2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00026

NVMe-CR: A Scalable Ephemeral Storage Runtime for Checkpoint/Restart with NVMe-over-Fabrics

Shashank Gugnani
The Ohio State University
gugnani.2@osu.edu

Tianxi Li
The Ohio State University
li.9443@osu.edu

Xiaoyi Lu
University of California, Merced
xiaoyi.lu@ucmerced.edu

***Abstract*—Emerging SSDs with NVMe-over-Fabrics (NVMf) support provide new opportunities to significantly improve the performance of IO-intensive HPC applications. However, state-of-the-art parallel filesystems can not extract the best possible performance from fast NVMe SSDs and are not designed for latency-critical ephemeral IO tasks, such as checkpoint/restart. In this paper, we propose a powerful abstraction called `microfs` to peel away unnecessary software layers and eliminate namespace coordination. Building upon this abstraction, we present the design of `NVMe-CR`, a scalable ephemeral storage runtime for clusters with disaggregated compute and storage. `NVMe-CR` proposes techniques like metadata provenance, log record coalescing, and logically isolated shared device access, built around the `microfs` abstraction, to reduce the overhead of writing millions of concurrent checkpoint files. `NVMe-CR` utilizes high-density all-flash arrays accessible via NVMf to absorb bursty checkpoint IO and increase the progress rates of applications obliviously. Using the ECP CoMD application as a use case, results show that our runtime can achieve near perfect ($>$ 0.96) efficiency at 448 processes and reduce checkpoint overhead by as much as 2x compared to state-of-the-art storage systems.**

***Index Terms*—Checkpoint/Restart, NVMe, NVMf, Exascale**

I. Introduction

With enormous compute power, upcoming exascale systems [1] will bring with them crippling frequencies of system failure. Prior work estimates that their mean time between failure (MTBF) will be less than 30 minutes [2]. Exascale applications must protect themselves from unavoidable failures by checkpointing internal state to persistent storage. System-level checkpoint dumps on existing multi-petaflop systems have been shown to have significant overhead [3]. This problem will be exacerbated on exascale systems as not only will checkpoint time increase, but checkpoint frequency will also increase to account for the decrease in MTBF. The IO runtime must bear the burden of storing vast amounts of data in as little time as possible. Consequently, the storage components of exascale systems must be redesigned.

Newly introduced NVMe SSDs based on flash and 3D-XPoint memory offer unprecedented performance and concurrency. For example, new SSDs offer write bandwidth up to 2.5GB/s [4], an order of magnitude faster than SATA SSDs. These devices are ideal for use in HPC systems to build high density storage arrays. These arrays can be stacked together to build a disaggregated storage cluster. With the introduction of the NVMe-over-Fabrics (NVMf) standard [5], low latency remote access to these arrays can be effectively provided. The NVMf standard can take advantage of fast Remote Direct Memory Access (RDMA) enabled networks in HPC systems to reduce the network overheads of remote access. Guz *et al.* [6] have shown that at the application level, NVMf only has ∼10% overhead compared to local IO.

*This work was supported in part by NSF research grant CCF #1822987.

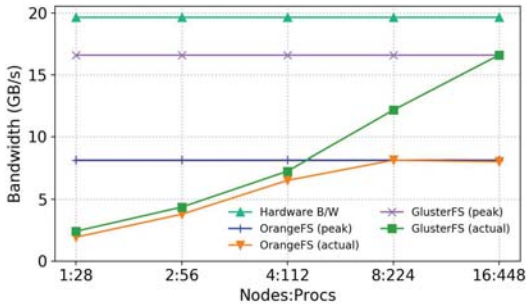

Figure 1: Weak scaling checkpoint bandwidth on local cluster compute nodes (see §IV-A for configuration). Note the gap between the OrangeFS and GlusterFS peak and available hardware IO bandwidth.

A. Challenges and Motivation

In a disaggregated HPC cluster setup, the most common form of storage runtime provided is a Parallel File System (PFS), such as Lustre [7]. In recent years, several distributed filesystems [8], [9], [10], [11], [12], [13] have been proposed to alleviate the bottlenecks in PFS. However, we find that these systems are still not ideal for highly concurrent checkpoint IO. To demonstrate the shortcomings of existing filesystems, we measure the sustained bandwidth of checkpoint IO on NVMe SSDs while varying the number of concurrent application processes.

Figure 1 shows the checkpoint bandwidth for OrangeFS [9] and GlusterFS [11] with the ECP CoMD application [14]. At best, OrangeFS and GlusterFS can only achieve 41% and 84% of the peak hardware bandwidth, respectively. There are two primary reasons for this. First, these storage systems overlay multiple software layers over POSIX filesystems which decrease the peak attainable bandwidth. Second, these filesystems expose significant overhead at high concurrency, which is because the strict POSIX semantics require `open` and `creat` syscalls to be atomic. The need for operation atomicity and metadata consistency requires complicated distributed synchronization mechanisms which suffer from scalability limitations [15], [16]. We further experimentally verified the performance impact of these two problems (shown in Figures 7(c) and 8(b)). Note that at lower process counts,

978-1-6654-4066-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00026

GlusterFS is unable to deliver their peak bandwidth. This is because GlusterFS uses consistent hashing to distribute files between storage devices which has high standard deviation of load under low concurrency, as shown in [17].

There has been work [18], [19] to alleviate these bottlenecks by reducing or eliminating the need for synchronous namespace (or directory hierarchy) access. Unfortunately, these works are unable to extract the best possible performance from fast NVMe devices. Control and data operations have to trap into the OS and go through multiple software layers, negating the benefits of using low latency NVMe devices. As such, these storage systems are only suitable for storing long-lived input and output data and not latency critical ephemeral checkpoint data. The characteristics of checkpoint IO are different. The data is ephemeral and the IO bandwidth required depends on the job scale. Therefore, for storing checkpoint data, we need a storage runtime which can be configured during a job's runtime based on its load factor. The runtime itself must be ephemeral and should terminate with the job. To the best of our knowledge, no storage runtime is available in the HPC community yet, which can offer direct access to storage using NVMf as well as synchronization-free control and data planes. Therefore, there is a clear need to design a coordination free storage runtime for checkpoint/restart, which can provide low latency direct access to remote SSDs using NVMf. In this paper, we attempt to design such a runtime. An ephemeral storage runtime for checkpoint data must satisfy three requirements: (1) full utilization of available bandwidth, (2) high resiliency of data, and (3) support for large number of concurrent clients.

