed to execute custom compute tasks (i.e., DFS policies).

Flexibility. DFS policies can be complex and data-dependent. For example, erasure coding policies need to access the entire packet payload to compute parities. For this, RDMA-based solutions would need to rely on vendor-specific accelerators for erasure coding [12]. Solutions based on P4 [13] do not provide iteration constructs (making deep-packet inspection challenging), pointers, references, and inter-packet state. Solutions based on eBPF/XDP [14] are limited by the programming model (e.g., bounded loops, limited number of instructions, limited packet forwarding capabilities) [15]. While it is possible to extend existing NIC designs [16], [17] to implement DFS policies in hardware, this would increase the complexity of expressing and deploying such policies.

User-level. DFSs should be able to install custom policies without needing privileged access to the system (i.e., admin rights). This capability opens up network acceleration also to user-level I/O libraries such as DAOS [18]. Expressing policies in eBPF/XDP would mean that the DFS-policy sees all packets coming through the network interface and reacts to specific ones (e.g., write requests). However, this rules out the possibility of having user-space applications (without elevated privileges) installing new policies because it would break isolation principles (i.e., a policy installed by an application could access packets targeting other applications). Similar issues arise for DPDK [19] and SmartNICs providing programming models based on eBPF/XDP [14] and DPDK [20].

1) Streaming processing in the network: Network communications, or messages, consist of streams of packets traveling between pairs of endpoints. With message-based processing, the receiving endpoint receives the full stream of packets before processing it. This is what happens when processing data received with, e.g., MPI point-to-point communications. With this approach, the NIC has to copy packet payloads to host memory before triggering CPU processing. A different approach is to leverage the streaming nature of network communications and process packets as the endpoints receive them. This approach exposes packet-level parallelism, potentially accelerating the processing of incoming data as multiple packets are processed in parallel. When data processing is offloaded to the NIC, streaming processing also allows to have lighter memory requirements: i.e., there is no need to buffer the full message on the NIC before being able to process it, but only the packets currently being processed.

sPIN [21] is an in-network compute solution operating on packet streams , which enables the offloading of DFS policies according to the above-described principles. In particular, sPIN enables applications to define lightweight kernels executed on incoming network packets directly on the NIC. These kernels,

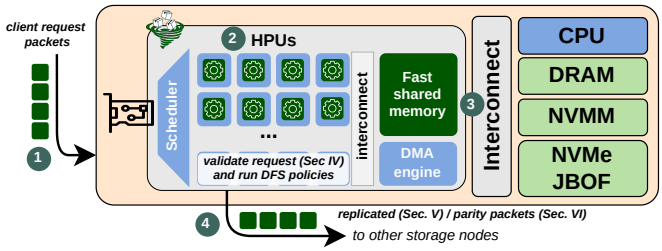


Fig. 2. Network-offloaded DFS policies overview.

called *handlers*, are defined on classes of messages. When packets of a message matching a given class arrive, sPIN schedules the corresponding handlers to be executed on the on-NIC *Handler Processing Units* (HPUs). To process packets, an application defines three handlers: the header handler (HH), executed only on the first packet of a message; the completion handler (CH), run on the last packet of a message (i.e., completion packet); and the payload handler (PH), executed on all packets of a message (included header and completion). sPIN requires that the network delivers the header packet first and the completion packet last, without introducing additional constraints on payload packets.

sPIN provides an RDMA+X programming model, where the X is a per-packet task defined by the application running on the host and executed on the NIC, fully implementing the **one-sided requests** principle. sPIN handlers can be expressed in C/C++ and, differently from P4 and eBPF/XDP, are not subject to restrictions on actions that can be performed on the incoming packets, providing **flexibility**. Finally, differently from solutions based on DPDK, P4, and eBPF/XDP, sPIN enables **user-level** applications to offload compute tasks to the NIC, providing isolation by design. This is achieved by matching packets to application-defined execution contexts (i.e., providing information on, e.g., which packet handlers to run), similarly to the way packets are matched to queue pairs in RDMA or to match list entries in Portals 4 [22].

In this work, we use an open-source implementation of sPIN, PsPIN [23]. PsPIN is a PULP-based [24] packet processor composed of 32 RISC-V cores running at 1 GHz (i.e., the HPUs), divided into four compute clusters. Each compute cluster is equipped with a 1 MiB single-cycle access memory (L1) and an off-cluster 4 MiB memory (L2). In addition, the accelerator includes a hardware packet scheduler that provides low-latency scheduling (1-2 cycles) and DMA engines to move data between NIC memories and interface with host memory.

III. NIC-offloaded DFS policies

Figure 2 shows an overview of our approach to network-offloaded DFS policies. Clients issue data access requests to storage nodes 1, triggering DFS policies that are executed on the HPUs of the sPIN-enabled NIC 2. The DFS handlers authenticate requests coming from the client (see Section IV), write data to host 3, and enforce other DFS policies 4 such as replication (Section V) and erasure coding (Section VI). By running DFS policies as sPIN handlers, we can perform actions (e.g., forward packets to the next replica node) before

data reaches host memory via the system interconnect, saving latency for small writes (e.g., a PCIe round-trip can take up to 400 ns [25]), and leveraging packet-level parallelism for larger ones (i.e., composed of multiple network packets).

We do not focus on a specific storage medium. Instead, we focus on showing the benefits of offloading DFS policy enforcement to the NIC and assume that the storage medium can digest data at network bandwidth or higher. For example, with in-memory or non-volatile-main-memory (NVMM) based DFSs, handlers would write directly to main memory, as any other RDMA DFS [9], [11]. For DFSs targeting NVMe just-a-bunch-of-flash (JBOF), handlers would directly issue NVMe writes via the system interconnect (e.g., PCIe).

A. Client request format

Figure 3 shows the layout of write and read requests. A write request consists of: an RDMA header (e.g., InfiniBand or RoCEv2); a generic DFS header carrying information to identify the request (e.g., operation type) and to authenticate it; a write request header (WRH) that carries write-specific information, such as the replication strategy and the number and coordinates of the replica nodes and respective address where data must be replicated. The headers are followed by the packet payload (i.e., the data to write). If the write spans multiple network packets, then only the first packet carries DFS-specific headers, while others consist of the RDMA header and the continuation of the data to write. A read request carries the RDMA header, the DFS header, and a read request header (RRH) with read-specific information.

We assume that request headers (DFS and WRH/RRH) always fit in a single network packet. This assumption is realistic for, e.g., RoCE networks, where packet sizes are limited by Ethernet maximum transmission unit (MTU), which typically ranges between 1.5 KiB and 9 KiB (jumbo frames).

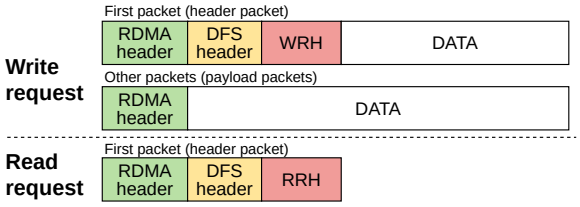


Fig. 3. Packet format for write and read requests.

B. sPIN handlers

Listing 1 shows the pseudocode of generic sPIN handlers for offloading DFS components. Each handler takes two arguments: a task descriptor and the pointer to a network packet. The task descriptor contains information about the corresponding execution context, such as NIC memory allocated for and shared by the handlers running on packets matched by this specific execution context.

There are two types of DFS-specific tasks that can be executed on each request: per-request tasks and per-packet tasks. Per-request tasks can be expressed in `DFS_request_init` and `DFS_request_fini`, which are called once per message, i.e., in the header and completion handlers, respectively. Per-request tasks can be used for, e.g., validating the request before processing other packets of the same request (i.e., sPIN guarantees that the payload handlers of a message are executed after the header handler of that message completes). As the completion handler is executed only once all packets of a message have been processed, the DFS can use `DFS_request_fini` for finalizing the handling of the request: e.g., send the write acknowledgment back to the client. Additionally, a payload handler is executed for each packet (including header and completion). Here, the DFS can run per-packet actions such as copying the payload to the storage media and sending the packet to the next replica node.

