

A Dynamic Cloud for the Opportunistic Usage of Frankfurt and GSI Clusters based on HTCondor

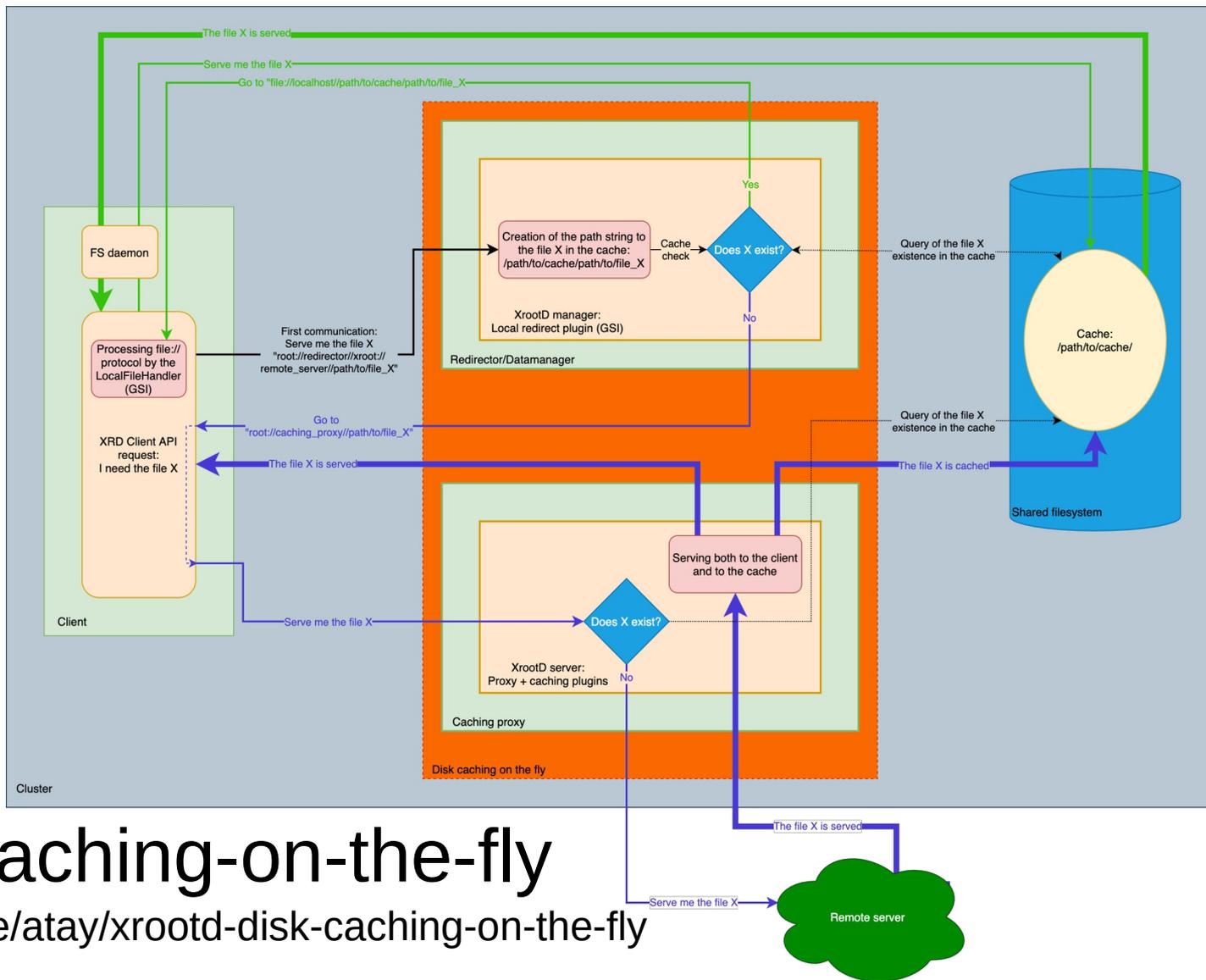
Volker Lindenstruth, Kilian Schwarz, Paul-Niklas Kramp, Dirk Hutter,
Serhat Atay

Goethe University, Frankfurt

GSI, Darmstadt

15 Feb 2022





Disk-Caching-on-the-fly

<https://git.gsi.de/atay/xrootd-disk-caching-on-the-fly>

Singularity Container Image Management

- A shell script (only for ALICE experiment)
 - to fetch the binaries from cvmfs repository for a specific ALICE environment tag and
 - to build a container image with the binaries included
- Prerequisites to run the script
 - A linux operating system (tested for centOS)
 - alice.cern.ch cvmfs repository mount
 - singularity (with superuser or with --fakeroot option enabled)
- Usage
 - `sudo ./singularity_alice.sh -t <environment_tag> -o <working_directory> -y`
 - `-t/--tag` specify a valid environment tag, for example "AliPhysics/vAN-20201018_ROOT6-1"
 - `-o/--output` specify a working directory, default: current directory
 - `-y/--yes` Yes to folder clean up. it will clean the content of working directory.
 - `-h/--help` displays help
- Continuous integration
 - Gitlab runner (being resurrected at GSI)
 - checks if the script is successfully run
 - checks if the container image is successfully run
 - checks if the environment is set inside the container
 - (later) pushes the image to a registry to distribute (continuous deployment)
- Source code with detailed documentation: https://git.gsi.de/atay/alice_singularity
- Generalization for other experiments (next milestone by 30 June 2021)
 - Continuous integration agent is general and can be used for other experiment
 - However, requires similar script for each experiment
 - Any volunteer to write a similar script for another experiment?

