Fast In-Memory Checkpointing with POSIX API for Legacy Exascale-Applications

Jan Fajerski, Matthias Noack, Alexander Reinefeld, Florian Schintke, Torsten Schütt, and Thomas Steinke

Abstract Exascale systems will be much more vulnerable to failures than today's high-performance computers. We present a scheme that writes erasure-encoded checkpoints to other nodes' memory. The rationale is twofold: first, writing to memory over the interconnect is several orders of magnitude faster than traditional disk-based checkpointing and second, erasure encoded data is able to survive component failures. We use a distributed file system with a tmpfs back end and intercept file accesses with LD_PRELOAD. Using a POSIX file system API, legacy applications which are prepared for application-level checkpoint/restart, can quickly materialize their checkpoints via the supercomputer's interconnect without the need to change the source code.

Experimental results show that the LD_PRELOAD client yields 69 % better sequential bandwidth (with striping) than FUSE while still being transparent to the application. With erasure encoding the performance is 17 % to 49 % worse than striping because of the additional data handling and encoding effort. Even so, our results indicate that erasure-encoded memory checkpoint/restart is an effective means to improve resilience for exascale computing.

1 Introduction

The path towards exascale computing with 10^{18} operations per second is paved with many obstacles. Three challenges are being tackled in the DFG project '*A Fast and Fault-Tolerant Microkernel-Based System for Exascale Computing*' (FFMK) [21]: (1) the vulnerability to system failures due to transient and permanent errors, (2) the performance losses due to workload imbalances in applications running on hundreds of thousands of cores, and (3) the performance degradation caused by interactions and noise of the operating system.

J. Fajerski (✉) • M. Noack • A. Reinefeld • F. Schintke • T. Schütt • T. Steinke
Zuse Institute Berlin (ZIB), Berlin, Germany
e-mail: fajerski@zib.de; noack@zib.de; reinefeld@zib.de; schintke@zib.de; schuett@zib.de; steinke@zib.de

H.-J. Bungartz et al. (eds.), *Software for Exascale Computing – SPPEXA 2013-2015*, Lecture Notes in Computational Science and Engineering 113,
DOI 10.1007/978-3-319-40528-5_19

This paper addresses the first challenge, that is, to improve the fault tolerance of exascale systems. Such systems consist of hundreds of thousands of components, each designed to be reliable by itself. But running them all together will render node failures a common event applications have to cope with [4, 7]. Several mechanisms for improving fault tolerance in HPC have been suggested, like fault-tolerant communication layers and checkpoint/restart (C/R) techniques.

C/R typically uses fast parallel file systems like Lustre,[1] GPFS [17], or Panasas [10] to materialize the checkpoints on disks. Unfortunately, C/R will reach its limits as the applications' memory footprint grows faster than the parallel I/O bandwidth. Writing a checkpoint to disk will make the system processors idle for a growing fraction of time, which becomes increasingly uneconomic. On a typical HPC system like the Cray XC40 at ZIB [2] it takes more than half an hour to write the main memory's capacity to the parallel Lustre file system.[2] Thus, reducing the time of checkpointing is of vital importance. It does not only improve the efficiency, but it will become a necessity when the mean time between failure (MTBF) becomes shorter than the time needed to persist a checkpoint to disk.

2 Related Work

In-memory checkpointing [12] has been known for a long time. Several schemes like Charm++ [23, 24] and FTI [3, 8] have been successfully deployed. SCR[3] implements multi-level checkpointing, where checkpoints are written to different media like RAM, flash or rotating disks. It uses a simple RAID5 encoding to be able to cope with additional component failures.

Unfortunately, the mentioned approaches are difficult to apply to legacy applications. They are either limited to the use of specific object-oriented programming languages like Charm++ or they require source code modifications to use specific APIs like the one used by SCR for reading and writing checkpoints. The BLCR [5] checkpoint framework is able to checkpoint unmodified applications but requires support from the MPI library, because it can only create a consistent checkpoint of MPI applications when no messages are in flight.

Our approach, in contrast, is based on POSIX which makes it suitable for legacy applications, since many applications are prepared to write and read their checkpoints using POSIX file system operations.

[1] http://wiki.lustre.org/

[2] The Cray XC40 'Konrad' is operated at ZIB as part of the North German Supercomputer Alliance. It comprises 1872 nodes (44.928 cores), Cray Aries network, 120 TB main memory, and a parallel Lustre file system of 4.5 PB capacity and 52 GB/s bandwidth.

[3] https://computation.llnl.gov/project/scr/

3 In-Memory Checkpointing with POSIX API

3.1 Implementation with XtreemFS

We use XtreemFS [19], a scalable distributed file system developed at ZIB, as a basis for our in-memory checkpointing mechanism. XtreemFS supports POSIX semantics for file operations while—transparently for the application—providing fault tolerance via file replication on distributed servers. We modified XtreemFS to perform I/O operations in-memory rather than on disk. This was possible because XtreemFS is a user-space file system and is therefore not as tightly integrated into the operating system kernel as other parallel file systems. Hence, it is well-suited for providing in-memory checkpointing on top of the L4-microkernel used in the FFMK project [21].