These handlers are triggered for all incoming client requests: i.e., the first packet of any request triggers a header handler, the last triggers a completion handler, and all packets trigger a payload handler. They are persistent and do not need to be installed on a per-request basis nor require an established connection between clients and storage nodes.

```
1  void header_handler(spin_task_t* task, pkt_t* pkt) {
2    dfs_state_t* state = (dfs_state_t*) task->mem;
3
4    bool accept_next_pkts = DFS_request_init(state, pkt);
5    // DFS_request_init sends NACK if request auth fails
6
7    int req_idx = task->flow_id;
8    state->req_table[req_idx].greq_id = pkt->dfs.greq_id;
9    state->req_table[req_idx].accept = accept_next_pkts;
10 }
11
12 void payload_handler(spin_task_t* task, pkt_t* pkt) {
13   dfs_state_t* state = (dfs_state_t*) task->mem;
14   int req_idx = task->flow_id;
15
16   if (state->req_table[req_idx].accept)
17   DFS_request_process_pkt(state, pkt);
18   //else packet is dropped
19 }
20
21 void tail_handler(spin_task_t* task, pkt_t* pkt) {
22   dfs_state_t* state = (dfs_state_t*) task->mem;
23   int req_idx = task->flow_id;
24
25   if (state->req_table[req_idx].accept)
26   DFS_request_fini(state, pkt);
27   //else packet is dropped
28 }
```

Listing 1. Generic sPIN handlers for offloading DFS tasks.

1) Data persistence: A write operation completes when the data reaches the storage target. If client requests are handled on CPUs of the storage nodes, the DFS can wait for the data to be written to the memory target before acknowledging the client. In RDMA-based DFSs, this task is slightly more complex. There, when a client gets an RDMA write completion event, the data could be on the storage node persistent memory or still be buffered somewhere between the NIC and the storage target (e.g., PCIe buffers). To overcome this limitation, the client can issue an RDMA read immediately after a write to flush DMA buffers, triggering a read-after-write dependency [25].

While RDMA extensions are being proposed to introduce an RDMA *flush* operation [26] that could be merged together with a write to save the additional RDMA read latency, this is a good example of the advantages of using SmartNICs. With sPIN, packet handlers can explicitly issue writes to the storage medium, making sure that the data is flushed before

acknowledging the client, as it would happen on the CPU. Additionally, a sPIN handler can overlap other activities (e.g., running other DFSs policies, such as replication) while waiting for the data to be flushed to the storage target.

2) Scalability: Each write request requires to keep a state in the NIC for the whole operation duration (i.e., `req_table` entries in Listing 1). Each entry is a write descriptor that takes 77 bytes and stores the current status of the request plus information that is carried only by the header packet and that is needed by payload handlers. For example, for data replication (see Section V), we use a source-based approach where the WRH carries replica addresses. As we need to forward all packets of the write request, we store these addresses in the per-request state to make forwarding possible.

In PsPIN, each one of the four compute clusters is equipped with a 1 MiB single-cycle access memory (L1). Additionally, there is an off-cluster NIC memory of 4 MiB (L2). We store client request descriptor into L1 and use L2 as a swap-out area. In total, we have 6 MiB of available memory to store client request entries, while the remaining 2 MiB are used to store DFS-wide state. This allows us to serve up to ∼82 K concurrent writes for each storage node. If a client request cannot be served because of lack of space, the request is denied, and the client will retry later.

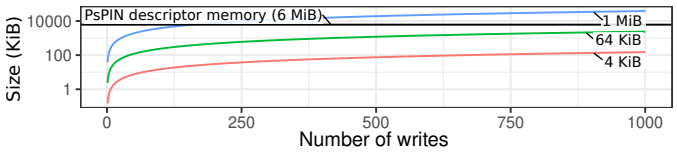


Fig. 4. Worst-case required NIC memory versus number of writes and write sizes. The horizontal line indicates the amount of NIC memory required.

The number of writes served by a storage node at any given time depends on the write size, sPIN handler running times, network bandwidth, and network state (e.g., congestion). We apply Little's law to make a worst-case analysis of the average number of writes being served at any given time by a storage node, assuming a constant flow of fixed-size writes arrive at full bandwidth. Figure 4 shows the required NIC memory to handle a specific number of writes (x-axis) and for different write sizes. In this analysis, we assume that sPIN handlers are not a bottleneck. Detailed data about sPIN handler running times are reported in the following sections.

C. Full-system design considerations

To offload policies to the NIC, the DFS software running on the storage nodes CPU instantiates a PsPIN execution context made of DFS handlers and a NIC memory region storing the DFS state. The execution context is installed into the storage node NIC and matches all incoming RDMA packets. The DFS software running on the CPU can communicate with sPIN handlers by writing to NIC memory (e.g., to update encryption keys, see Section IV). On the other side, handlers can communicate with the DFS software through application-specific event queues (e.g., error conditions, logging information, and other policy-specific events).

The DFS endpoint is similar to RDMA-based DFS. A client must be able to retrieve metadata to build read/write requests (e.g., addressing information for primary and secondary storage nodes). On the storage nodes, requests can be handled either by PsPIN, as we show in this work, or by the DFS software running on the storage node CPU (e.g., by appending requests to RPC command queues via RDMA [27], [28]). For example, the execution context can be configured to steer requests to host memory, bypassing PsPIN, if the SmartNIC is not keeping up with line rate (e.g., overwhelmed by writes requiring erasure coding, see Section VI).

D. Experimental methodology

Until now, we described sPIN, the programming model that best fits the principles of Section II, and discussed the skeleton of the sPIN handlers used to offload the DFS policies. The next sections show how three different DFS policies can be offloaded to the NIC and the respective performance benefits.

The evaluation shown in the following sections is performed through cycle-accurate and functional simulations. We use the PsPIN toolchain to produce cycle-accurate timings of sPIN handlers and the Structural Simulation Toolkit (SST) [29] to simulate multi-node scenarios (e.g., clients and storage nodes). PsPIN handlers are compiled with a PULP-custom version of GCC 7.1.1 (riscv32, `-O3 -flto`). We configure SST to simulate a 400 Gbit/s network, with a maximum transmission unit (MTU) of 2048 B and 20 ns link latency.

IV. Client request authentication

To access data, clients first get metadata information and then directly contact the storage nodes to access data. In this scenario, storage nodes can either trust or authenticate client requests. However, it is not always possible to fully trust clients in shared distributed systems, as a malfunctioning or malicious application/node could bring the DFS to an inconsistent state.

The way client requests are authenticated is DFS-specific. While RDMA provides protection via *rkeys* (i.e., a node can access a remote region only if it has the respective rkey), this offers limited authentication capabilities for client requests. For example, having a single *rkey* for all files would not prevent clients from accessing data they do not own, while having a rkey for each file would require registering $\#files \times \#rights$ memory regions, which can lead to scalability issues [30]. On the other side, validating requests on the CPU of the storage nodes increases overhead, requiring either to buffer the full write before committing it to the storage target (losing the zero-copy benefit of RDMA) or to introduce an additional network round-trip before the RDMA-write. Figure 5 (left) sketches the scenario where CPU-based request authentication is performed. In this case, after the client authenticated itself with the management node and queried the metadata node, it sends a request to the storage node. The request is validated on the CPU of the storage node and, after that, the actual RDMA data transfer can begin.