Recap for DCOTF and Singularity

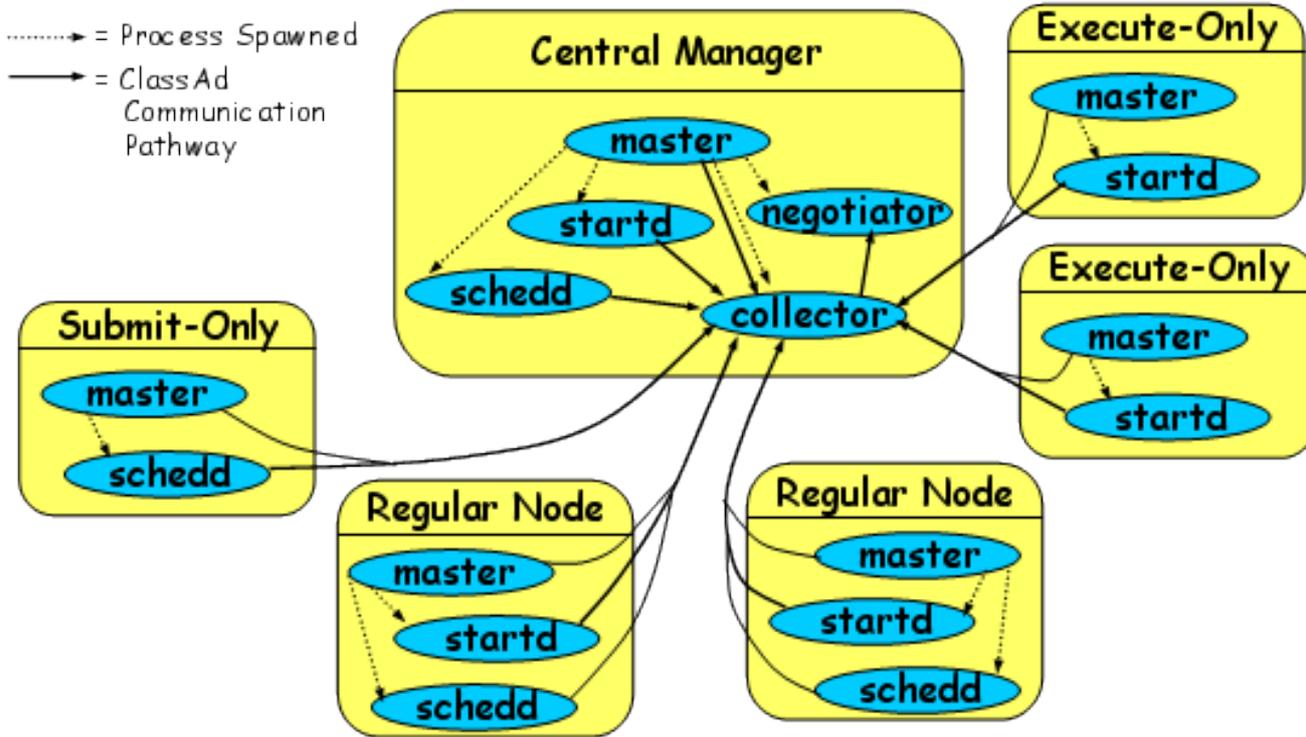
- Environment specific standalone ALICE singularity image script is ready with image management system
- DCOTF and XCache DCA has a similar performance when reading data
 - XCache DCA is natively supported by XRootD
 - DCOTF requires a special XRootD cms plugin and a modification on the caching plugin to alter UNIX permissions of the files in the cache
- XCache creates a bottleneck in terms of CPU and bandwidth for the cached data in a shared filesystem
- Containerization doesn't affect the performance

An HTCondor Dynamic Cloud

- Milestone
 - B4-M2: 30.06.2021: The container system developed in work package A1 is to be expanded to include an interface to HTCondor. The goal is a **dynamic cloud created with HTCondor**, whose jobs running **in singularity containers** extend over the batch systems of the HPC farms of **GSI and the LOEWE-CSC** and their data either from the **"Disk Caching on the fly"** set up in work package A2 "at LOEWE-CSC or directly from the Lustre Cluster of GSI, if necessary via WAN access. Performance measurements are to be carried out.
- An HTCondor dynamic cloud to distribute the jobs over multiple HPC centers with a single queue would fulfill the milestone.

How HTCondor Works

- **master** daemons exist on all the condor machines to make sure other daemons running
- Execute (**startd**) and submit (**schedd**) machines advertise themselves to the central manager (to the **collector**)
- **negotiater** makes the match and notify **schedd** about available **startd**
- **schedd** then submits job to available **startd**