An instance of XtreemFS comprises three services: the *Directory Service (DIR)* is a central registry for all XtreemFS services and is used for service discovery. The *Metadata and Replica Catalog (MRC)* stores the directory tree and file metadata, and it manages user authentication and file access authorization. The *Object Storage Device (OSD)* stores the actual file data as objects. Figure 1 illustrates the architecture of XtreemFS. Clients and servers (MRC, OSD, DIR) are connected via some network with no specific requirements in terms of security, fault tolerance and

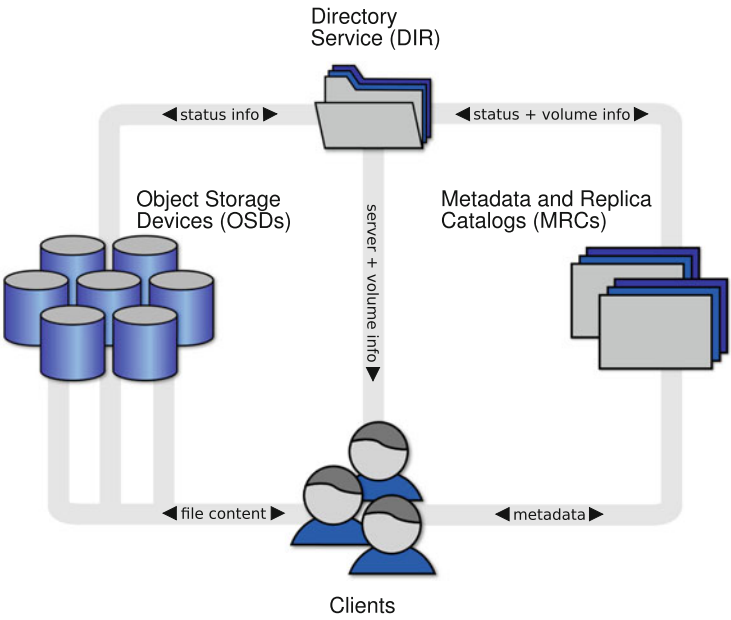

Fig. 1 XtreemFS architecture illustrating the three services (OSD, MRC and DIR) and the communication patterns between them and a client

performance. The separation of the metadata management in the MRCs from the I/O-intensive management of file content in the OSDs is a design principle found in many object-based file systems [10, 20]. To maximize scalability, metadata and storage servers are loosely coupled. They have independent life cycles and do not directly communicate with each other.

OSDs store data in their local directory tree. The underlying file system is only required to offer a POSIX compliant interface. We configured the OSDs to use the `tmp` directories of their respective nodes for data storage. The `tmp` directory is a *tmpfs* file system that exports Linux' disk caching subsystem as a RAM-based file system with an overflow option into swap space when the main memory capacity is exhausted. Thereby, all data sent to an OSD is stored solely in the node's main memory.

To create an XtreemFS instance, at least one OSD, MRC and DIR are needed. Though to make full use of XtreemFS' fault tolerance and scalability features, several OSDs should be started on different servers. Once all desired services are started and a volume is created, it can be mounted on any number of clients. All clients will see the same file system with the same directory tree. An XtreemFS volume is mounted through the *FUSE* kernel module[4] to provide a virtual file system. All operations in this file system are passed to the XtreemFS client library, which distributes the data to the OSD devices.

3.2 Fault-Tolerance and Efficiency with Erasure Codes

Data replication is a frequently used option in distributed file systems to provide fault tolerance. However, replication implies storage and communication overhead compared to the number of tolerated failures. The commonly used 3-way replication [18, 19] causes a $^2/_3$ overhead of the available raw storage capacity but can only tolerate one failed replica.

Erasure codes (EC) offer a more space-efficient solution for fault tolerance and have recently gained a lot of attention [1, 6, 9, 11, 14–16]. EC are a family of error correction codes that stripe data across k chunks. Every k data words are encoded to $k + m = n$ words such that the original data words can be recovered from any subset $\{s \mid s \in \mathscr{P}(n), |s| \geq k\}$ (cf. Fig. 2). Compared to replication, EC offers the same or higher fault tolerance at a fraction of the storage overhead.

An EC is considered *systematic* if it keeps the original k data words in its original representation. This property is desirable for storage applications since a fault-free read operation can simply consider the data striped across k objects. They can be read in parallel and no decoding is necessary.

[4] FUSE—Filesystem in Userspace allows the creation of a file system without changing Linux kernel code.