B. Contribution

In this paper, we present the design of `NVMe-CR`, a scalable ephemeral userspace storage runtime for storing checkpoint data with NVMf. The design of `NVMe-CR` can be summarized by three main principles: unprivileged userspace direct device access for low latency, uncoordinated control and data planes for scalability, and logical isolation with shared device access for concurrency and security. To achieve these, we manifest our vision as a design template for coordination-free filesystems, called `microfs`. `Microfs` is an abstraction which allows for synchronization-free control and data planes and userspace access to storage devices. Building upon this abstraction, `NVMe-CR` enables userspace direct access to storage devices and handles data distribution and load balancing on a per-job basis. The runtime mirrors the lifespan of the application and can be configured to run on any number of storage devices.

Evaluation with the ECP CoMD application shows that `NVMe-CR` can achieve greater than 96% efficiency at 448 processes. Our analysis shows that current-generation filesystems (such as Lustre [7], GlusterFS [11], and OrangeFS [9]) are unable to handle the highly concurrent nature of application-level checkpointing. This leads to sub-optimal progress rates[1], decrease in efficiency, and increase in total cost of ownership (TCO) of the system. Compared to these state-of-the-art systems, NVMe-CR can reduce checkpoint overhead by as much as 2x. Furthermore, through increased efficiency and low software overhead, our runtime can lower the required hardware IO bandwidth (and the overall TCO as a consequence) by as much as 2x.

[1]Application progress rate is an important metric for measuring the efficiency of large scale systems. It is defined as the ratio of application time spent in compute to total application time.

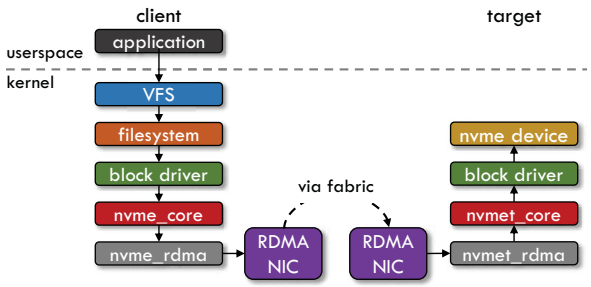

Figure 2: Existing application-oblivious approach to access remote storage using NVMf. The entire data plane lies in the kernel space.

The rest of this paper is organized as follows. We discuss the background and related work in §II and present the `NVMe-CR` design in §III. §IV presents a comprehensive experimental analysis of our designs and §V concludes the paper.

II. Background and Related Work

We present recent trends in NVMf-based storage systems and related work on optimizing checkpoint IO in this section.

A. NVMf-based Storage and Application-Level Checkpointing

The NVMe-over-Fabrics (NVMf) standard is an extension of the NVMe standard to allow remote access to SSDs using fast RDMA-enabled networks. The ability to provide low latency access to remote SSD devices with NVMf support is a great incentive to deploy high-density all-flash arrays on next-generation HPC clusters. While there have been several works [8], [20] which utilize NVMf to speed up data storage for Big Data applications, the same is not fully embraced yet for HPC applications requiring checkpoint/restart (C/R). For applications to transparently take advantage of NVMf, the common approach is to leverage existing kernel modules. Figure 2 shows the flow of the kernel based approach. From the application perspective, the remote access is transparent and it operates under the illusion that the filesystem is backed by a local SSD. Internally, the kernel handles the remote access via RDMA using the `nvme_rdma` and `nvmet_rdma` modules. As is clear from the figure, the entire data plane resides within the kernel, which negates the benefits of user-level RDMA protocols, such as low latency and high bandwidth.

Application-level checkpointing is a common approach taken to provide insulation from inevitable system failures. In a disaggregated storage setup, NVMf can be used to reduce checkpoint overhead. However, to take full advantage of this new standard, a new approach is required which is not only transparent but is also able to expose the raw NVMf performance to end applications. This approach must reconsider the need to trap into the OS for every operation, instead providing unprivileged userspace storage access. It is

thus important to design *userspace* storage runtimes which can take advantage of these resources and transparently improve application performance.

B. Related Work

There have been several works which focus on reducing the overheads of application checkpointing. CRUISE [21] is a userspace filesystem backed by DRAM to enable fast checkpoints. Zest [22] uses a log structured design with a burst buffer layer to reduce checkpoint overhead. CRFS [23] improves checkpoint IO by aggregating small IO operations into larger operations which are written asynchronously to disk. PLFS [24] is another filesystem aims at optimizing checkpoints which follow the N-1 pattern. Other works like PapyrusKV [25], UnifyCR [12], and BurstFS [13] present a burst buffer design using node local storage to accelerate C/R IO as opposed to NVMe-CR that is targeted towards a disaggregated setup. Storage systems like Hermes [26], DDN IME [27], Cray DataWarp [28], GlusterFS [11], and OrangeFS [9] overlay multiple software layers on kernel filesystems to access data. While these works have improved checkpoint overhead, they suffer from two basic limitations. First, these filesystems still suffer from the use of POSIX filesystems to access devices. Direct userspace access to devices via NVMf, as achieved by our proposed NVMe-CR, is not supported. Second, serialization of metadata operations and synchronization between clients could prevent applications from fully utilizing available IO bandwidth. BSCFS [10] and Crail [8], [29] do support an NVMf-based data plane, however, both require applications to be modified to use their specific non-POSIX API. BLCR [30] is an orthogonal approach for system-level C/R as opposed to application C/R, which we focus on.

NVMe-CR's microfs abstraction is most related to the design of DeltaFS [18]. DeltaFS does not expose a single namespace, but a collection of snapshots that can be used by applications to construct their own namespace view. This provides the ability to execute large numbers of parallel metadata operations with minimal coordination. Microfs extends this idea to completely remove coordination requirements for both metadata and data operations. It assumes the complete absence of a global namespace and partitions SSDs between processes to achieve logical isolation. To reduce the amount of data to be checkpointed, techniques such as *incremental checkpointing* [31], *cooperative checkpointing* [32], *multi-level checkpointing* [33] and *data compression* [34] have been proposed. While these approaches reduce checkpoint overhead, they still rely on existing inefficient IO subsystems. Thus, these works are complementary to the designs proposed in this paper and can be combined for improved performance.

III. NVMe-CR Design

In this section, we present the design of NVMe-CR.

A. The microfs Abstraction

We introduce micro filesystem (or simply microfs) as a powerful design template for ephemeral distributed filesystems that wish to provide the minimal functionalities expected of a storage system. Microfs is designed to peel off the unnecessary software layers that hinder performance and allow applications to directly access storage devices. As opposed to conventional kernel filesystems, microfs runs purely in userspace which allows it to bypass the kernel virtual filesystem (VFS) and block driver, eliminating the need to trap into the kernel for every operation. Furthermore, microfs abstains from providing a shared namespace, instead only providing a private one for each application process. It is this choice which obviates the requirement to synchronize across all microfs instances, a common challenge for user-level filesystems. With this observation, we present microfs as a high-performance alternative for storing ephemeral application data.