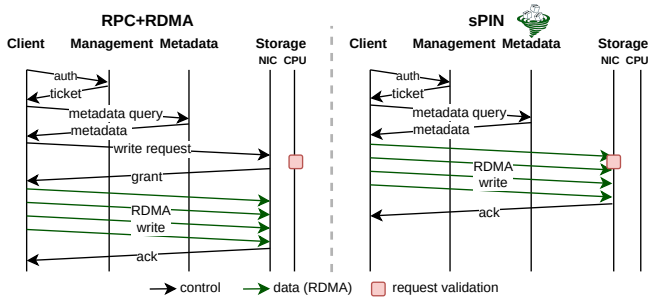


Fig. 5. Authentication strategies. Left: authentication performed on the host CPU (RPC+RDMA). Right: on the NIC with sPIN. The storage node is represented with both NIC and CPU to show where activities are executed.

The sPIN case is shown in Figure 5 (right). The validation is offloaded to the NIC and implemented in the `DFS_request_init` function of Listing 1. This allows the client to directly issue the RDMA write as the request validation will be performed on-the-fly by sPIN handlers, saving the extra RTT required for authenticating the client.

The way requests are validated depends on the threat model: if we trust both the clients and network (e.g., sRDMA [31]), then the ticket can be a plain-text secret given to the client by the management node and checked by the handlers. Nobody can read the ticket from the client or intercept it in the network. DFSs like Orion [9], where storage nodes are accessed by the clients via RDMA, assume this threat model.

If clients are not trusted but the network is, then we need authentication capabilities: the client gets a ticket from the metadata node that contains a capability descriptor. This descriptor determines the operations that the client is allowed to perform and the data it can access. The handlers verify the capability, which is signed with a key shared among DFS services, and check that the requested operation is allowed by the capability [32]. In this work, we assume this threat model. If the network is not trusted, then handlers need to authenticate each network packet in order to exclude tampering.

We analyze the impact of processing packets through sPIN on the overall write latency. The write latency is the time spanning from issuing the write request to receiving the respective write response. In this case, we consider writes where only client request authentication is performed, without additional DFS policies (e.g., replication or erasure coding). We consider the following write protocols:

- **RPC+RDMA**: The client first sends the write request to the storage node via RPC. The RPC handler, executed on the CPU of the storage node, runs DFS policies (e.g., validating the client request) and issues an RDMA read towards the client for getting the data to store.
- **RPC**: The client sends the write request and the data to store to the storage node via RPC. The storage node buffers the data to write, runs DFS policies, and eventually stores the data in the storage target.
- **sPIN**: The client sends the write request and data to store to the storage node in a single RDMA write. Packets are intercepted by sPIN at the NIC of the storage node, where client requests are authenticated by sPIN handlers.

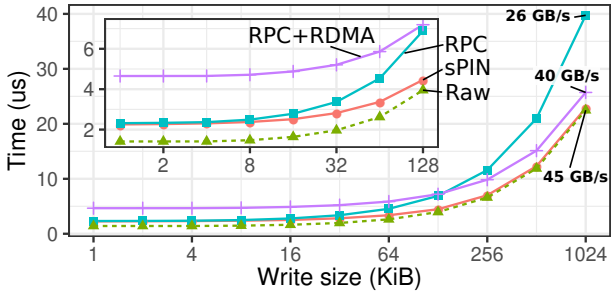


Fig. 6. Write latency with different protocols and write sizes. Raw writes are reported as a (speed-of-light) reference as they do not enforce any policy.

- **Raw writes:** this is our speed-of-light scenario. No DFS policies are enforced on incoming writes. The client issues a single RDMA write to the storage node.

A. Request authentication overhead

Figure 6 shows write latencies for different write sizes. The sPIN handlers validate client requests by checking the capability carried in the write request header (see Section III-A). For RPC and RCP+RDMA, the same validation is performed on the CPU. For large writes, RPC is penalized by the additional memory copy needed to buffer the write while the request is validated. We notice how sPIN introduces small overheads over raw writes, which do not perform validation of incoming write requests. For small writes, sPIN pays the latency of having packets traverse the on-NIC accelerator and validate requests, hence it shows higher overheads (up to 27%) than the raw writes. Figure 7 breaks down the overheads of processing packets in PsPIN: a packet is first copied into the packet buffer (32 cycles for a 2 KiB packet), then scheduled to one of the four processing clusters (2 cycles). At this point, the packet is copied into the cluster-local, single-cycle access memory (43 cycles) and finally scheduled to an idle HPU (1 ns). The DFS handler that validates client requests takes 200 cycles. For large writes, the per-request validation performed only on the first packet of the write becomes negligible, making sPIN-processed writes approach the RDMA (speed-of-light) latency.

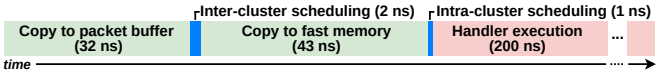


Fig. 7. Packet processing overheads in PsPIN (for 2 KiB packets).

V. Data Replication

This DFS policy replicates data on k storage nodes, where k can be either a global, per-pool, or per-file parameter. In this way, data can survive the failure of $k-1$ storage nodes. To enforce replication, the data written by a client should be propagated to k storage nodes. In the following, we use k to indicate the replication factor, that is the number of nodes on which data is replicated.

Figure 8 shows different strategies for data replication. If the CPU of the storage nodes is involved, then data can be broadcasted among the k storage nodes following different strategies (i.e., broadcast schedules). The optimal broadcast schedule depends on k, the data size, and the interconnection network characteristics [33], [34].

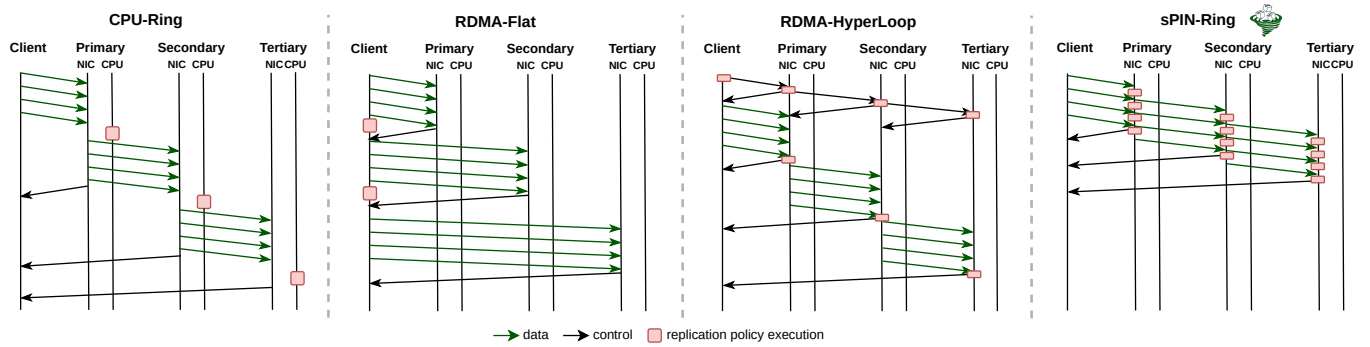


Fig. 8. Data replication policy implemented with different strategies. CPU-Ring involves the CPU of the storage nodes to implement a ring broadcast. With RDMA-Flat, clients write directly to the storage node memory but need to make k different RDMA writes, one for each replica. In the RDMA-Hyperloop case, the RDMA ring needs to be configured with a smaller, pre-defined broadcast for metadata (i.e., for updating RDMA work queue entries to serve the actual data broadcast). With sPIN, the policy is offloaded to the NIC of the storage nodes, that propagate the data on a per-packet basis.

In RDMA-based DFSs, data replication can be performed by either delegating the replication process to the client (RDMA-Flat, i.e., the client performs k different RDMA writes) or with pre-posted RDMA operations that need to be configured by the client out-of-band [35] (RDMA-HyperLoop).