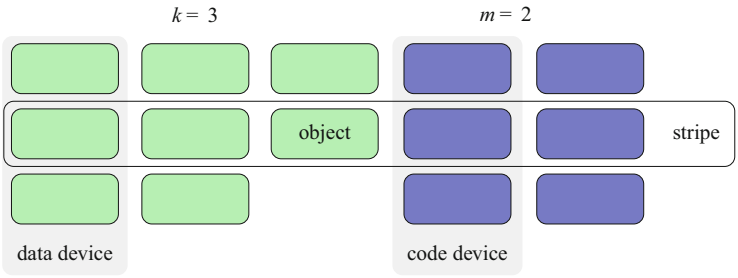

Fig. 2 Scheme of a systematic erasure code where data is striped across three devices and two code words per stripe are stored on two coding devices. Note that the last stripe only contains two data words

In order to exploit these advantages we implemented generic $k+m$ erasure coding in XtreemFS using the *Jerasure* library [13]. The encoding and decoding operation is implemented in the client library of XtreemFS.

File system operations are passed to the client library which then translates the request into a number of object requests, that are sent to the OSDs. A read request is performed optimistically by the client as it tries to read only the necessary data objects. If all OSDs with the corresponding data objects answer, the replies are concatenated and returned. If one or more OSDs fail to reply, the client will send out the necessary requests to OSDs storing coding objects. As soon as at least k out of n objects per stripe have been received, the data can be decoded and returned.

The implementation of a write operation is more complex. Since the encoding takes place in the client (i.e. on the client system) it needs a sufficient amount of data to calculate the coding objects. This establishes two requirements for write operations: (1) the size of a write operation should be $k * object\_size$ or a multiple thereof and (2) a write operation should align with stripes, i.e. the operation's offset should be a multiple of $k * object\_size$. These two requirements would diminish the POSIX compliance as POSIX does not put any restrictions on a write operation's size or offset.

This problem can be solved by implementing a *read-modify-write (rmw)* cycle in the client. When the client library receives a write request that violates the requirements above, it simply reads the necessary data to fulfill both requirements. The write request is then padded with the additional data, encoded and sent to the OSDs for storage. A write to the end of a file can simply be padded with zeros, since they act as neutral elements in the encoding operation.

However, we decided not to implement a rmw cycle, because we observed that writing checkpoint data is usually an append operation to the end of a file rather than updating random file locations. This means that both requirements of our client side implementation can be satisfied by simply caching write operations until a full stripe can be encoded and written to the OSDs or the file is closed. In the latter case the left over data is padded with zeros, encoded and then written to the OSDs.

At a later stage we will add a server side implementation of erasure coding to XtreemFS that does not suffer from the described shortcomings and will be fully POSIX compliant.

4 Deployment on a Supercomputer

4.1 Access to RAM File System

For Linux and Unix systems, there are two client solutions: the FUSE-based client that allows to mount an XtreemFS volume like any other file system, and the *libxtreemfs* library for C++ and Java, which allows application developers to directly integrate XtreemFS support into applications.

Since many HPC systems use a Linux-based operating system, the FUSE-client of XtreemFS would be a natural choice to use for our C/R system. But for performance reasons Linux configurations on HPC systems are often optimized and kernel modules like FUSE are typically disabled. We therefore developed a third client that intercepts and substitutes calls to the file system with the LD_PRELOAD mechanism. LD_PRELOAD allows to load libraries that are used to resolve dynamically linked symbols before other libraries (e.g. *libc*) are considered.

We implemented a *libxtreemfs_preload* that can be specified via the environment variable LD_PRELOAD. It intercepts and substitutes file system calls of an application. If an intercepted call relates to an XtreemFS volume or file, it is translated into its corresponding *libxtreemfs* call, which is similar to what the FUSE adapter does. Otherwise calls are passed through to the original *glibc* function, which would have handled it without the pre-load mechanism in place. Whether or not XtreemFS should be used is determined via a configurable path prefix, that can be thought of as a virtual mount point. For example, copying a file to an XtreemFS volume via FUSE using `cp` as application would be performed as follows:

```
$> mount.xtreemfs my.dir.host/myVolume /xtreemfs
$> cp myFile /xtreemfs
$> umount.xtreemfs /xtreemfs
```

The same operation with the LD_PRELOAD and *libxtreemfs_preload* instead of FUSE could be achieved with the following command:

```
$> XTREEMFS_PRELOAD_OPTIONS="my.dir.host/myVolume /xtreemfs" \
   LD_PRELOAD="libxtreemfs_preload.so" \
   cp myFile /xtreemfs
```

This example can be easily generalized into a shell script that wraps `cp` or other applications, such that the environment setup is hidden.

Figure 3 shows all three client solutions in comparison. The LD_PRELOAD client combines the transparency of the FUSE client with the potential performance benefits of directly using the *libxtreemfs*. Section 5 provides benchmark results on the different XtreemFS client solutions.