To attain the goals of microfs, we formulate the following design principles.

Principle 1: Direct userspace access to storage devices. To enable unprivileged userspace access to devices, microfs uses the vfio kernel module. IO operations can then be submitted using memory mapped IO and completed by issuing DMA operations purely in userspace. This allows for bypassing the kernel VFS and block driver while also providing fine-grained control over the IO pipeline. We expect microfs instances to implement a run-to-completion request pipeline (by avoiding locks and using polling instead of interrupts) to reduce IO latency.

Principle 2: Maintaining storage device integrity. Storage devices are shared across several microfs instances. Integrity is maintained by logically and physically partitioning the device between the instances. This partitioning is done at initialization time using a shared communication runtime. We create group communicators for processes sharing hardware to enable coordinated access to storage devices. Once the partitioning is complete, each runtime instance can access its portion of the device without any need for coordination. In this manner, unprivileged direct access to storage can be provided efficiently.

Principle 3: Synchronization-free control and data planes. Data plane operations are designed to be synchronization-free by allocating a separate hardware IO queue for each microfs instance. This enables parallel IO by exploiting the large number of queues in modern storage devices. For example, the Intel Optane P4800X SSD controller can support up to 32 hardware queues. The use of a single IO queue per instance guarantees that IO operations are completed in the order they are received. Control plane (metadata) operations are synchronization-free since no inter-microfs coordination is required, by definition.

Principle 4: Data and metadata durability. Traditional kernel filesystems rely on an OS-level cache to buffer IO, journaling both data and metadata to maintain durability. Instead, microfs writes data directly to device-level RAM and transparently leverages device capacitance to guarantee durability. Metadata consistency is maintained using lightweight operation logging. Furthermore, the runtime periodically checkpoints internal filesystem states to the storage device to prevent

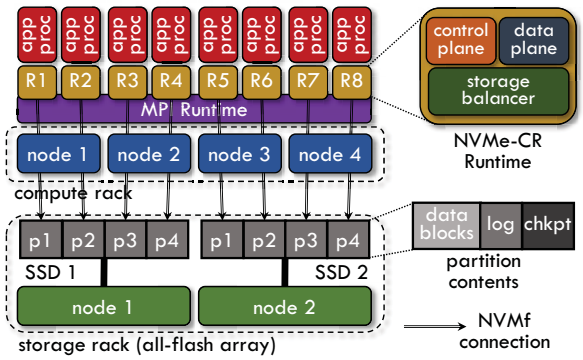

Figure 3: NVMe-CR Runtime Architecture. Each runtime instance directly accesses its own remote SSD partition via NVMf.

the log from growing without end.

B. Architecture

NVMe-CR is a storage runtime designed for scalable C/R capabilities using the emerging NVMf protocol. NVMe-CR is built upon the microfs abstraction to enable low latency IO. The goal of NVMe-CR is to provide C/R support for systems with disaggregated storage. NVMe-CR does not provide a global namespace but only private per-process namespaces. This is a conscious design decision to eliminate synchronization. Figure 3 shows the architecture of NVMe-CR. Each application process runs its own storage runtime which is mapped to a single partition of a remote SSD, connected using NVMf. Each runtime instance implements three critical functionalities – the control plane, data plane, and storage balancer. The control plane is responsible for creating and storing metadata of files and directories. The data plane provides a block device like interface to access the remote SSD partition using NVMf. The storage balancer partitions available storage devices between compute processes and provides a conflict free many-to-one mapping. These three components work in tandem to expose an ephemeral storage runtime to disaggregation oblivious applications. The sub-sections that follow provide a detailed description of the working of these components.

C. Application Obliviousness

One of the primary goals we want to achieve is portability, i.e., enhancing the storage runtime without application modification. We use the common approach of symbol interception provided by the GNU ld linker to achieve this. We intercept all the standard POSIX IO library calls and redirect them to the NVMe-CR runtime. NVMe-CR implements all of the IO library calls while remaining purely in userspace by following the design principles of microfs. For management of the NVMe-CR runtime, we also intercept the MPI_Init and MPI_Finalize calls. Runtime initialization and finalization is handled by these wrappers. Internally, we leverage the MPI runtime for coordination between multiple instances as well as for identification purposes. It should be noted that coordination is only necessary in the initialization routine. Subsequent control and data plane requests are not required to synchronize with other runtime instances. By using the symbol

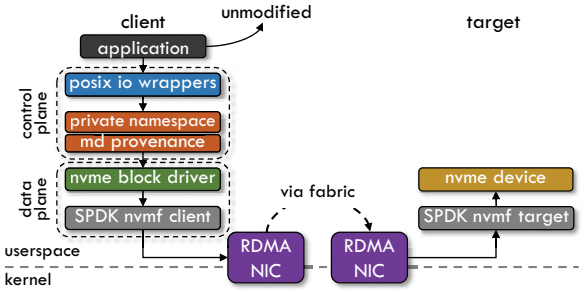

Figure 4: NVMe-CR application-oblivious approach to access remote storage using SPDK and NVMf. The entire data plane lies in userspace, a stark contrast from the approach in Figure 2.

interception method, we can efficiently separate the filesystem API from remote data storage and run unmodified application binaries over NVMe-CR.

D. Data Storage using NVMf – Data Plane

NVMe-CR can transparently leverage NVMf as the data plane conduit to access the high density storage arrays. Currently, our implementation uses the RDMA transport for data exchange. Figure 4 shows how the data conduit is built. Note the direct contrast between this approach and Figure 2. The entire software stack is shifted to userspace from the kernel. To enable userspace access to remote SSDs via NVMf we use Intel's SPDK library [35]. We chose SPDK as opposed to other systems like NVMeDirect [36], and libaio [37] because it is the current state-of-the-art solution. Further, it has negligible software overhead and its NVMf server is multi-tenant. SPDK NVMf server daemons are deployed on each storage node for handling client NVMe requests. SPDK NVMf clients, embedded within the NVMe-CR runtime are responsible for communication with server daemons. In this manner, all data plane operations are internally translated into NVMf requests and handled by the NVMe-CR runtime.

Data Durability. NVMe-CR eschews buffering write requests, instead writing data directly to internal device-level RAM, if available. Otherwise, it is directly written to flash memory. Flash devices with RAM usually support enhanced power-loss data protection [38]. In the event of power failure, device capacitors will safely flush volatile data to non-volatile flash memory. Writing to device RAM does not limit the overall data storage capacity but only improves performance in cases where data fits within device RAM. If data does not fit in RAM, performance gets limited to SSD bandwidth. By avoiding data buffering, we assure that data is always persistent and can survive temporary power failures. This design choice was made based on the observation that buffered IO reduces overall application progress rate.