With sPIN, storage nodes can exploit the exposed NIC computing capabilities to distribute replicated data using different broadcasting strategies without involving the host CPU. Specifically, we consider two different replication strategies: *ring*, where each replica sends to another replica only, and *binary tree*, where each replica sends to two children. We show in Figure 8 an example where the sPIN handlers on the storage node propagate the data in a pipeline with the replica nodes virtually arranged on a ring (sPIN-Ring). Since each packet must be forwarded, the broadcast algorithm is implemented in the `DFS_request_process_pkt` function of Listing 1.

A. Broadcast schedules in sPIN

Any broadcast schedule offloaded with sPIN is naturally pipelined on network packets. In particular, for participating in the broadcast, each packet handler needs to (1) identify their position in the broadcast tree (a *ring* can be seen as a unary tree) and (2) send a copy of the data to its children if any.

We require that the write request header (see Figure 3) carries the following information:

- Replication strategy: *ring* or *pipelined binary tree* (pbt);
- Virtual rank: an ID that identifies the node in the tree;
- Replica coordinates: a list of tuples representing the replica nodes. Each tuple includes the network address of the replica and the respective storage address.

This information is contained in the first packet of the write request and needs to be also propagated to the handlers processing the subsequent packets. For this reason, we keep an array of replica coordinates (*coord_array*) in the DFS state stored in NIC memory: this array has a length equal to the arity of the broadcast tree (i.e., 1 if *ring*, 2 if *pbt*) and is filled in by the header handler. The header handler uses the virtual rank, the replication strategy, and the list of replica coordinates to identify the children where to send data. The payload handlers check the DFS state and send the data to all replica coordinates into *coord_array*.

By having the request carrying information on how to progress the communication, we create a client-driven broadcast that does not require the involved storage nodes to keep CPU-initialized stateful data structures to progress it (e.g., to know where to forward the data). We still need a stateful data structure (i.e., the *coord_array*), but that can be initialized when the first packet of the request arrives.

B. Data replication performance

To analyze how per-packet processing impacts data replication performance, we measure the write latency for different data sizes and replication factors. We consider different replication strategies, ranging from fully offloaded with sPIN to CPU based:

- **CPU-Ring.** Replica nodes are virtually arranged in a ring. The CPU of the storage nodes is notified of incoming writes and forwards data to the next node in the (pipelined) ring.
- **CPU-PBT.** Similar to *CPU-Ring* but replica nodes are arranged in a binary tree.
- **RDMA-Flat.** A client issues as many writes as the number of replica nodes.
- **RDMA-HyperLoop.** A client first updates the work-queue-elements (WQEs) on the RDMA NICs of the storage nodes and then starts an RDMA ring broadcast [35]. The WQEs need to be updated in order to define where to write the data on the storage node.
- **sPIN-Ring**. Replicas are arranged in a ring. Data is forwarded by sPIN handlers running on the NIC of the storage nodes. Data is naturally pipelined on network packets.
- **sPIN-PBT**. Similar to *sPIN-Ring* but replica nodes are arranged in a binary tree.

RDMA-Flat and RDMA-HyperLoop do not enforce request validation and fully trust clients. We report data from pipelined executions with optimal chunk size for all non-sPIN strategies implementing a ring and binary-tree broadcast.

1) Write latency: Figure 9 (left) and Figure 9 (center) show latencies of writes with replication factors of k=2 and k=4, respectively. Each plot shows the write latency as function of

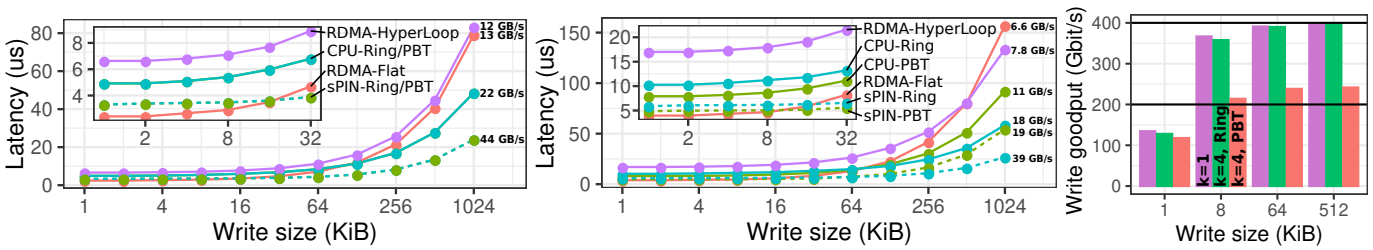


Fig. 9. Left: Write latency with replication factor (k) set to 2 for different write sizes and replication strategies. Center: Write latency with replication factor (k) set to 4. Right: Goodput sustained by a single storage node for different write sizes and offloaded replication strategies.

the write size and for different replication strategies. With k=2, there are no differences between ring and pbt replication strategies as the primary storage node has only one children. In both scenarios, RDMA-Flat provides the lowest latency for small writes (up to 16 KiB), being up to 27% faster than sPIN-Ring for 1 KiB writes and k=2. However, RDMA-Flat would require an additional round-trip per replica to validate write requests. For writes bigger than 16 KiB, the data injection cost paid by the client starts impacting RDMA-Flat performance, making sPIN-based solutions faster. RDMA-Hyperloop is penalized by configuration overheads (i.e., writing of WQEs at storage nodes, see Figure 8). These overheads get better amortized for long replication chains (k=4) and large message sizes. Overall, sPIN-based solutions achieve up to 2x and 2.16x lower latency (w.r.t. the best alternative) for k=2 and k=4, respectively. CPU-based replication strategies are negatively impacted by the cost of moving data to and from host memory.

2) Write goodput: Figure 9 (right) shows the goodput sustained by a single network-accelerated storage node for different write sizes and replication strategies. This is the amount of data per second, excluding headers, that can be ingested by a storage node.

A 1 KiB write fits in a single packet (MTU is 2 KiB) that triggers all handlers (header, payload, and completion). Since each packet triggers three handlers, the overall throughput that sPIN can sustain is limited. With larger writes, the number of packets per write increases, better amortizing header and completion handlers costs (executed once per write). Starting from 8 KiB writes sPIN-Ring line rate. Writes replicated with sPIN-PBT achieve about half the bandwidth because, for each incoming packet, two new packets (i.e., one per children) must be sent out. However, the bandwidth cost is compensated by a lower overall latency of the binary tree for small writes and/or large values of k (see Figure 10 (left)), allowing to achieve

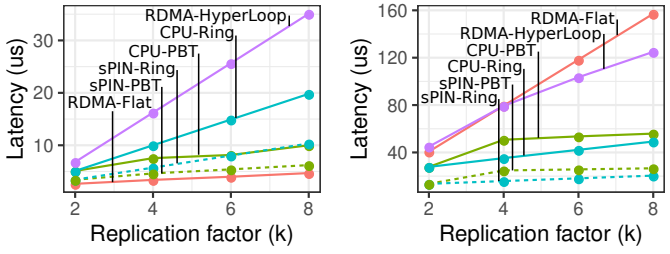


Fig. 10. Write latency for small (4 KiB, left) and large (512 KiB, right) writes with different replication strategies and replication factors (k).

write latencies better or similar to sPIN-Ring in those cases.

3) Varying the replication factor: Figure 10 shows the write latency as function of the replication factor for 4 KiB (left plot) and 512 KiB (right plot) writes. For small writes, RDMA-Flat has the lowest latency for any replication factor. For large writes, the injection cost increases, limiting the performance of RDMA-Flat that grows linearly with k. CPU-based pipelined strategies become more efficient for large replication factors, but they are still penalized by memory-copy overheads. On the other hand, sPIN-based versions are less sensible to k because of the smaller per-packet overheads. As expected, *pbt*-based replication performs better than *ring*-based ones for small writes and large values of k.