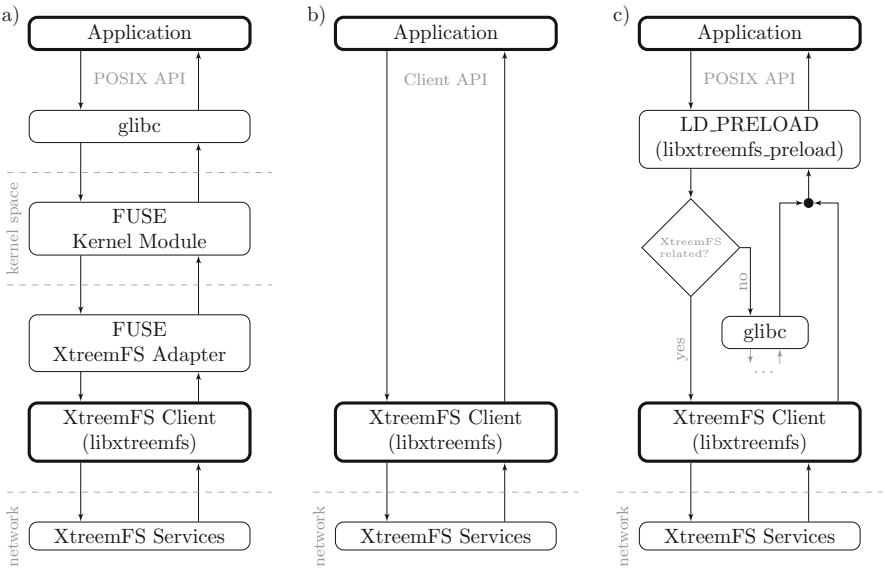

Fig. 3 Three different client solutions for XtreemFS. (**a**) The application interacts transparently with XtreemFS via FUSE. (**b**) The direct use of the client library avoids overhead but is intrusive. (**c**) The interception with LD_PRELOAD is non-intrusive and avoids FUSE as a bottleneck

Note that a similar approach is available with the *liblustre*[5] library in the Lustre parallel file system. Here, data of a Lustre volume can be accessed via LD_PRELOAD without the need of mounting. However, *liblustre* targets more at portability than performance.

4.1.1 Issues with LD_PRELOAD

The LD_PRELOAD mechanism is only able to intercept calls to dynamically linked functions. In most cases this works fine for the low-level file system calls of interest. However, there are situations where some of the calls are inaccessible. If we wanted, for instance, to intercept all `close()` calls made by an application there are two possible situations: the application either directly calls `close()` or it uses a higher-level operation like `fclose()` which then calls `close()` indirectly. The second case is problematic, since both calls are inside *glibc* and the inner `close()` call could have been inlined or statically linked, depending on the *glibc*-build. If so, it can not be intercepted by the LD_PRELOAD mechanism. One possible workaround is to also intercept the higher-level calls, but this would mean re-implementing and maintaining large parts of the *glibc*, which is not a good choice. A more practical

[5] http://wiki.lustre.org/index.php/LibLustre_How-To_Guide

workaround is to use a simple test-program to detect whether or not all needed calls can be intercepted, and if not use a specifically built *glibc*.

4.2 Placement of Services

One important decision is, where to store the erasure encoded checkpoint data so that it can be later safely retrieved—even under harsh conditions with correlated component failures. A look into the operators' machine books at major HPC sites reveals that single hardware components like memory DIMMs, processors, power supplies or network interface cards (NIC) fail independently, but they cause larger parts of the systems to crash. Similarly, software crashes also cause parts of the system to fail. In all cases the failure unit is typically a single node, which comprises some CPU sockets with associated ccNUMA memory and a NIC.

On the Cray XC40, the smallest failure unit is a compute blade with four nodes because they share one Aries NIC and other support hardware. Consequently, erasure encoded checkpoint data should be distributed over several blades in the Cray. As shown in another work [22], the latency increases only by a negligible amount when writing data from one blade to another in the same electrical unit via the Aries network (Fig. 4). This is also true when using the longer-distance fiber optics cables between different electrical units (pairs of cabinet). Moreover, the bandwidths are almost homogeneous across the entire system.

When an application crashed due to a node failure, additional resources are need to resume the application from a checkpoint. If all resources in the system are fully utilized, it may be difficult to provide additional nodes for the crashed application. On exascale systems, however, we expect the cost of reserving a few spare nodes per job to be negligible. Alternatively, the system provider could provide spare nodes from job fragmentation. The job will then be restarted with the same number of nodes but a slightly different job placement. As shown in [22] the cost for different placements varies only by a few percent.

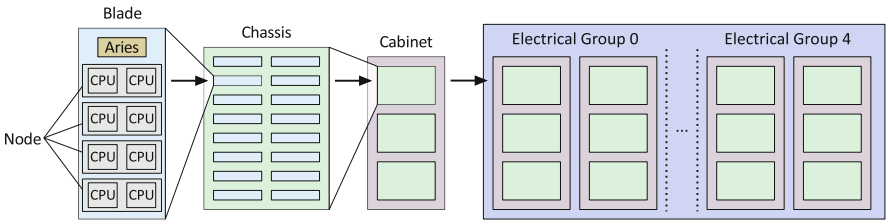

Fig. 4 Cray XC40 component hierarchy: node, blade, chassis, cabinet, electrical group

4.3 Deployment on a Cray XC40

Jobs on a Cray XC40 may be either run in extreme scalability mode (ESM) or in cluster compatibility mode (CCM). ESM is designed for scalable high performance applications. Only a minimal amount of system services are running on the compute nodes to minimize interference. Applications need to use Cray MPI and to be started with the command-line tool `aprun`. After the application finishes, the node-checker checks the node for errors and cleans up all remaining traces of the previous job. The ESM mode is not suited for our approach, because we need to run both, XtreemFS and the application on the same node. Additionally, we need to restart the application on these nodes and read the checkpoint from XtreemFS resp. the RAM-disk which is impossible with ESM.