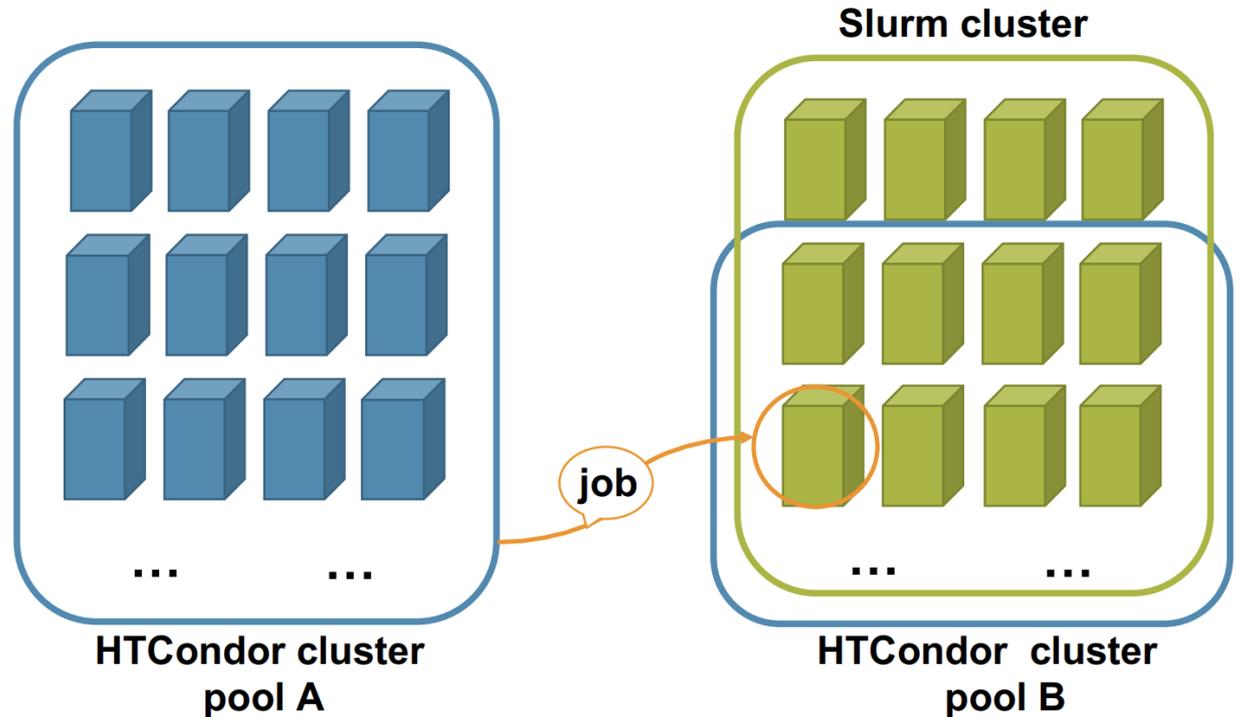


Planning of the Dynamic Cloud

- Requirements
 - First of all, everything must be **transparent** to the users
 - Every user needs to **access to the Job submit portal** on cloud.
 - Jobs must be **submitted as HTCondor jobs**.
 - There must be a **single queue** of jobs
 - Jobs must be able to run **inside the singularity container**
 - Data must be received **from XRootD caching solutions**, e.g. disk-caching-of-the-fly etc.
- All of these can be achieved by the **HTCondor flocking method** over other batch schedulers, e.g. SLURM scheduler

HTCondor Flocking over SLURM

- An HTCondor pool can be flocked over the SLURM cluster
- Jobs waiting in the queue at main HTCondor pool are redirected to new HTCondor execute machines flocked on SLURM pool.



**A Feasibility Study on workload integration between HTCondor and Slurm Clusters*
R. Du et. al. EPJ Web of Conferences 214, 08004 (2019)

HTCondor Connection Brokering

- Distribution of HTCondor execute machines over multiple HPC centers is possible by **flocking method**.
- However, the execute (startd) machines need to have **incoming and outgoing connections** with the submit (schedd) machine on other HPC centers.
- All the HPC centers will **not allow to have incoming connection** to the worker nodes from outside
- HTCondor Connection Brokering (CCB) is the answer to overcome this issue without necessity of incoming connection to the WNs of HPC centers.
- Upon advertisement of the execute machines to the central manager, CCB **keeps the TCP connection alive** to allow further communication e.g. file transfer for the job and the output.

An HTCondor Dynamic Cloud for ErUm Project

- An HTCondor central manager and the submit machine is set on a machine in cloud with public IP (erum-cloud). Any user can login to that machine from the internet.
- HTCondor execute machine daemons are started on GSI and Goethe-HLR SLURM WNs to join the HTCondor pool that runs on erum-cloud.
- A script running on GSI and Goethe-HLR clusters allocates (and also releases) the SLURM WNs depending on demands from erum-cloud HTCondor pool.
- Any HPC center which allows the outgoing connections can join the HTCondor pool that runs on erum-cloud machine with the correct configuration.
- Jobs can be submitted inside the singularity containers. Images will be sent to execute machines via open TCP connection
- If the caching proxy address is prefixed to the data path, then data are served via DCOTF. (XRootD client plugin can make this process transparent to the user)

SLURM Job Submit Script

- If there are jobs waiting on the HTCondor job queue, it will allocate a new SLURM WN and start condor execute daemons
- It checks all the previous SLURM jobs if they run any condor jobs and flags the SLURM WNs without condor jobs for the next check
- If a flagged SLURM WNs is not running any condor job, it releases the SLURM WN to the SLURM pool.
- This process is repeated at each 5 minutes
 - to dynamically allocate a SLURM WN to provide resources to condor jobs, and
 - not to waste too much resources from SLURM pool in case of no running condor jobs.
- Slurm script can be found [here](#)