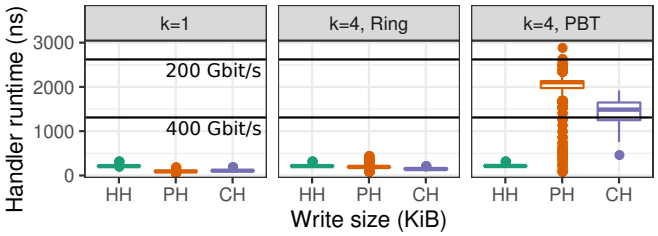


Fig. 11. Handlers running time for writes with replication. Left: no replication; Middle: 4 replicas, sPIN-Ring; Right: 4 replicas, sPIN-PBT.

4) Handlers runtime analysis: Figure 11 shows handler running times for different handler types and replication strategies. We plot lines indicating the cycle budget available to each handler to sustain 400 Gbit/s and 200 Gbit/s line rates with 2 KiB packets. We observe that the duration of handlers for writes without replication ($k = 1$) and writes with ring-based replication (sPIN-Ring) always stays below the 400 Gbit/s budget. The higher running times of sPIN-PBT handlers justify the lower write goodput discussed above. As shown in Table I, the longer duration of these handlers does not correspond to an higher complexity but to a lower number of instructions per cycle (IPC). This is a consequence of the fact that the egress network bandwidth is limited (400 Gbit/s) and that each incoming packet generates two outgoing ones.

Type	Duration (ns)			Instructions			IPC		
	HH	PH	CH	HH	PH	CH	HH	PH	CH
k=1	211	92	107	120	55	66	0.57	0.60	0.62
k=4, Ring	212	193	146	120	105	65	0.57	0.54	0.44
k=4, PBT	214	2106	1487	120	130	82	0.56	0.06	0.06

TABLE I
HANDLER STATISTICS FOR DIFFERENT REPLICATION STRATEGIES.

VI. Erasure coding

The main disadvantage of replication is the storage cost, which is linear in the replication factor. With erasure coding (EC), data is split into k chunks and stored together with m parity chunks. The $k+m$ chunks are normally stored on different storage nodes and failure domains. In case of failure, the missing chunks can be recovered by using the remaining ones. Reed-Solomon [36] codes (RS) are erasure codes employed in a variety of storage systems. RS is maximum distance separable, meaning that a $RS(k,m)$ code can survive up to m corrupt chunks. Also, RS codes are *systematic*: k of $k+m$ encoded chunks are identical to the original k data chunks and can be read without any decoding process.

Figure 12 shows an example for $RS(3,2)$: the encoding matrix (5×3) is multiplied by the data chunks (3×1), obtaining a 5×1 matrix with the 3 data chunks and 2 parity chunks.

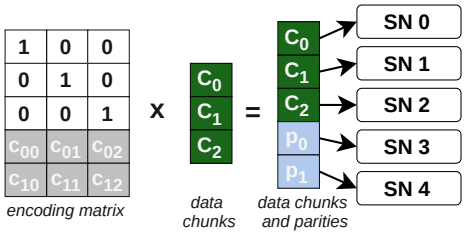


Fig. 12. Encoding an object with $RS(3,2)$. SN: Storage Node.

A. INEC-TriEC

Offloaded EC schemes have been investigated by Shi et al. [12], [37]. In particular, TriEC is a distributed EC scheme where encoding/decoding activities are distributed among multiple storage nodes. With INEC [37], Shi et al. introduce a set of in-network EC primitives to accelerate both encoding and decoding phases of different EC schemes, including TriEC (here defined as INEC-TriEC). TriEC distributes the encoding of data chunks to different storage nodes, generating streams of data to be processed (i.e., data chunks). As sPIN enables application-defined data stream processing on the NIC, it is a good candidate for accelerating TriEC.

Figure 13 (left) shows the erasure coding of a data block in an $RS(2,1)$ scheme with INEC-TriEC. The client sends two different data chunks to two different storage nodes (*SN 0* and *SN 1*), where data is moved to the main memory, as for normal RDMA write (not shown in the figure). Once the transfer is complete, the EC computation is triggered and executed directly on the NIC, reading data from main memory and computing the parity chunks. These are sent to a different storage node (*SN 2* in the example). In general, for each data chunk, m different intermediate parity chunks are generated and distributed to different parity nodes.

B. sPIN-TriEC

With sPIN, the TriEC approach can be re-interpreted with a per-packet vision, allowing to encode data in a streaming fashion, hence avoiding waiting for the full chunk of the data to be received (and copied into host memory) first. We focus on the encoding part since its latency contributes to the write

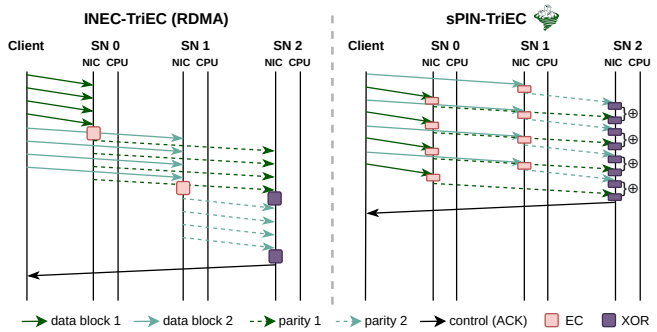


Fig. 13. Diagram of RS(2,1) with INEC-TriEC (left) and sPIN-TriEC (right).

latency if strong consistency is required. The decoding process should preferably be performed offline to not impact write latency. For example, monitoring services can check the status of the storage nodes and start the recovery process if some of them become unreachable [6], [38].

We consider a write request format similar to the one discussed for the replication strategies (see Section V). We assume a write request header carrying enough information to allow the storage node to identify its role in the distributed algorithm. In particular, the write request header carries:

- Erasure coding scheme: *RS-k-m*, where k and m are the number of data and parity chunks, respectively;
- Role: indication of whether this node stores data or parity chunks, determining the actions performed by the handlers;
- Parity node coordinates: the coordinates of the parity nodes.

Replication and erasure coding are normally mutually exclusive: a file is either replicated or erasure-coded. For this reason, the write request header carries a *resiliency strategy* option, telling us whether replication, EC, or no resiliency schemes should be used for this write. This option is followed by either replication or EC parameters.

1) Sending packets: In Figure 13 (right), the client transmits packets to *SN 0* and *SN 1* in an interleaved fashion [39]–[41]. The specific way packets are sent from the client does not influence per-message processing approaches, such as INEC-TriEC, where the full message has to be received anyway to start the encoding. However, in packet-processing settings, the interleaving of the packets allows intermediate storage nodes to work in parallel on different packets, enabling the overlapping of the encoding of following packets with the aggregation (at the parity node) of the already encoded ones.

2) Intermediate encoding: We define two handlers that are executed according to the role played by the storage node in the data encoding: i.e., storing data or parity. In the first case, for each packet, the packet payload gets encoded with the selected RS scheme and m new intermediate parity packets are sent to the respective parity nodes. For encoding, we use the Galois field $GF(2^8)$. While this requires the handlers to scan the payload byte per byte, it allows us to use 256×256-byte lookup table to implement fast Galois field multiplication. The table is copied into NIC memory at DFS-initialization time and is shared between all DFS handlers.

3) Final parities: Nodes selected to store parity chunks need to XOR the k intermediate parity chunks to compute

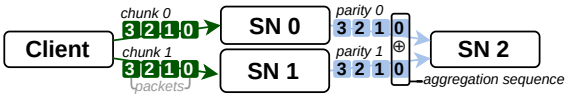


Fig. 14. RS(2,1) packets and aggregation sequence.

the final parity chunk: i.e., $\forall i \in [0, n) : p_i^0 \oplus p_i^1 \oplus \cdots \oplus p_i^{k-1}$, where p_i^j indicates the i-th packet of the message carrying the intermediate parity computed by data node j. We define an *aggregation sequence* as the sequence of packets i coming from the k data storage nodes: $\forall j \in [0, k) : p_i^j$. Figure 14 shows an example where the aggregation sequence for $i = 0$ is marked: there, *SN 2* must aggregate packets coming from both *SN 0* and *SN 1*, in the same order as they are produced.