In CCM, the reserved compute nodes can be treated like a traditional cluster. Standard system services, like an ssh daemon, are available. However, Cray MPI is not available in CCM. Instead, Cray provides an InfiniBand (IB) verbs emulation on top of the Aries network. For our experiments, we ran OpenMPI over the IB verbs emulation.

For a proof of concept we used the parallel quantum chromodynamics code BQCD.[6] It has a built-in checkpoint/restart mechanism. At regular intervals, it writes a checkpoint of its state to the local disk. In case of a crash, it can be restarted from these checkpoints.

We used ssh to start the XtreemFS services on the nodes in CCM mode. The services were distributed as follows: DIR, MRC and one OSD on the first node, and one OSD on all other nodes. We used the LD_PRELOAD client and OpenMPI to start BQCD. In this setup BQCD's snapshots are written to XtreemFS. We manually killed the BQCD job and successfully restarted it from the memory checkpoint.

5 Experimental Results

First, we evaluate the three XtreemFS client solutions described in Sect. 4.1. In order to compare the cost of the different data paths depicted in Fig. 3, we performed micro-benchmarks of the read and write operations to an XtreemFS volume with each solution. The different XtreemFS services ran on a single node, so there is no actual network traffic that might pose a bottleneck. A node has two Intel Xeon E5-2630v3 with 64 GB main memory and runs a Ubuntu 14.04 with a 3.13 kernel. Caching mechanisms of the kernel and FUSE were disabled by using the `direct_io` option of FUSE. This ensures that all requests reach the XtreemFS client and the OSDs, and are not just cached locally, which is especially important

[6] https://www.rrz.uni-hamburg.de/services/hpc/bqcd.html

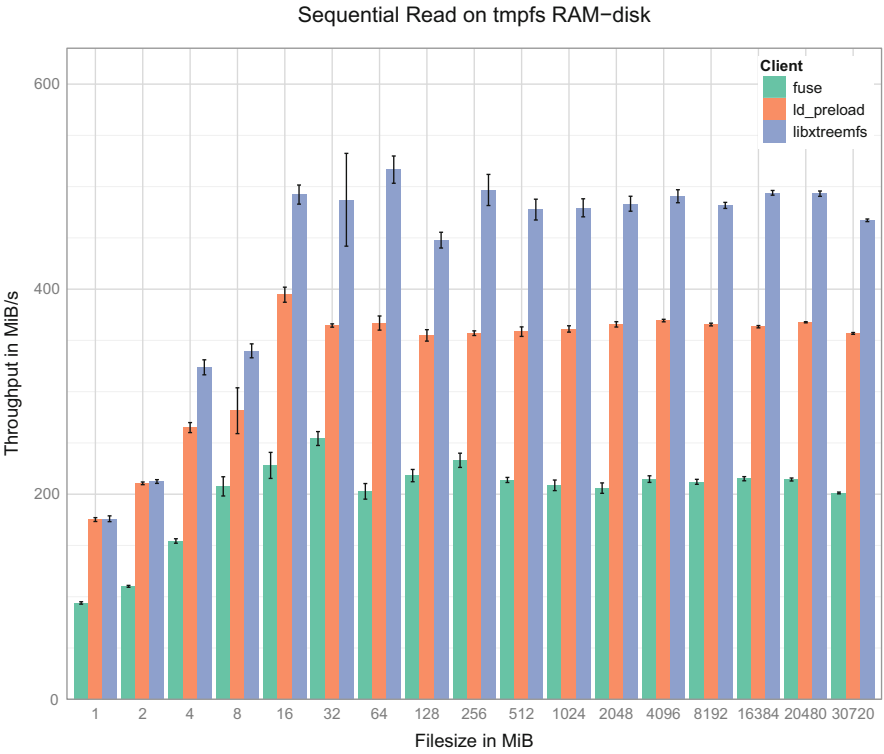

Fig. 5 Sequential read performance using the three different client approaches (RAM-disk)

for checkpoint data. All result values are averages over multiple runs, the error bars visualize the standard error.

Figures 5 and 6 show the results for sequential reading and writing, respectively. In both cases, the results match our expectations: *libxtreemfs* is faster than LD_PRELOAD which is faster than FUSE. For reading, LD_PRELOAD is between 35 % and 91 % faster than FUSE with an average of 69 %. Compared to *libxtreemfs*, it is around 21 % slower on average (between 0.5 % and 29 %). Writing performance is similar. LD_PRELOAD compared to FUSE is around 74 % faster (between 44 % and 91 %), and 23 % (between 6 % and 33 %) slower when compared to *libxtreemfs*. The results show that the newly developed LD_PRELOAD client approach yields a better sequential bandwidth than FUSE while still being transparent to the application. Regardless of performance, LD_PRELOAD is the only solution for applications that run in an environment where FUSE is not available and where modifying the application code is not possible.