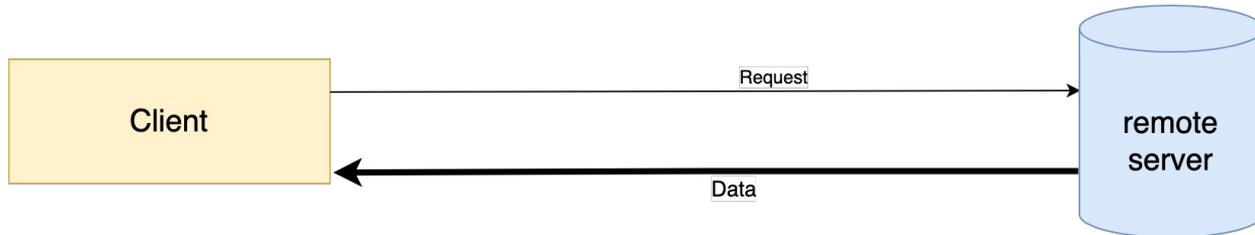
Summary

- DCOTF performs as same as XCache with direct cache access.
- ALICE environments have a singularity file to generate standalone images with all necessary binaries inside.
- A prototype HTCondor dynamic cloud machine successfully manages a single condor job queue over multiple SLURM (and possibly many other) job scheduler.
 - Condor jobs can also run inside singularity due to native singularity support of HTCondor
 - Data can be retrieved from DCOTF if the path specified
 - DCOTF can be used as default for data access if the XRootD client plugin is used