E. Metadata Management – Control Plane

The control plane is responsible for metadata management, including block allocation, data and metadata consistency, and exposing a POSIX compatible interface. Its goal is to eliminate the need for complex distributed synchronization and minimize network IO. NVMe-CR's control plane design

has several advantages. First, by exposing only private namespaces, metadata operations never have to coordinate across processes. In contrast, other systems must use distributed locking algorithms for each metadata operation to maintain a consistent global namespace. Second, metadata is entirely maitained in DRAM with lightweight operation logging used for consistency and durability. The only network IO involved is for reading/writing file data and writing compact log records. Other systems must trasmit additional data over the network (such as inodes and large sized physical log records), reducing overall system efficiency. In addition, the control plane uses a combination of several techniques and designs to maximize performance. These are explained in detail below.

Hugeblocks. For space management, we divide the SSD into *blocks*, the smallest unit of storage allocation. Given that checkpoint files are several MBs, if not GBs, we allow files to be managed in large block sizes. We call these *hugeblocks* because of their similarity to *hugepages*. In this paper, we use a *hugeblock* size of 32KB as opposed to kernel filesystems which support block sizes only up to 4KB. We use a circular block pool for $O(1)$ *hugeblock* allocation. The use of *hugeblocks* significantly lowers the amount of information that must be kept to track file blocks, which not only reduces the space overhead but also makes the process of block allocation faster. Further, we submit NVMf IO requests in *hugeblock* units as well, which allows us to attain peak hardware bandwidth regardless of the number of clients sharing an SSD. The reason for this is that NVMe SSDs internally break up large IO requests into several smaller (usually hardware block sized) requests which can then be split across the available flash channels for highly parallel IO. The entire SSD bandwidth can then exploited even if there is only a single client accessing the device.

POSIX Semantics. By using SPDK for remote IO, we can indeed bypass the kernel, but we need to provide a POSIX filesystem like interface to applications. `NVMe-CR` implements wrappers for commonly used POSIX IO syscalls. For providing a filesystem like runtime with POSIX semantics, we borrow several conventional filesystem concepts and techniques, such as *inodes* to store file metadata and *directory* files to store directory entries. The control plane handles failure recovery using a write-ahead log. All metadata operations executed during the following functions are logged: `mkdir`, `open`, `write`, and `unlink`. The log is flushed before a subsequent operation is processed. Since we do not buffer writes and we journal metadata operations for `open` and `write`, our runtime provides stronger data durability guarantees than required by POSIX. As a consequence, metadata will always be consistent, even with unexpected failures. This is an important property because it guarantees that a completely written checkpoint file will never hold corrupted data and can safely be used for recovery.

Per-process Private Namespace. If we take a look at common checkpointing patterns used by applications, we find that two patterns are prevalent – N-1 and N-N [24]. In the N-1 pattern, processes write to a single shared file, whereas in the N-N pattern each process writes to a unique file. Recent work [39] has estimated that 90% application runs use the N-N pattern. Due to this reason, the designs proposed in this paper are specifically targeted towards the N-N pattern. The concurrent creation of millions of files is a scalability challenge for filesystems. The choice of private per-process namespaces in `NVMe-CR` allows us to overcome the scalability bottlenecks with the N-N pattern. Each runtime instance is only tasked with the creation of a single file per checkpoint. Each runtime instance exposes a private root directory which is not accessible to other processes. The root directory is a file which resides on the remote SSD partition for the process. Since the root directory files are independent for each process, the namespace exposed by them are private. The directory hierarchy is constructed using a set of directory files indexed by a DRAM resident B+Tree. The B+Tree contains mappings of directory and file names to their root inode. Any file manipulation operation (`creat`, `open`, `unlink`, `read`, or `write`) is handled exclusively by the `NVMe-CR` runtime associated with the originating process. Each runtime not only handles the maintainance of the private namespace but also block and inode allocation for files. In this manner, no coordination is required for any operation.

Metadata Provenance. To ensure that checkpoint files are available despite application failure, we must ensure that metadata is safely made durable. A simple approach could involve storing metadata on the remote SSDs and updating it on each `fsync` or `write` call. This would lead to unnecessarily high remote IO operations over NVMf. To reduce the metadata overhead imposed on each runtime instance, we allow the control plane to store metadata (inodes, block pool, and B+Tree) locally, within the memory of compute nodes. To guarantee metadata durability and consistency, we journal filesystem metadata operations using a compact operation log stored on remote SSDs. Each syscall that modifies an inode needs to be logged. Only the syscall type and its parameters need to be added to the log. The use of operation logging significantly reduces the amount of data that must be sent over network to the remote SSDs. We call this approach metadata provenance because we store every operation in the log, giving us the ability to track fine-grained changes to metadata. During recovery in the event of a crash, the runtime reconstructs metadata by replaying operations recorded in the log. An in-memory B+Tree is used to keep mappings of filenames to their inodes allowing fast lookups of files. The state of the B+Tree can also be reconstructed upon recovery from a crash.

To limit the size of the log, the runtime checkpoints internal DRAM state (which includes the inodes, block pool, and B+Tree) to a reserved region on the remote SSD. To ensure that checkpoints do not affect user IO requests, the checkpoint process is overlapped with the application compute phase. In this manner, the checkpoint process is completed in the background using a dedicated checkpoint thread without affecting application performance. The background thread can exactly determine when the application checkpoint process is complete by monitoring the number of open files. When

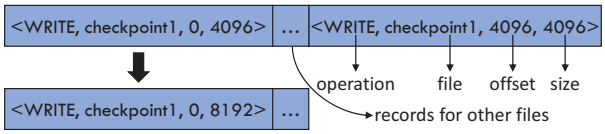

Figure 5: Log Record Coalescing

the number of open files is zero and the number of free log records is below a predefined threshold, the background thread kickstarts the process of checkpointing internal volatile state. Further, the checkpoint process is designed to be atomic. Log records are only discarded once the checkpoint is complete. A failure during checkpoint will not affect the durability and consistency of data.

Log Record Coalescing. To reduce the number of log records written, we propose a technique called log record coalescing. In this technique, we take advantage of the sequential nature of checkpoint IO to combine near-adjacent log records as long as they represent consecutive writes to the same checkpoint file. Figure 5 shows how this process works. Instead of adding new log records for each write, we can simply update the log record for the previous write. We use a sliding window to find the log record for the previous write and update it accordingly. In this manner, the log fillup rate is significantly lowered, which means that filesystem state does not need to be checkpointed frequently to clean log space. In addition, the number of records that must be replayed on recovery is significantly lowered too, which provides near-instantaneous recovery of the `NVMe-CR` runtime itself.