We keep a pool of accumulators (of size equal to the packet payload size) in the DFS state in NIC memory. The header handler tries to allocate an accumulator from the shared pool. If the pool is empty, then the aggregation cannot be done to the NIC and we fall back to a CPU-based aggregation. Otherwise, a mapping between the aggregation sequence ID i and the accumulator is stored in an on-NIC hash table, allowing the subsequent payload handlers to target the same accumulator with atomic memory operations (i.e, XOR in this case).

Without the interleaved transmission of packets at the client, *SN 0* and *SN 1* would not be able to compute and send intermediate parities in parallel, delaying aggregation at *SN 2* and negatively impacting the overall latency. Additionally, this would increase the time at *SN 2* between the receiving of consecutive packets belonging to the same aggregation sequence, extending the allocation period of accumulator buffers.

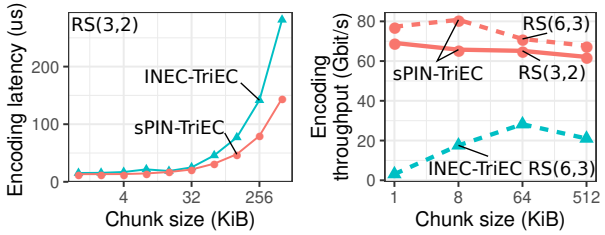


Fig. 15. Encoding latency and throughput.

C. Erasure coding performance

Unlike replication, where the sPIN handlers only forward packets to the next children in the broadcast tree, erasure coding needs to fully process packet data to encode it. This makes it challenging to achieve, e.g., 400 Gbit line rate where, with 2 KiB packets and 32 HPUs, each handler should not last more than ∼1310 ns to not become a bottleneck.

a) Encoding latency: In Figure 15 (left) we compare the write latency of sPIN-TriEC with the one of INEC-TriEC [37]. INEC-TriEC operates on a per-chunk basis: at the intermediate storage nodes, chunks are first written into main memory, then read from the on-NIC EC accelerator to be encoded and sent to the parity nodes. With sPIN, we operate on a per-packet basis and encode packets on the fly without passing by the host's main memory. This allows sPIN-TriEC to have up to 2x lower latency. Since the TriEC results are taken from the INEC paper [37] where a 100 Gbit/s network is used for experiments, we scale our simulated network to the same bandwidth.

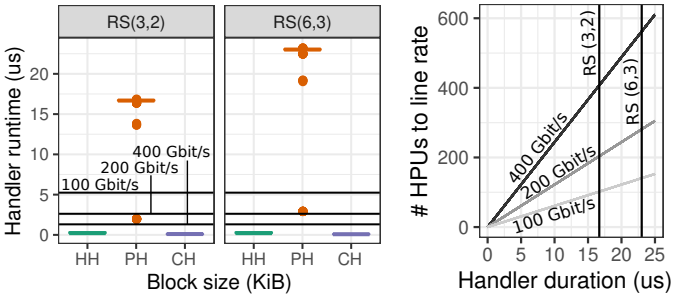


Fig. 16. Left: handler running times for RS(3,2) and RS(6,3). Right: Number of HPUs needed to sustain 400 Gbit/s and 200 Gbit/s (2 KiB packets) for different average handler duration (x-axis).

b) Encoding bandwidth: The encode bandwidth is computed with the same methodology of the INEC paper and common to window-based messaging benchmarks, that is *bandwidth = (size of generated data)/(elapsed time)*. Figure 15 (right) shows the PsPIN-TriEC encoding bandwidth for RS(3,2) and RS(6,3). For comparison, we plot the encoding bandwidth of INEC-TriEC for RS(6,3). sPIN-TriEC achieves up to 29x and 3.3x better bandwidth for 1 KiB and 512 KiB writes, respectively. For small block sizes, INEC-TriEC is penalized by memory copy overheads, which get better amortized with larger blocks. The sPIN-TriEC bandwidth does not directly depend on the block size because it always operates on packets but still experiences a 12% drop in the throughput from 1 KiB to 512 KiB blocks. This is caused by the higher system utilization (i.e., more packets) that leads to more contention on NIC memory.

c) Handlers analysis: Figure 16 (left) shows handler running times for RS(3,2) and RS(6,3). The horizontal lines show the per-handler time budget to sustain different network speeds. Outliers are due to smaller payload packets carrying additional payload (i.e., when the MTU minus the packet header does not divide the block size). The running time of the payload handlers is dominated by the encoding loop, which goes over all the bytes of the packet payload and issues 5 instructions per byte for RS(3,2) and 7 for RS(6,3). Table II reports statistics about handlers running times, number of instructions, and instructions per cycle (IPC). As these handlers are data-intensive, they do not sustain line rate on the selected PsPIN configuration. Figure 16 (right) shows how many HPUs are needed to sustain different network speeds given an average handlers duration time (x-axis). For example, the figure shows how for RS(6,3), a PsPIN configuration with 512 HPUs would allow sustaining 400 Gbit/s line rate for these handlers. Assuming to have a storage backend able to ingest this bandwidth, the modular architecture of PsPIN can be scaled out to sustain these types of workloads at line rate. For example, by increasing the number of clusters in PsPIN, we can increase the number of HPUs without introducing additional load on the per-cluster memories (L1).

Type	Duration (ns)			Instructions			IPC		
	HH	PH	CH	HH	PH	CH	HH	PH	CH
RS(3,2)	215	16681	105	120	11672	35	0.56	0.7	0.33
RS(6,3)	215	23018	82	120	16028	35	0.56	0.7	0.43

TABLE II
HANDLER STATISTICS FOR DIFFERENT EC STRATEGIES.

VII. Discussion

What if sPIN handlers do not run at line rate? To sustain line rate, handlers must process packets within a limited time budget, depending on the line rate, the packet size, and the number of cores. If this is not the case, we have the same scenario as a receiver not being able to receive (i.e., to process in our case) fast enough. This can be mitigated by applying back pressure in the network [42], [43] or by dropping packets.

How is data consistency (e.g., concurrent writes) handled? Clients and metadata services coordinate to guarantee data consistency and avoid that, e.g., multiple clients write to the same part of a file. This coordination phase is part of the control plane, while the actual data access is a part of the data plane. This separation of concerns is typical of several DFSs. Ceph [38] adopts the concept of capability that gives access rights (e.g., write) to clients and is granted by the management servers. To be able to issue a write, a client must first obtain the respective capability, eventually triggering its revocation from a client currently holding it. Similarly, HDFS [6] clients need to be granted permission to write by the name nodes. As this work focuses on the offloading of DFS policies in the data plane, we do not assume any specific coordination protocol.

What happens if packets are lost? We assume a lossless network where packet re-transmission (e.g., in case of data corruption) is performed directly by the NIC. In particular, we assume that once a packet is injected into the on-NIC PsPIN accelerator, the NIC already verified that the packet is not corrupted and that is not a re-transmission. This assumption is satisfied by modern lossless RDMA interconnects [44], [45].

What happens if a storage node fails? A storage node that fails will not send acknowledgments to clients. A client that does not receive an acknowledgment for an ongoing operation after a predefined time threshold can start communicating with the metadata service to signal the failure and start the recovery process. The specific way the recovery is handled is DFS-dependent and not within the scope of this paper.

What happens if a client fails? A client that fails while performing a write operation can leave some dangling state in the NIC of the storage nodes (e.g., `req_table` in Listing 1). We extend PsPIN to associate a cleanup handler with each offloaded execution context. The cleanup handler is triggered by the PsPIN scheduler after an incoming message (i.e., a message for which the header packet has been received but the completion packet is still to come) is inactive for a specified amount of time. For the DFS execution context, the cleanup handler cleans any dangling state and generates an event on the storage node, signaling that a client write has been interrupted and allowing the DFS software to handle the client failure.