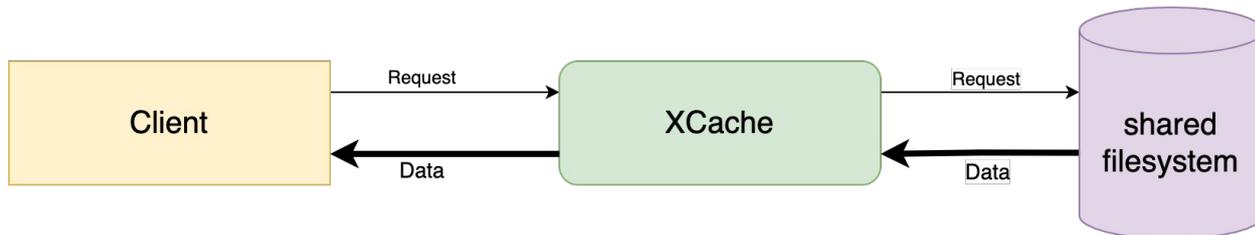
Thanks

Cache Scenarios for Benchmarks

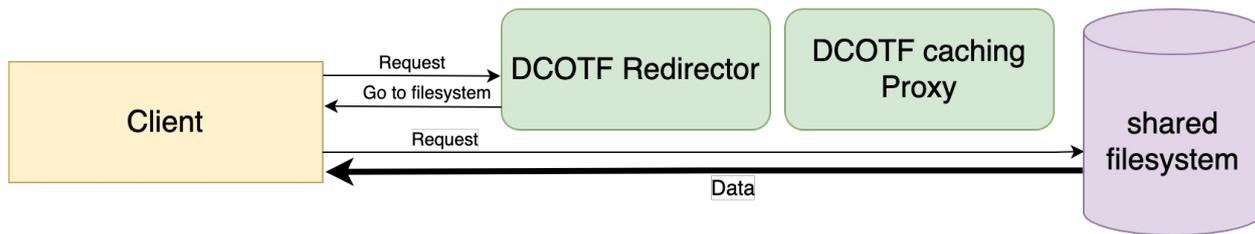
- No cache



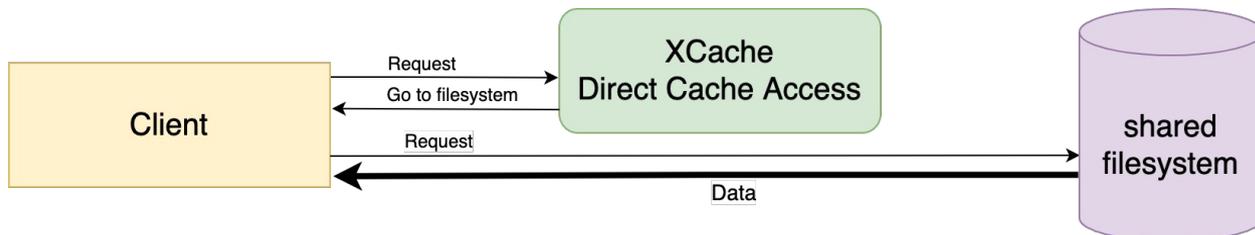
- XCache, data in cache



- DCOTF, data in cache

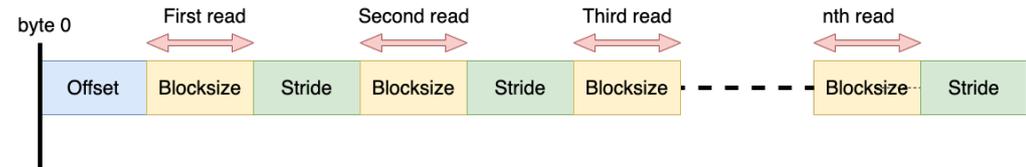


- XCache DCA, data in cache

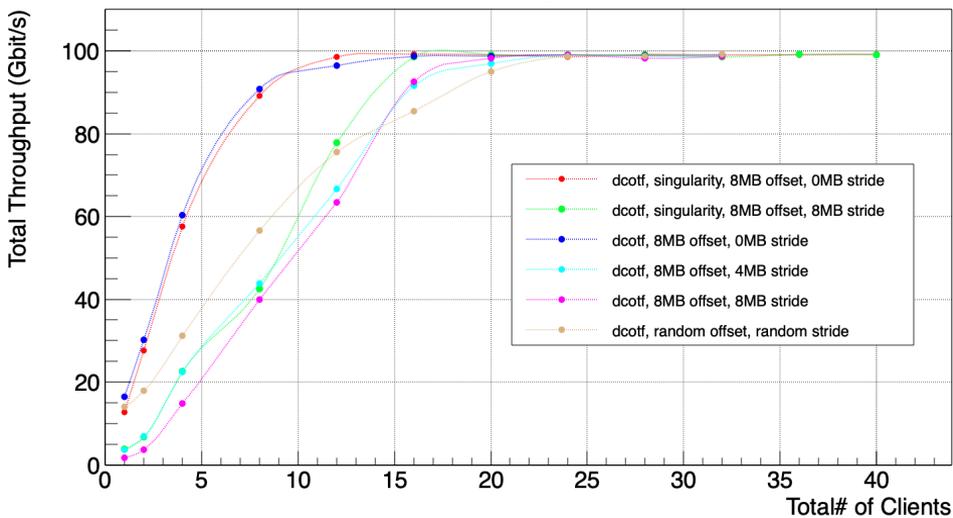


Generic Read Benchmark with XrdCl

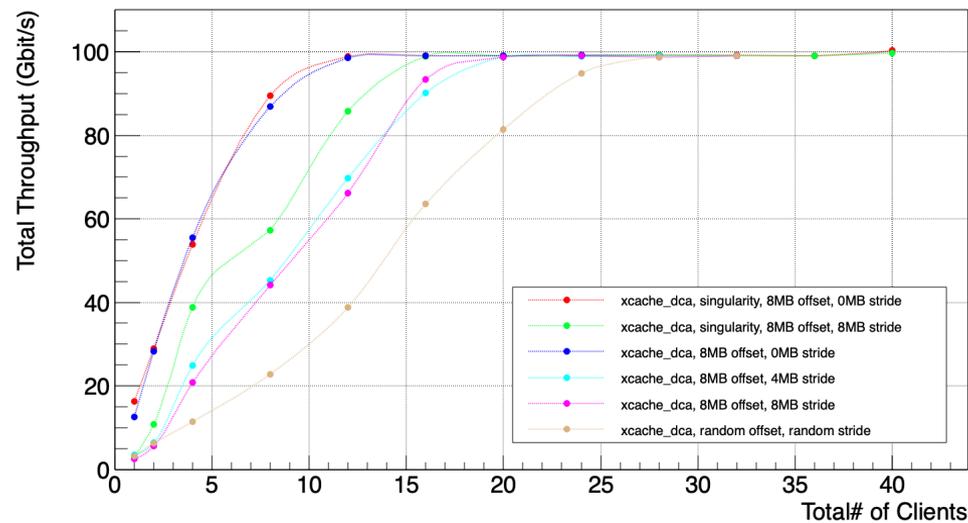
- In Frankfurt (Goethe-HLR)
 - SLURM scheduler
 - Local cache location under BeeGFS filesystem at Goethe-HLR with infiniband fabric
- Caching node: node48-004, client nodes: node48-[001-003]
 - 2x (20 core, 40 thread) CPUs, 192GB RAM, EDR link (100Gbit/s)
- Blocksize: 8MiB (8.38MB)
- Plots in Gbit/s (10^9 bits)
- Datasets at GSI Lustre filesystem with 10Gbit/s link to Goethe-HLR
 - Files filled with `"/dev/urandom"` at Lustre
 - 80 x 50GiB files (for DCOTF and XCache DCA) or
 - 160 x 25GiB files (for XCache, **memory flush with 200GB file between SLURM jobs**)
 - with different names (to avoid memory caching in the node)
 - with exactly same contents
- Reading algorithm
 - No multiple read of the same blocks,
 - offset in each iteration: $\text{offset} + (\text{blocksize} + \text{stride}) * \text{counter}$
 - measuring interval: 1 second (# blocks / second)
 - Measurement time: 20-100 second
- Time stamp and read interval matching
 - Reading time and interval are matched with respect to system time when summing up clients