F. Load-aware IO – Storage Balancer

The storage balancer is responsible for both allocation of storage nodes for a job and balancing IO load between available storage devices. `NVMe-CR` considers two vital factors while distributing data between remote SSDs – load balancing and fault-tolerance. Load balancing is important to ensure that we utilize all of the available IO bandwidth and all processes get an equal distribution of the bandwidth. Fault-tolerance is another important factor because we need to place checkpoint data in a separate failure domain than what the process is in. Otherwise, it is likely that failure of a process coincides with loss of its checkpoint data. To account for both of these factors, `NVMe-CR` uses a novel data distribution algorithm.

First, we identify the failure domains for each node by using the network topology. Nodes which share hardware are placed in the same domain. For example, all nodes within a rack and all nodes sharing a power distribution unit are placed in the same domain. Next, we create partner failure domains, such that nodes in both partners are in separate failure domains. For each failure domain, we create a list of partner domains sorted by the number of switch hops between them. Finally, we create a mapping of processes and storage nodes, such that their respective failure domains are partners and the load on each storage node is as equal as possible. We construct the mapping using a greedy algorithm to minimize communication cost. Storage devices for a job are allocated on the closest (fewest hops away) available partner domain. Processes within a job

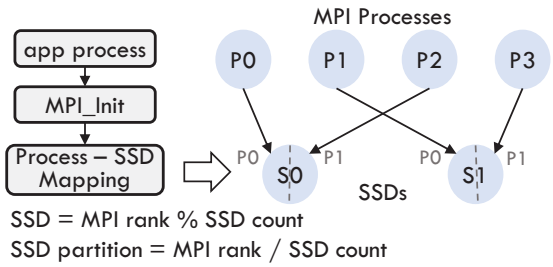

Figure 6: NVMe-CR Storage Allocation Process

are assigned to the allocated SSDs in a round robbin manner to achieve load balancing. Once this mapping is defined, we create an MPI communicator for all processes sharing an SSD, called `MPI_COMM_CR`. The SSD is then partitioned between processes in the associated communicator. Each process gets a contiguous segment of the SSD based on the its rank and the communicator size. Figure 6 shows how the available storage is partitioned and allocated to MPI processes.

Security Model. To ensure device integrity despite userspace access, we rely on the namespace feature in the NVMe standard. All SSDs are divided into at least two namespaces. The job scheduler assigns storage to jobs at the granularity of an NVMe namespace. If there are no free namespaces, new ones are created from unused SSD space. Although the number of namespaces supported by each SSD is limited, the number of concurrent jobs an SSD can support is only limited by its bandwidth. This is because a few concurrent jobs can easily saturate its bandwidth. This approach allows SSDs to be shared between applications while relying on the isolation property of namespaces to maintain security.

This enforcement requires integration with the cluster job scheduler. In practice, we do not anticipate this to be a problem. For example, by using Slurm's generic resources plugin [40], we were able to support this design on our cluster easily. Further, `NVMe-CR`'s control plane acts as a trusted intermediary between the application and SSD. The control plane performs access control checks for file IO so that POSIX permissions are respected and unauthorized users cannot corrupt or read checkpoint data.

For each job, the user must specify the number of storage devices required for checkpoint data. This number can be determined by ensuring that the process:SSD ratio is in the range 56–112. This is based on our experimental results showing that at this ratio NVMe SSD bandwidth is utilized to its maximum. The storage balancer works along with the job scheduler to allocate SSDs based on allocated compute nodes and the network topology, as explained earlier. The load balancing does not require maintainance of additional information because job schedulers already have topology information readily available. When the user runs an application, the storage balancer is again invoked to partition the allocated devices among the application processes. Once the partitioning (which is done during runtime initialization) is complete, the load balancer does not need to be involved during the lifetime of the application.

Handling Cascading Failures. For `NVMe-CR`, our stor-

age balancer tries to keep application processes and their checkpoint data on separate failure domains to minimize the possibility of both failing. Although rare, cascading failures can happen on large-scale clusters and must be handled to protect checkpoint data. This is challenging because such failures may disrupt the application and also availability of its checkpoint data. To protect data despite cascading failures, we use multi-level checkpointing [33]. In this solution, most checkpoints are still handled by `NVMe-CR`, but every so often, one checkpoint is put on a slower but more reliable parallel filesystem, such as Lustre. Through redundancy mechanisms, such as replication, such systems can guarantee that data is available even with cascading failures. Therefore, performance is not compromised because most checkpoints are handled by the fast `NVMe-CR` runtime, and fault-tolerance to cascading failures is also not compromised by relying on the redundancy of parallel filesystems. Our approach is complimentary to existing multi-level checkpointing approaches for fault-tolerance.

IV. Experimental Analysis

In this section, we present the evaluation of `NVMe-CR`.

A. Experimental Testbed

We consider a disaggregated cluster for our evaluation. Our cluster has one storage rack with 8 nodes and one compute rack with 16 nodes. Each node in the storage rack has a 28 core skylake (Gold 6132@2.6GHz) CPU with 192GB memory running Linux 3.10 and an Intel P4800X Optane SSD. Each node in the compute rack has a 28 core broadwell (E5-2680v4@2.4GHz) CPU with 128GB memory running Linux 3.10. All nodes are equipped with Mellanox ConnectX-5 adapters and are connected using 100Gbps EDR InfiniBand. Lustre is used as the PFS and is configured with 4 separate storage servers, each using one 12Gbps RAID controller.

We use ECP CoMD [14] as a representative HPC application to show the practical benefits of our proposed `NVMe-CR` runtime. Most applications in the ECP application suite, including AMG, Ember, ExaMiniMD, and miniAMR have similar behavior and are likely to show similar improvements as CoMD. We compare `NVMe-CR` performance with OrangeFS, GlusterFS, Crail [8], XFS, and ext4.

We only compare with Crail in a single server configuration because its publicly available version only supports a single NVMf server. Although we could not compare with Crail in a multi-server setting, we expect it to perform worse than NVMe-CR because Crail uses a single metadata server which becomes a bottleneck at high-concurrency. We were also unable to compare with DeltaFS; despite significant effort, we were unable to run it on our cluster.

B. Optimal Hugeblock Size

Chosing an optimal *hugeblock* size is important for obtaining the best performance. If the size is too small, metadata overhead and IO request count will be high. On the other hand, a large block size will increase the waiting time for each hardware IO queue, reducing multi-process performance. To determine the optimal *hugeblock* size, we measure the overhead of writing 512MB checkpoint files for a full subscription run. Results (see Figure 7(a)) show that 32KB is the optimal size for achieving the lowest latency. Using a 32KB block size instead of the standard 4KB delivers 7% improvement in latency. It also results in an 8x reduction in the size of the block pool and the number of inodes used. Henceforth, we use a 32KB *hugeblock* size for all experiments.