How to offload complex protocols? DFSs can implement complex protocols and need large data structures to operate. For example, consensus protocols [46], [47] perform activities such as leader election, log replication, and sharding. While we do not advocate for the offloading of the full DFS logic to the NIC, we note that consensus protocols have been accelerated by extending RDMA primitives [48]. As sPIN provides an RDMA+X paradigm, these primitives can easily be implemented as sPIN handlers, delivering performance benefits without waiting for a their vendor-specific implementation.

Offloading DFS building blocks in the cloud. One challenge of deploying sPIN in the cloud is handling fairness and quality-of-service (QoS) of NIC computing resources for multiple tenants. While there is no multitenancy to enforce in disaggregated storage systems employing dedicated storage nodes, it is necessary to guarantee fairness and QoS in systems exploiting client-local persistent memories. In these systems, network offloading is even more important due to (1) higher operating-system noise and (2) the need to reserve CPU time to dedicate to other tenants.

DFS	RDMA	Policies			Notes
		Aut.	Rep.	EC	
Lustre [4]	👍	👍	✖	✖	RPC+RDMA
IBM Spectrum Scale [5]	✖	👍	👍	👍	
BeeGFS [49]	👍	👍	👍	✖	RDMA compatible
Ceph [38]	✖	👍	👍	👍	
HDFS [6]	👍	👍	👍	👍	RPC+RDMA [50]
Intel DAOS [51]	👍	👍	👍	👍	RPC+RDMA
MadFS [52]	👍	👍	✖	✖	
WekaIO Matrix [53]	👍	👍	✖	👍	
PanFS [7]	👍	👍	✖	👍	RPC+RDMA
OrangeFS [8]	👍	👍	👍	✖	RPC+RDMA [54]
Gluster [55]	👍	👍	👍	👍	
Orion [9]	👍	✖	👍	✖	Client-based replication.
Octopus [10]	👍	👍	✖	✖	RPC+RDMA
FileMR [11]	👍	👍	👍	✖	

TABLE III
DFS CHARACTERISTICS. **RDMA**: SUPPORT FOR RDMA. **AUT.**: CLIENT AUTHENTICATION. **REP.**: REPLICATION. **EC**: ERASURE CODING. 👍: PROVIDED, 👍: PARTIALLY PROVIDED, ✖: NOT PROVIDED.

VIII. Related work

Table III surveys state-of-the-art DFS systems, focusing on two main characteristics: RDMA support and use of different policies (client authentication, resilience via data replication, and resilience via erasure coding).

RDMA- and SmartNIC-enabled storages. With the evolution of networking and storage technologies, classical software storage stack with the operating system and host CPU in the loop have become bottlenecks. This led many distributed file systems to employ RDMA [4]–[11], [51], and let data flow from storage nodes to clients and vice versa without CPU or OS intervention. One-sided RDMA operations are preceded by an RPC communication that validates the file access request and exposes the interested memory region over RDMA. Other approaches [9], [11], utilize RDMA one-sided operations directly, relying on RDMA protection mechanisms.

LineFS [56] accelerates DFS by exploiting NVIDIA BlueField NICs [57]. Differently from LineFS, we target an event-based architecture (i.e., the event is the packet arrival) that is generally simpler (no hardware caches, simple RISC-V cores) but more parallel and specialized for packet processing. Being a DPDK-based approach, LineFS does not implement the user-level principle described in Section II-B.

1RMA [58] proposes a data-center-optimized version of RDMA. They point out that in datacenter-scale storage systems the traditional connection-oriented RDMA approach can face scalability challenges. Our approach is orthogonal to 1RMA as it does not rely on long-lived RDMA connections.

In iPipe [59], an actor-based framework that schedules tasks between SmartNIC and host CPU, the authors discuss a replicated key-value store implementation based on the RDMA-Flat approach discussed in Section V.

Network-offloaded data replication. Hyperloop [35] implements a ring-replication algorithm that can be offloaded to RDMA-capable NICs. It exploits triggered communication offered by Mellanox NICs, that can be used to express happen-before dependencies between pre-posted RDMA operations. Since pre-posted RDMA work requests do not depend on the content of the incoming message that triggered it, a client needs to configure them by remotely writing to their descriptor with an RDMA write, thereby configuring the broadcast ring. Once requests are configured, a client can start the offloaded replication by triggering requests on the first node of the ring. Tailwind [60] implements RDMA-accelerated replication that targets monotonically growing logs and delegates the replication process to the primary storage node, which can then use RPC+RDMA to communicate with replica nodes.

Network-offloaded erasure coding. TriEC [12] proposes a new EC NIC offload strategies that overcome many limitations of current-generation NIC-offloaded schemes for EC. INEC [37] introduces a set of network primitives to accelerate NIC offloaded EC schemes that, similarly to Hyperloop, rely on pre-posted triggered communications that allow reducing EC encoding and decoding latencies.

IX. Conclusion

We show how fully programmable SmartNICs fill the gap between full-RDMA solutions, which provide the best performance for one-sided accesses but do not expose enough compute capabilities to implement DFS policies, and CPU-based solutions, where DFS policies can be fully expressed at the cost of additional memory or network latencies. Moreover, we show how on-NIC packet processing techniques can accelerate replication and erasure coding policies without requiring CPU or OS intervention. These results also demonstrate how, by having fully-programmable SmartNICs, fundamental DFS components can be offloaded to the network without depending on vendor-specific features and deployments.

All in all, these approaches to offload DFS policies can be followed by next-generation DFSs interfacing with fast storage media, where minimizing operation latencies (including policy enforcement) will be fundamental to minimize I/O overheads.

Acknowledgments

This work has been partially funded by the European Projects RED-SEA (grant no. 955776) and DEEP-SEA (grant no. 955606). Daniele De Sensi is supported by an ETH Postdoctoral Fellowship (19-2 FEL-50).

References

[1] A. Sainio, "NVDIMM: changes are here so what's next," *Memory Computing Summit*, 2016.

[2] J. Min, M. Liu, T. Chugh, C. Zhao, A. Wei, I. H. Doh, and A. Krishnamurthy, "Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 106–122.

[3] I. T. Association, "InfiniBand Architecture Specification, Volume 1, Release 1.2," 2004.

[4] P. Braam, "The Lustre storage architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[5] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, no. 19, 2002.

[6] D. Borthakur *et al.*, "HDFS architecture guide," *Hadoop Apache Project*, vol. 53, no. 1-13, p. 2, 2008.

[7] L. Wang, Y. Ma, A. Y. Zomaya, R. Ranjan, and D. Chen, "A parallel file system with application-aware data layout policies for massive remote sensing image processing in digital earth," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1497–1508, 2014.

[8] R. B. Ross, R. Thakur *et al.*, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th annual Linux showcase and conference*, 2000, pp. 391–430.

[9] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A distributed file system for non-volatile main memory and RDMA-capable networks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 221–234.

[10] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an RDMA-enabled distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 773–785.

[11] J. Yang, J. Izraelevitz, and S. Swanson, "FileMR: Rethinking RDMA Networking for Scalable Persistent Memory," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 111–125.

[12] H. Shi and X. Lu, "TriEC: tripartite graph based erasure coding NIC offload," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–34.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[14] J. Kicinski and N. Viljoen, "eBPF hardware offload to SmartNICs: clsbpf and XDP."

[15] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–8.

[16] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA–an open platform for gigabit-rate network switching and routing," in *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. IEEE, 2007, pp. 160–161.

[17] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen, "Corundum: An open-source 100-Gbps Nic," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 38–46.

[18] M. Hennecke, "Daos: A scale-out high performance storage stack for storage class memory," *Supercomputing frontiers*, p. 40, 2020.