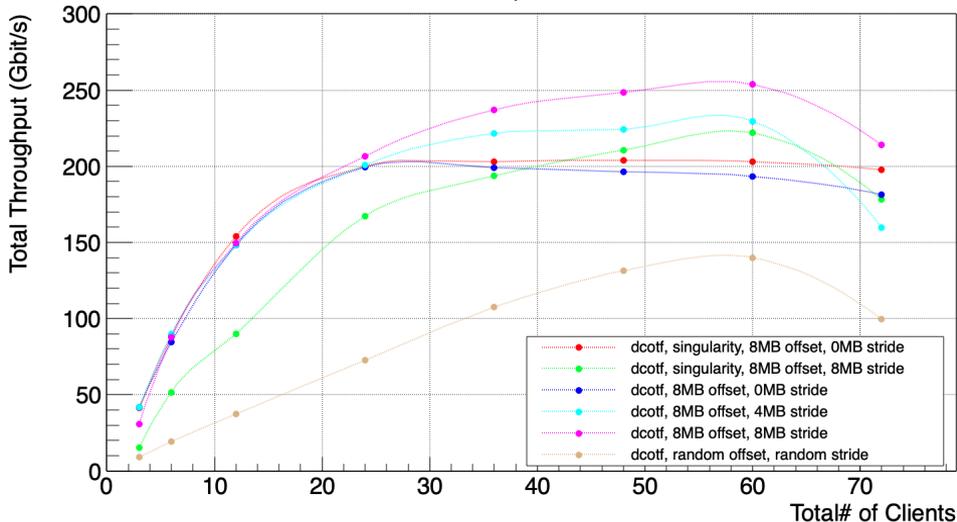
DCOTF, 1 NODE



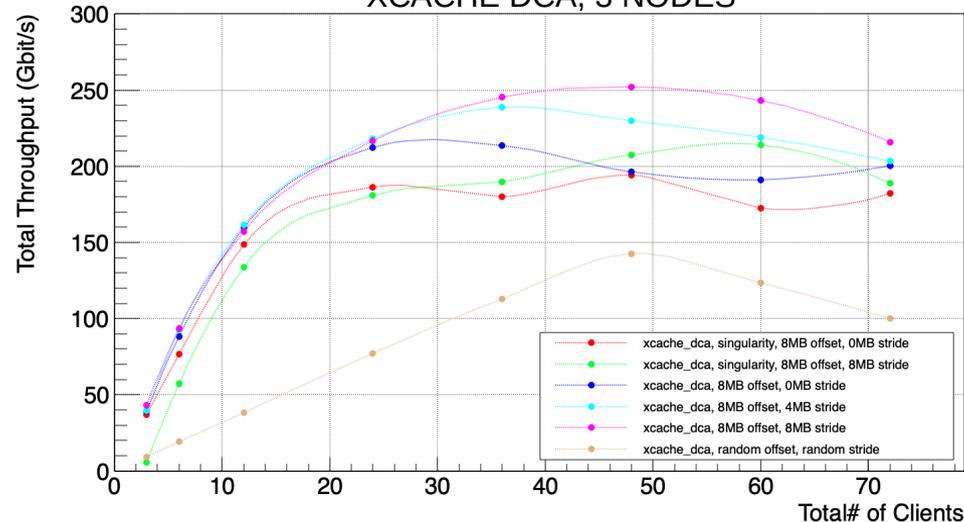
XCACHE DCA, 1 NODE



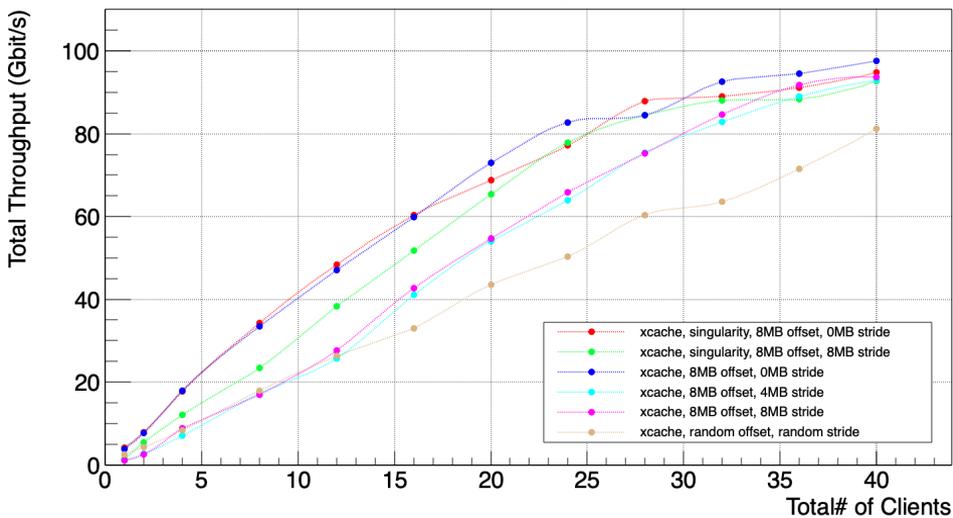
DCOTF, 3 NODES



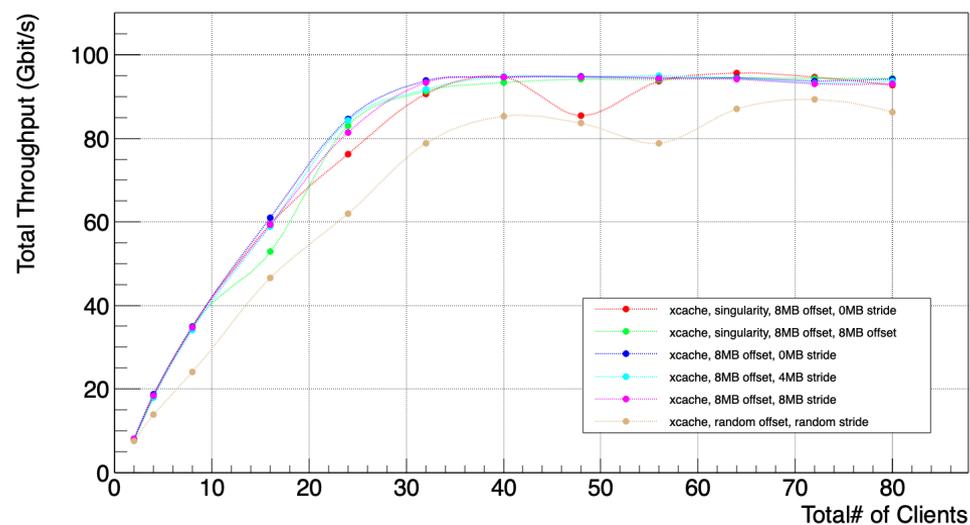
XCACHE DCA, 3 NODES



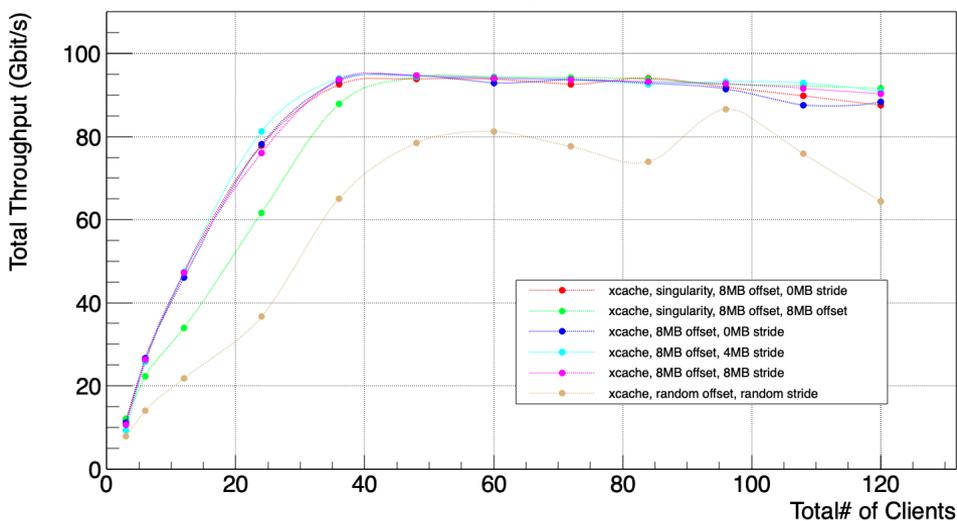
XCACHE, 1 NODE



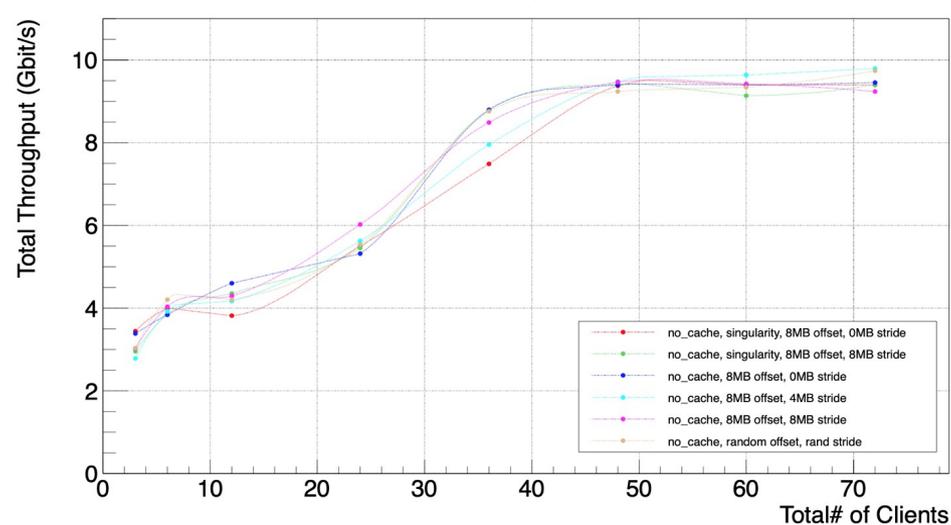