C. Load Imbalance Evaluation

Load balancing is important to ensure that available hardware resources are efficiently utilized. To measure load imbalance for different storage systems, we compare the coefficient of variation (ratio of standard deviation and mean) of load (size of data stored) on each storage server. Figure 7(b) shows the value of this coefficient when CoMD is run at different process counts. GlusterFS uses a consistent hashing algorithm to distribute data, which has been shown to have high standard deviation at low concurrency [17], just as observed. OrangeFS stripes file data across servers which works much better than consistent hashing at low concurrency but has noticable overhead at higher process counts. In contrast, `NVMe-CR` achieves perfect load balancing regardless of the level of concurrency. The reason for this is that our storage balancer creates a mapping between processes and storage devices using a round-robbin policy. Since each process creates a file of the same size, the load on each server is then exactly equal. GlusterFS and OrangeFS cannot achieve perfect load balancing because they are not associated with a single application. A mapping of processes to storage devices cannot be created since the filesystem does not know which application each client belongs to.

D. Direct Access Evaluation

To quantify the benefits of direct access, we measure the dump time for different checkpoint sizes using `NVMe-CR`, `XFS`, `ext4`, and SPDK on a local NVMe SSD for a full subscription (28 cores) run. Figure 7(c) shows the results of this analysis. Checkpoint data is written using the `write` system call and persistence is guaranteed by calling `fsync`. We find that direct access is crucial in lowering the latency for large sized checkpoints. For 512MB data, we see 19% and 83% improvement compared to `XFS` and `ext4`, respectively. This improvement comes from the ability to bypass the kernel as well as the use of *hugeblocks* and metadata provenance, which significantly reduce metadata overhead. For example, `hugeblocks` lowers the number of SSD blocks to be allocated and tracked by 4x. Compared to SPDK, `NVMe-CR` has no noticeable overhead. This clearly highlights the elimination of software and metadata overheads. Note that SPDK alone cannot handle all the IO challenges (POSIX compliance, metadata management, and private namespace) of an NVMf-enabled distributed storage system as we have discussed in this paper. We also measure the percentage of benchmark time spent in the kernel. By enabling userspace device access in `NVMe-CR`, the benchmark only spends 10% of its time in the kernel compared to 76.5% for `XFS` and 79% for `ext4`.

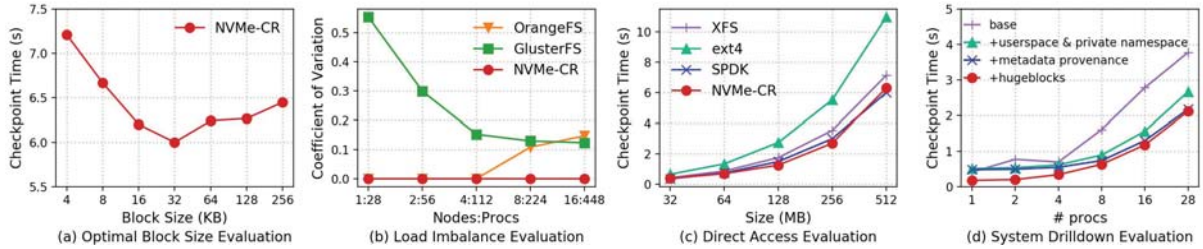

Figure 7: (a) Checkpoint times for different *hugeblock* sizes, (b) Load imbalance for `NVMe-CR`, OrangeFS, and GlusterFS, (c) Full subscription performance of `NVMe-CR`, `ext4`, `XFS`, and SPDK, and (d) Drilldown evaluation showing impact of optimizations.

This reduction is because all POSIX IO syscalls are now resolved completely in userspace. `NVMe-CR` provides userspace implementations of all IO syscalls. The 10% time spent in the kernel is because of non-IO syscalls made by the benchmark itself and during the initialization and finalization routines in `NVMe-CR` (for example as a result of calling `malloc`). We also note that on increasing data size, the performance gap increases. This is because metadata overhead has a linear correlation with file size.

E. Drilldown Evaluation

To understand the impact of the different optimizations and designs in `NVMe-CR`, we measure the checkpoint time with the CoMD application on a single node. We start with a base design resembling a traditional kernel filesystem and add optimizations one-by-one, measuring the checkpoint time for each case. Figure 7(d) shows this analysis. Bypassing the kernel and eliminating the global namespace provides up to 44% improvement compared to baseline. The improvement is also higher at scale which is because the overheads of global namespace synchronization are high. Metadata provenance improves performance up to 17% (v/s + userspace & private namespace) by lowering the size of log records. At low concurrency it improves performance by reducing software overhead while at high concurrency, it improves performance by increasing the available data bandwidth. Finally, using hugeblocks improves performance up to 62% (v/s + metadata provenance). The improvement is mostly noticable at low concurrency because performance is dictated more by software overhead than IO bandwidth. Overall, we conclude that the private namespace improves performance at high concurrency, hugeblocks improves performance at low concurrency, and metadata provenance improves performance in all cases. Combining all three optimizations delivers the best performance at all levels of concurrency.

F. NVMf Overhead

We measure the overhead of NVMf by comparing checkpoint performance on a local and remote SSD for a full subscription (28 cores) run. We also compare with Crail, a userspace storage runtime which supports NVMf via SPDK. The remote access is as demonstrated in Figure 4 over EDR InfiniBand fabric. Results (see Figure 8(a)) show that there is no noticable overhead in latency for writing checkpoints. In fact, the maximum overhead we observe is below 3.5% and is independent of the size of the checkpoint. The benefits of

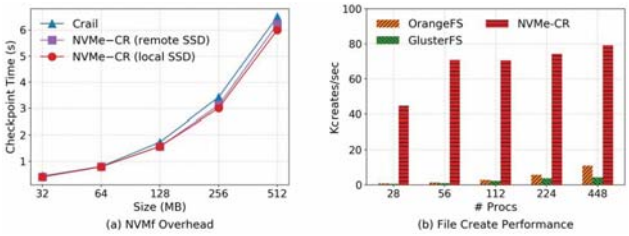

Figure 8: (a) Full subscription performance of `NVMe-CR` on a local and remote SSD, and (b) File create performance of `NVMe-CR`, OrangeFS, and GlusterFS.

OrangeFS*	GlusterFS*	NVMe-CR#
2686.25	3.5	445.25

Table I: Metadata overhead with CoMD in MB. * per storage node # per runtime

metadata provenance are clear when we compare `NVMe-CR` and Crail. Even though both use SPDK for NVMf support, `NVMe-CR` consistently provides up to 5-10% lower overhead for remote access. There are two factors which contribute to achieving negligible overhead. First, using the SPDK NVMf driver instead of the kernel NVMf driver eliminates kernel space overhead. Second, the use of metadata provenance reduces the amount of additional data that must be sent via NVMf, minimizing the overhead of metadata operations. These results highlight the advantages of enabling userspace remote access over fast RDMA networks. It is clear that the proposed userspace data and control planes are successful in creating a low latency pipeline.