The absolute throughput values of these micro-benchmarks are limited by the synchronous access pattern and the use of only a single data stream and a single client. In a real world scenario, there would be at least one client or data stream (i.e.

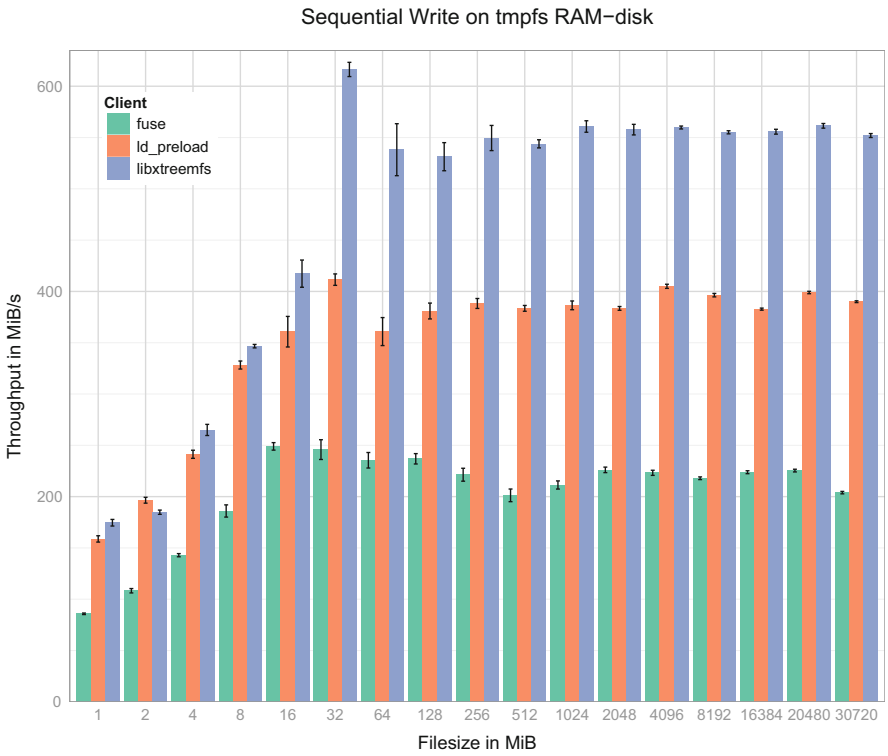

Fig. 6 Sequential write performance using the three different client approaches (RAM-disk)

file) per rank or thread, whose throughput would add up to an overall throughput that would be limited by some physical bound of the underlying hardware (i.e. memory, disk, or network bandwidth).

In a second experiment we compared the sequential throughput and scaling characteristics of the existing striping implementation and the client side erasure-coding solution. We used a distributed XtreemFS setup with 2–13 data OSDs. On each data OSD runs one IOR[7] process that reads/writes 1 GiB of data via the FUSE interface. MRC and DIR run on a separate machine. All machines have two Intel Xeon E5-2630v3 and 64 GB main memory, and all machines are interconnected with 10 Gbit/s. In the erasure-coding experiment, the XtreemFS instance has two additional OSDs for coding data and thus provides a RAID6-like configuration. All result values are averages over 10 runs with error bars that visualize the standard error.

[7]IOR is a I/O micro benchmark software by NERSC. https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/ior/