[19] R. Rajesh, K. B. Ramia, and M. Kulkarni, "Integration of LwIP stack over Intel DPDK for high throughput packet delivery to applications," in *2014 Fifth International Symposium on Electronic System Design*. IEEE, 2014, pp. 130–134.

[20] J. Liu, C. Maltzahn, C. Ulmer, and M. L. Curry, "Performance Characteristics of the BlueField-2 SmartNIC," *arXiv preprint arXiv:2105.06619*, 2021.

[21] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "sPIN: High-performance streaming Processing in the Network," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–16.

[22] B. Barrett, R. Brightwell, R. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, T. Hoefler, A. Maccabe, and T. Hudson, "The Portals 4.2 Network Programming Interface," 11 2018.

[23] S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beranek, L. Benini, and T. Hoefler, "A RISC-V in-network accelerator for flexible high-performance low-power packet processing," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[24] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "PULP: A parallel ultra low power platform for next generation IoT applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 2015, pp. 1–39.

[25] A. Kalia, D. Andersen, and M. Kaminsky, "Challenges and solutions for fast remote persistent memory access," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 105–119.

[26] T. Talpey, "RDMA extensions for remote persistent memory access," in *12th Annual Open Fabrics Alliance Workshop*, 2016.

[27] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 185–201.

[28] K. Taranov, S. Di Girolamo, and T. Hoefler, "CoRM: Compactable Remote Memory over RDMA," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1811–1824.

[29] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[30] X. Wang, G. Chen, X. Yin, H. Dai, B. Li, B. Fu, and K. Tan, "StaR: Breaking the Scalability Limit for RDMA," in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 2021, pp. 1–11.

[31] K. Taranov, B. Rothenberger, A. Perrig, and T. Hoefler, "sRDMA – Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access," in *Proceedings of the 2020 USENIX Annual Technical Conference*. USENIX, Jul. 2020.

[32] H. Gobioff, G. Gibson, and D. Tygar, "Security for network attached storage devices," CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1997.

[33] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser, "Optimal broadcast and summation in the LogP model," in *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, 1993, pp. 142–153.

[34] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, 1995, pp. 95–105.

[35] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "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*, 2018, pp. 297–312.

[36] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[37] H. Shi and X. Lu, "INEC: fast and coherent in-network erasure coding," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2020, pp. 924–940.

[38] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[39] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: Scalable NIC for end-host rate limiting," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 475–488.

[40] A. Gulati, D. K. Panda, P. Sadayappan, and P. Wyckoff, "NIC-based rate control for proportional bandwidth allocation in Myrinet clusters," in *International Conference on Parallel Processing, 2001.* IEEE, 2001, pp. 305–312.

[41] I. Pratt and K. Fraser, "Arsenic: A user-accessible gigabit ethernet interface," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, vol. 1. IEEE, 2001, pp. 67–76.

[42] IEEE, "RFC 802.1Qbb – Priority-based flow control." [Online]. Available: https://1.ieee802.org/dcb/802-1qbb/

[43] S.-A. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken, "An overview of QoS capabilities in InfiniBand, advanced switching interconnect, and ethernet," *IEEE Communications Magazine*, vol. 44, no. 7, pp. 32–38, 2006.

[44] D. D. Sensi, S. D. Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the Slingshot interconnect," 2020.

[45] Mellanox, "Introducing 200G HDR InfiniBand Solutions," accessed: 2021-04-07. [Online]. Available: https://www.mellanox.com/related-docs/whitepapers/WP_Introducing_200G_HDR_InfiniBand_Solutions.pdf

[46] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[47] D. Ongaro and J. Ousterhout, "The raft consensus algorithm," 2015.

[48] M. Poke and T. Hoefler, "DARE: High-Performance State Machine Replication on RDMA Networks," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'15)*. ACM, 06 2015, pp. 107–118.

[49] J. Heichler, "An introduction to BeeGFS," 2014.

[50] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance RDMA-based design of HDFS over InfiniBand," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–12.

[51] Z. Liang, J. Lombardi, M. Chaarawi, and M. Hennecke, "DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory," in *Asian Conference on Supercomputing Frontiers*. Springer, 2020, pp. 40–54.

[52] J. Lu, B. Du, Y. Zhu, and D. Li, "MADFS: the mobile agent-based distributed network file system," in *2009 WRI Global Congress on Intelligent Systems*, vol. 1. IEEE, 2009, pp. 68–74.

[53] weka.io, "WekaIO Matrix Architecture," 2019, Technical white paper.

[54] J. Wu, P. Wyckoff, and D. Panda, "PVFS over InfiniBand: Design and performance evaluation," in *2003 International Conference on Parallel Processing, 2003. Proceedings.* IEEE, 2003, pp. 125–132.

[55] A. Davies and A. Orsaria, "Scale out with GlusterFS," *Linux Journal*, vol. 2013, no. 235, p. 1, 2013.

[56] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 756–771.

[57] NVIDIA, "NVIDIA BlueField," accessed: 2021-05-20. [Online]. Available: https://www.nvidia.com/en-us/networking/products/data-processing-unit/

[58] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. Martin, M. McLaren, P. Chandra, R. Cauble *et al.*, "1RMA: Re-envisioning remote memory access for multi-tenant datacenters," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 708–721.

[59] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "iPipe: A Framework for Building Distributed Applications on Multicore SoC SmartNICs," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[60] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes, "Tailwind: fast and atomic RDMA-based replication," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 851–863.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

The experiments of this paper are based on cycle-accurate and functional simulations.

We provide a docker image that includes all software needed to reproduce the results presented in the paper. The image includes the following main components: - SST simulator. This is a modified version of the Structural Simulation Toolkit that has been adapted to inteface PsPIN cycle-accurate simulations. - PsPIN (SST version): this is a version of the PsPIN hardware and simulation infrastracture that has been adapted for being integrated into SST. - PsPIN (standalone): this is a standalone version of PsPIN. We use this for simulations outside SST (e.g., sigle-node throughput).

Additionally, we include scripts to run simulations and produce plots for all figures and tables included in the paper. We now describe the steps needed to start the container and reproduce each figure and table.

A detailed README.md, describing how to reproduce each figure, is included in the artifact.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://doi.org/10.5281/zenodo.6461589`

Artifact name: Building Blocks for Network-Accelerated Distributed File Systems - AD/AE

Reproduction of the artifact with container: We ran the scripts included in the container on a machine with the following characteristics: "‘ $ lscpu Architecture: x86_64 CPU op-mode(s): 32-bit, 64-bit Byte Order: Little Endian Address sizes: 44 bits physical, 48 bits virtual CPU(s): 64 On-line CPU(s) list: 0-63 Thread(s) per core: 2 Core(s) per socket: 8 Socket(s): 4 NUMA node(s): 4 Vendor ID: GenuineIntel CPU family: 6 Model: 46 Model name: Intel(R) Xeon(R) CPU X7550 @ 2.00GHz Stepping: 6 CPU MHz: 2075.744 BogoMIPS: 3990.11 Virtualization: VT-x L1d cache: 32K L1i cache: 32K L2 cache: 256K L3 cache: 18432K NUMA node0 CPU(s): 0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60 NUMA node1 CPU(s): 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61 NUMA node2 CPU(s): 2,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62 NUMA node3 CPU(s): 3,7,11,15,19,23,27,31,35,39,43,47,51,55,59,63 Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 x2apic popcnt lahf_lm pti ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid dtherm ida

$ grep MemTotal /proc/meminfo MemTotal: 1056847416 kB (1 TiB)

$ uname -a Linux einstein 4.19.0-6-amd64 #1 SMP Debian 4.19.67-2 (2019-08-28) x86_64 GNU/Linux

$ docker -v Docker version 20.10.6, build 370c289 "‘

The time taken to run all the scripts was about 24-30 h.