G. Metadata Overhead

In the N-N checkpoint pattern, the number of files scales linearly with the job size. The number of file creates that a storage system can perform is an important metric for checkpoint IO. We compare the create performance of storage systems at different job scales to determine how well they can perform under heavy load. Results are presented in Figure 8(b). `NVMe-CR` provides 7x and 18x higher create performance at 448 processes. The primary reason for this is the absence of a global namespace. Each process can create files in parallel and avoid serialization of operations. For each file create, a corresponding entry must be added to the directory file stored on the remote SSD. Hence, the file create performance is only limited by hardware bandwidth and not software latency. OrangeFS and GlusterFS use consistent hashing to lower metadata overhead. Despite this optimization,

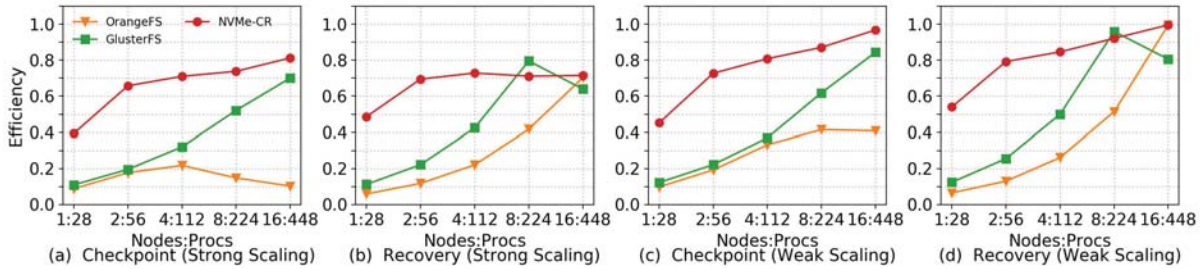

Figure 9: Efficiency of storage systems during checkpoint and recovery of the CoMD application state. Efficiency of a storage system is defined as the ratio of application perceived IO bandwidth and the hardware IO bandwidth.

both must add file entries to a single common directory file which effectively serializes file creates, leading to poor performance.

We also measure the storage overhead of metadata and checkpoints compared to OrangeFS and GlusterFS. As we can see from Table I, `NVMe-CR` requires ∼450MB storage per runtime, OrangeFS requires ∼2.6GB, and GlusterFS only requires 3.5MB per storage node. The total overhead for `NVMe-CR` can be calculated by multiplying this value with the number of application processes. For high concurrency runs, the total overhead for `NVMe-CR` is higher than both OrangeFS and GlusterFS, but remains under acceptable limits. For instance, on our cluster only 1.7% of SSD space is utilized for metadata and checkpoint storage, in the worst case. We also measure the DRAM footprint and find that NVMe-CR consumes less than 512MB per-instance – 404MB for inodes and 102MB is for B+Tree. This minor increase in metadata overhead can be justified by the significant performance benefits of our design. For GlusterFS, the overhead is minimal because it uses consistent hashing which requires little metadata to be maintained. On the other hand, OrangeFS has high overhead as it needs to store both file metadata and striping information.

H. Application Evaluation

We measure the overhead of checkpointing the CoMD application state under both weak and strong scaling configurations. We compare checkpoint as well as recovery efficiency with `NVMe-CR`, OrangeFS, and GlusterFS. We do not compare with Crail because it currently only supports a single storage server for its NVMf tier. We define the *efficiency* of a storage runtime as the ratio of the peak IO bandwidth visible to applications to the peak theoretical bandwidth offered by hardware. Checkpoint performance is only dependent on overall bandwidth of the storage system. Therefore we use efficiency as a metric which allows us to compare the relative checkpoint performance of different storage systems. For all cases we use the aggregate SSD bandwidth as the hardware peak.

Strong Scaling. For strong scaling analysis, the problem size is fixed to 16,384K atoms for a total fixed checkpoint size of 86GB (for 10 checkpoints). Figures 9(a) and 9(b) show the checkpoint and recovery efficiency for multi-node full subscription runs. Overall, we find that `NVMe-CR` achieves better efficiency for both cases than other storage systems. Synchronization-free control and data planes are responsible for good scalability. The load balancer in `NVMe-CR` allows it to attain better efficiency than other systems at lower process counts. This is because our greedy load balancing algorithm ensures that all of the SSDs are utilized efficiently. Apart from GlusterFS, other systems are unable to handle the metadata burden when 448 processes concurrently checkpoint data. The reason is the use of a global namespace and high software overhead. During recovery, however, they perform much better since there is no metadata overhead involved. GlusterFS is better than other systems we compare with because of its decentralized design. Nevertheless, its checkpoint performance is still ∼ 13% lower than `NVMe-CR` because of its poor create performance (see Figure 8(b)) and reliance on POSIX IO (see Figure 7(c)). At high concurrency, `NVMe-CR` achieves the best efficiency of all systems because of its coordination-free design.

Metric	OrangeFS	GlusterFS	NVMe-CR
Checkpoint Time (s)	85.9	44.5	39.5
Recovery Time (s)	3.6	4.5	3.6
Progress Rate	0.252	0.402	0.423

Table II: CoMD evaluation with multi-level checkpointing at 448 processes. For progress rate, higher values are better.

Weak Scaling. We fix the problem size to 32K atoms per process while scaling up to 448 processes. We take 10 periodic checkpoints during application runs for a total checkpoint dump of 700GB. This experiment was designed to show that `NVMe-CR` can handle large data volume. Figures 9(c) and 9(d) show the results of this analysis. Overall, we find that `NVMe-CR` achieves near perfect efficiency (0.96 for checkpoint and 0.99 for recovery) at 448 processes. Recovery efficiency is so high because of log record coalescing which allows the runtime to recover instantaneously and fully utilize available bandwidth. For checkpoints, other systems suffer from the high synchronization overhead of creating a large number of concurrent files, leading to poor efficiency. During recovery, just like in the strong scaling case, their performance improves significantly because there is less synchronization involved to read files. However, GlusterFS performance dips at 448 processes because its metadata server is unable to handle the large influx of read requests. In contrast, `NVMe-CR` achieves good efficiency at both low and high concurrency by avoiding synchronization and achieving direct device access.