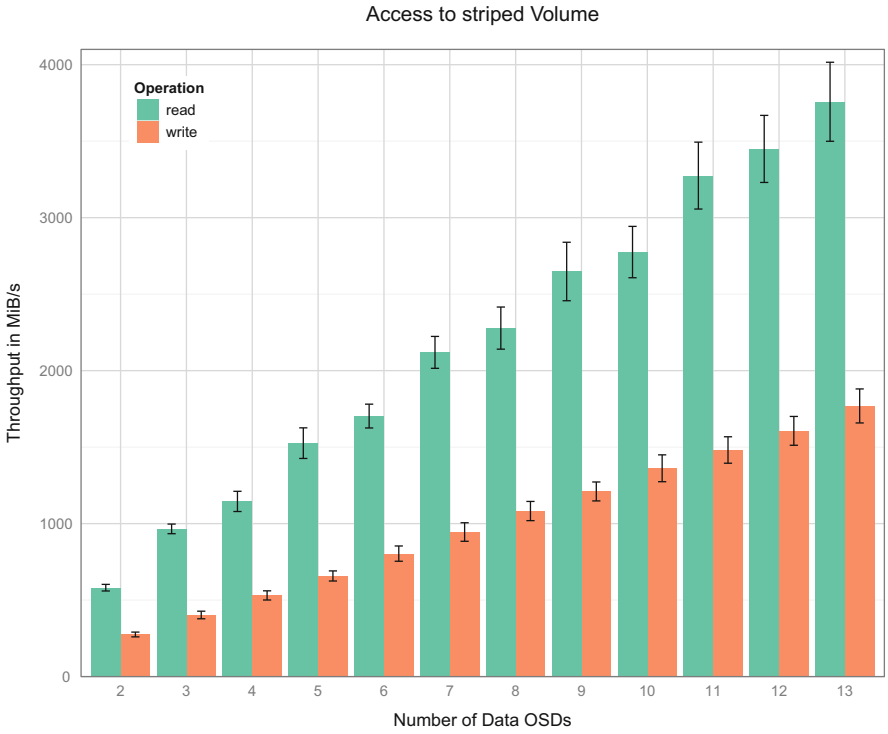

Fig. 7 Sequential read/write performance on a striped XtreemFS volume with an increasing number of OSDs and clients

Figures 7 and 8 show the results for reading and writing to variably sized XtreemFS instances. Both the striping and erasure-coding configuration exhibit good scaling characteristics. Compared to the striping configuration, writes to the erasure coded volume are 17–49 % slower, which reveals the overhead caused by the additional coding data. This corresponds roughly to the 15 % to 50 % data overhead the coding induces. For reference, a replicated setup that provides the same level of fault tolerance would induce a 200 % data overhead. The read operation exhibits a slowdown between 5 % and 14 % in the erasure-coding configuration.

The results show a performance penalty for using erasure codes in both reading and writing. For writes this slowdown was to be expected since each write operation creates a coding data overhead. When the achieved fault tolerance is taken into consideration the overhead appears insignificant.

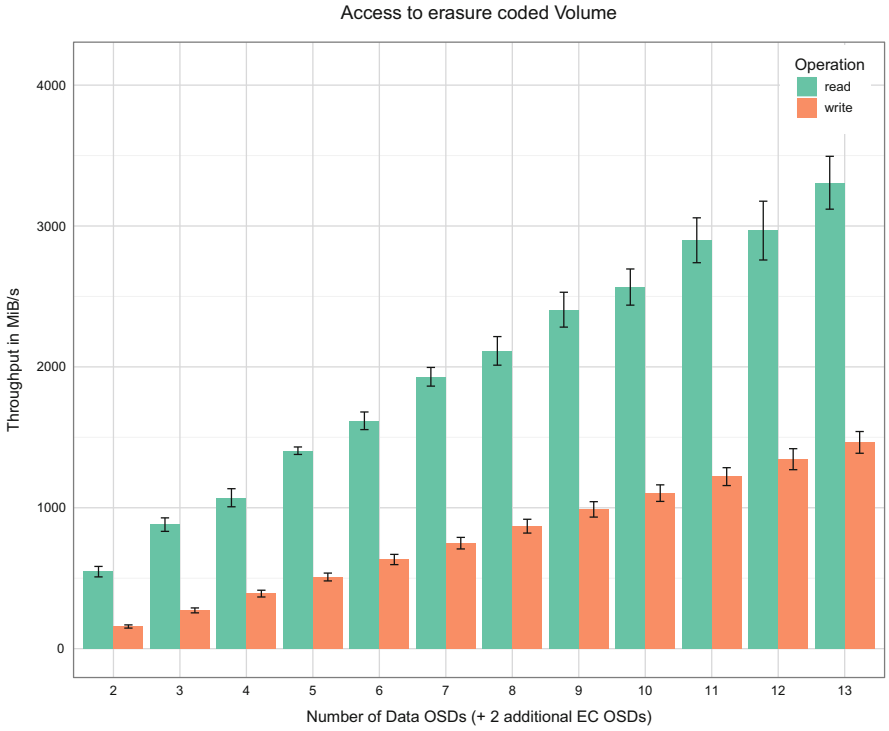

Fig. 8 Sequential read/write performance on an erasure coded XtreemFS volume with an increasing number of OSDs and clients

6 Summary

Checkpoint/Restart is a viable means to increase failure tolerance on supercomputers. We presented results on the implementation of a POSIX based checkpoint/restart mechanism. Checkpoints are stored in a RAM based distributed file system using XtreemFS. For fault tolerance checkpoints are encoded using erasure codes.

We evaluated our solution on a Cray XC40 with the quantum chromodynamics code *BQCD* which is already prepared for application-level checkpointing. XtreemFS provides three different clients: a FUSE based client, LD_PRELOAD and *libxtreemfs*. The first requires the FUSE kernel module to be loaded, which is typically not available on supercomputer environment. The last client, *libxtreemfs*, requires the application code to be modified and is therefore also not a good choice. The LD_PRELOAD client results in performance improvements for sequential access and extends the number of supported platforms and applications. The new client implementation transparently bypasses the operating system overhead by intercepting POSIX file system calls and redirecting them to *libxtreemfs*.

Acknowledgements We thank Johannes Dillmann who performed some of the experiments. This work was supported by the DFG SPPEXA project '*A Fast and Fault-Tolerant Microkernel-Based System for Exascale Computing*' (FFMK) and the North German Supercomputer Alliance HLRN.