I. Multi-Level Checkpointing Evaluation

To evaluate `NVMe-CR` in a real-world use case with cascading failures, we conduct an experiment with CoMD at 448

processes. We use different systems for the first checkpoint level and Lustre as the second level, where one checkpoint in ten is written to Lustre. We compare checkpoint time, recovery time, and application progress rate for different systems in Table II. It is clear that NVMe-CR is the best for all metrics, which is because its efficiency is near ideal. Compared to GlusterFS, it reduces checkpoint time by 11%, recovery time by 20% and progress rate by 5%. The large improvement in recovery is due to log record coalescing (without coalescing, recovery takes 4s) which allows very fast metadata recovery. Therefore, using NVMe-CR directly results in benefits at the application level, reducing overall runtime and increasing the probability of applications running successfully.

V. Conclusion and Future Work

In this paper, we presented the design of NVMe-CR, a storage runtime for disaggregated clusters with NVMf. First, we presented a powerful design template for filesystems meant for storing ephemeral application data. By following the design principles of this template, filesystems can achieve low latency direct access to device. Built upon these principles, NVMe-CR provides synchronization-free control and data planes as well as a fault-tolerance and load-aware storage balancing. By proposing techniques like metadata provenance, log record coalescing, and hugeblocks, NVMe-CR is well suited for storing checkpoint data. Experimental analysis with CoMD shows that on a local cluster our runtime can achieve near perfect (> 0.96) efficiency at 448 processes. As a result, our runtime lowers checkpoint overhead by up to 2x, while increasing job progress rates by as much as 1.6x. In the future, we plan to study the impact of a cache layer over NVMe-CR and evaluate with more applications.

References

[1] ORNL, "U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL," https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl, 2019.

[2] A. Agrawal, G. H. Loh, and J. Tuck, "Leveraging Near Data Processing for High-Performance Checkpoint/Restart," in *SC'17*, p. 60.

[3] J. Cao, K. Arya, R. Garg, S. Matott, D. K. Panda, H. Subramoni, J. Vienne, and G. Cooperman, "System-level Scalable Checkpoint-Restart for Petascale Computing," in *ICPADS'16*, pp. 932–941.

[4] B. Kim, J. Kim, and S. H. Noh, "Managing Array of SSDs When the Storage Device is No Longer the Performance Bottleneck," in *HotStorage'17*.

[5] NVMe Express, "NVMe over Fabrics," http://www.nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf, 2016.

[6] Z. Guz, H. H. Li, A. Shayesteh, and V. Balakrishnan, "NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation," in *SYSTOR'17*, p. 16.

[7] P. Schwan, "Lustre: Building a File System for 1000-node Clusters," in *Proceedings of the Linux Symposium*, 2003, pp. 380–386.

[8] Apache, "Crail," https://crail.incubator.apache.org/, 2019.

[9] The OrangeFS Project, "OrangeFS," http://www.orangefs.org/, 2019.

[10] IBM, "BSCFS," https://github.com/IBM/CAST, 2018.

[11] R. Hat, "GlusterFS," https://www.gluster.org/, 2019.

[12] LLNL, "UnifyCR," https://github.com/LLNL/UnifyFS, 2018.

[13] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An Ephemeral Burst-Buffer File System for Scientific Applications," in *SC'16*, p. 69.

[14] ExMatEx, "CoMD: Classical Molecular Dynamics Proxy Application," https://github.com/ECP-copa/CoMD, 2016.

[15] V. Meshram, X. Besseron, X. Ouyang, R. Rajachandrasekar, R. P. Darbha, and D. K. Panda, "Can a Decentralized Metadata Service Layer Benefit Parallel Filesystems?" in *Cluster'11*, pp. 484–493.

[16] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding Manycore Scalability of File Systems," in *USENIX ATC'16*, pp. 71–85.

[17] J. Lamping and E. Veach, "A Fast, Minimal Memory, Consistent Hash Algorithm," *arXiv preprint arXiv:1406.2294*, 2014.

[18] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers," in *Proceedings of the 10th Parallel Data Storage Workshop*. ACM, 2015, pp. 1–6.

[19] Q. Zheng, K. Ren, and G. Gibson, "BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers," in *9th Parallel Data Storage Workshop*, 2014, pp. 1–6.

[20] Q. Xu, M. Awasthi, K. Malladi, J. Bhimani, J. Yang, and M. Annavaram, "Performance Analysis of Containerized Applications on Local and Remote Storage," in *MSST'17*.

[21] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda, "A 1 PB/s File System to Checkpoint Three Million MPI Tasks," in *HPDC'13*, pp. 143–154.

[22] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield, "Zest Checkpoint Storage System for Large Supercomputers," in *Petascale Data Storage Workshop*. IEEE, 2008, pp. 1–5.

[23] X. Ouyang, R. Rajachandrasekar, X. Besseron, H. Wang, J. Huang, and D. K. Panda, "CRFS: A Lightweight User-Level Filesystem for Generic Checkpoint/Restart," in *ICPP'11*, pp. 375–384.

[24] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *SC'09*, p. 21.

[25] J. Kim, S. Lee, and J. S. Vetter, "PapyrusKV: A High-Performance Parallel Key-Value Store for Distributed NVM Architectures," in *SC'17*, pp. 1–14.

[26] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System," in *HPDC'18*, pp. 219–230.

[27] DDN, "Infinite Memory Engine," https://www.ddn.com/products/ime-flash-native-data-cache/, 2019.

[28] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and Design of Cray DataWarp," *Cray User Group*, 2016.

[29] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler, "Unification of Temporary Storage in the NodeKernel Architecture," in *USENIX ATC'19*, pp. 767–782.

[30] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, pp. 494–499, 2006.

[31] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "libhashckpt: Hash-Based Incremental Checkpointing Using GPU's," in *EuroMPI'11*, pp. 272–281.

[32] A. J. Oliner, L. Rudolph, and R. K. Sahoo, "Cooperative Checkpointing: A Robust Approach to Large-scale Systems Reliability," in *ICS'06*, pp. 14–23.

[33] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *SC'10*, pp. 1–11.

[34] D. Ibtesham, K. B. Ferreira, and D. Arnold, "A Checkpoint Compression Study for High-Performance Computing Systems," *The International Journal of High Performance Computing Applications*, vol. 29, no. 4, pp. 387–402, 2015.

[35] Intel, "SPDK," https://github.com/spdk/spdk, 2019.

[36] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs," in *HotStorage 16*.

[37] Linux, "libaio," https://docs.oracle.com/cd/E36784_01/html/E36873/libaio-3lib.html, 2014.

[38] Intel, "Enhanced Power-Loss Data Protection in the Intel Solid-State Drive 320 Series," November 2016, available at: https://newsroom.intel.com/wp-content/uploads/sites/11/2016/01/Intel_SSD_320_Series_Enhance_Power_Loss_Technology_Brief.pdf (Sep. 2018).

[39] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell *et al.*, "The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems," in *SC'18*, p. 52.

[40] Slurm, "Slurm Generic Resources," https://slurm.schedmd.com/gres.html, 2019.