References

1. Asteris, M., Dimakis, A.G.: Repairable fountain codes. In: 2012 IEEE International Symposium on Information Theory Proceedings (ISIT), pp. 1752–1756. IEEE (2012)
2. Baumann, W., Laubender, G., Läuter, M., Reinefeld, A., Schimmel, C., Steinke, T., Tuma, C., Wollny S.: HLRN-III at Zuse Institute Berlin. In: Vetter, J. (ed.) Contemporary High Performance Computing: From Petascale Toward Exascale, vol. 2, pp. 85–118. Chapman & Hall/CRC Press (2014)
3. Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: FTI: high performance fault tolerance interface for hybrid systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), New York, pp. 32:1–32:32. ACM (2011)
4. Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., Snir, M.: Toward exascale resilience: 2014 update. Supercomput. Front. Innov. **1**(1), 1–28 (2014)
5. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters. In: Proceedings of SciDAC 2006, Denver (2006)
6. Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., Yekhanin, S.: Erasure coding in Windows Azure storage. In: Presented as Part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), Boston, pp. 15–26. ACM (2012)
7. Lucas, R., et al.: Top ten exascale research challenges. Department of Energy ASCAC subcommittee report (2014)
8. Moody, A., Bronevetsky, G., Mohror, K.K., de Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10), New York. ACM (2010)
9. Mu, S., Chen, K., Wu, Y., Zheng, W.: When Paxos meets erasure code: reduce network and storage cost in state machine replication. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14), New York, pp. 61–72. ACM (2014)
10. Nagle, D., Serenyi, D., Matthews, A.: The Panasas activescale storage cluster: delivering scalable high bandwidth storage. In: Proceedings of the SC'04, Pittsburgh, p. 53. ACM (2004). http://dl.acm.org/citation.cfm?id=1049998
11. Peter, K., Reinefeld, A.: Consistency and fault tolerance for erasure-coded distributed storage systems. In: Proceedings of the Fifth International Workshop on Data-Intensive Distributed Computing Date (DIDC'12), New York, pp. 23–32. ACM (2012)
12. Plank, J., Li, K.: Diskless checkpointing. IEEE Trans. Parallel Distrib. Syst. **9**(10), 972–986 (1998)
13. Plank, J.S., Simmerman, S., Schuman, C.D: Jerasure: a library in C facilitating erasure coding for storage applications. Technical report CS-07-603, University of Tennessee Department of Electrical Engineering and Computer Science (2007)
14. Rashmi, K.V., Shah, N.B., Gu, D., Kuang, H., Borthakur, D., Ramchandran, K.: A "Hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. SIGCOMM Comput. Commun. Rev. **44**(4), 331–342 (2014)
15. Rashmi, K.V., Nakkiran, P., Wang, J., Shah, N.B., Ramchandran, K.: Having your cake and eating it too: jointly optimal erasure codes for I/O, storage, and network-bandwidth. In: 13th USENIX Conference on File and Storage Technologies (FAST 15), Santa Clara, pp. 81–94. USENIX Association (2015)

16. Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A.G., Vadali, R., Chen, S., Borthakur, D.: XORing elephants: novel erasure codes for big data. Proc. VLDB Endow. **6**(5), 325–336 (2013)
17. Schmuck, F., Haskin, R.: GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the USENIX FAST'02, Monterey. USENIX Association (2002)
18. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST'10), Washington, DC, pp. 1–10. IEEE Computer Society (2010)
19. Stender, J., Berlin, M., Reinefeld, A.: XtreemFS – a file system for the cloud. In: Kyriazis, D., Voulodimos, A., Gogouvitis, S., Varvarigou, T. (eds.) Data Intensive Storage Services for Cloud Environments. IGI Global (2013)
20. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: 7th Symposium on Operating Systems Design and Implementation (OSDI'06), Seattle, pp. 307–320. ACM (2006)
21. Weinhold, C., Lackorzynski, A., Bierbaum, J., Küttler, M., Planeta, M., Härtig, H., Shiloh, A., Levy, E., Ben-Nun, T., Barak, A., Steinke, T., Schütt, T., Fajerski, J., Reinefeld, A., Lieber, M., Nagel, W.E.: FFMK: a fast and fault-tolerant microkernel-based system for exascale computing. In: Proceedings of SPPEXA Symposium, Garching. Springer (2016)
22. Wende, F., Steinke, T., Reinefeld, A.: The impact of process placement and oversubscription on application performance: a case study for exascale computing. In: Exascale Applications and Software Conference (ESAX-2015), Edinburgh (2015)
23. Zheng, G., Shi, L., Kalé, L.V.: FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In: 2004 IEEE International Conference on Cluster Computing, San Diego, pp. 93–103. IEEE (2004)
24. Zheng, G., Ni, X., Kalé, L.V.: A scalable double in-memory checkpoint and restart scheme towards exascale. In: Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS), Boston, pp. 1–6. IEEE (2012)