XCACHE, 2 NODES



XCACHE, 3 NODES



NO CACHE, 3 NODES



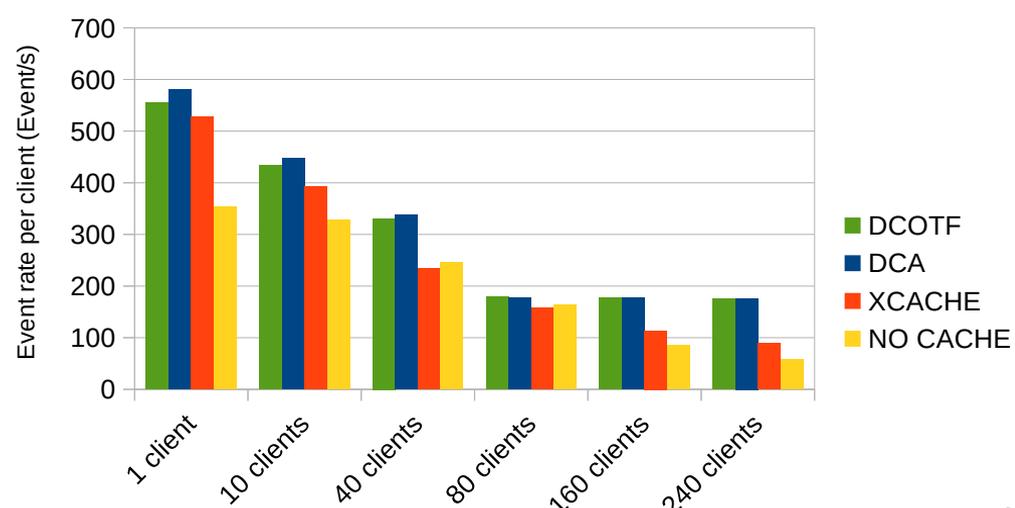
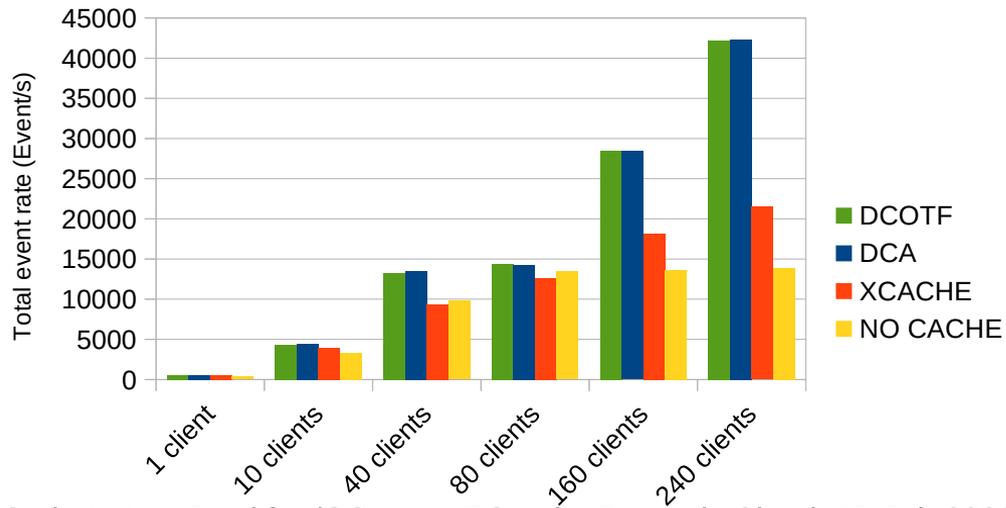
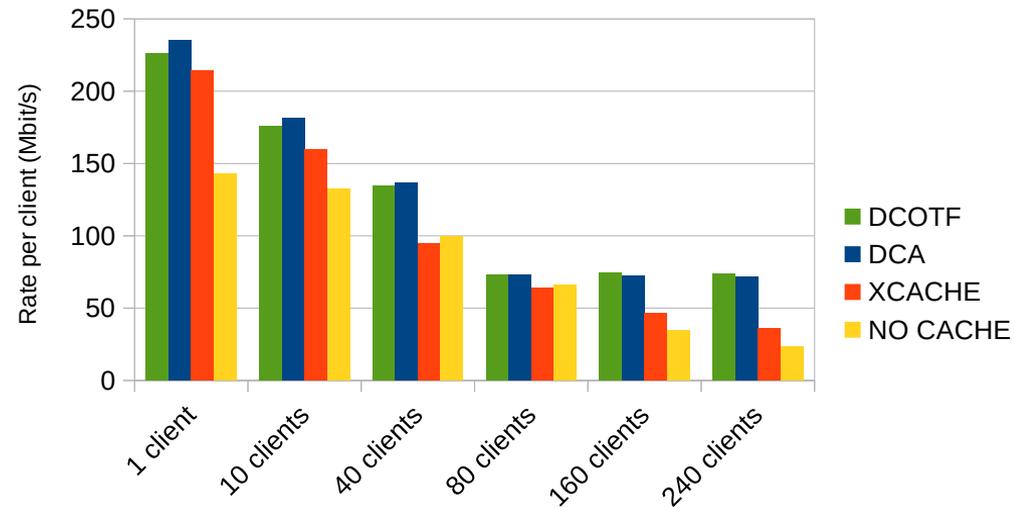
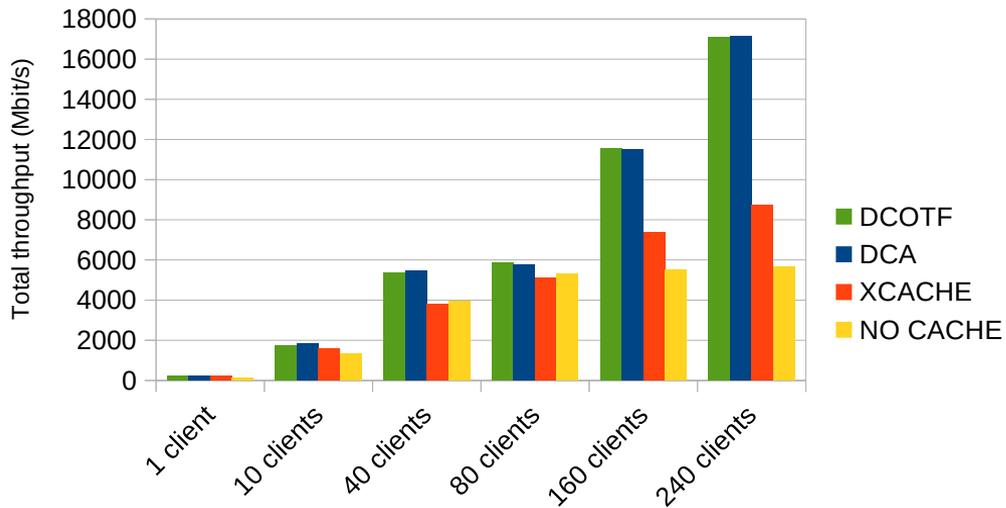
Results for Micro Benchmark

- Successful saturation of the node and filesystem bandwidth for DCOTF and XCache DCA
- XCache server with slightly lower bandwidth of the caching node (~95Gbit/s)
- Saturation 10Gbit/s link between GSI and Frankfurt without cache
- Highest rate per client with 0MB stride
- Similar results with singularity container
- Lower read performance with random stride
- Filesystem bandwidth
 - Highest observed: 251Gbit/s (during easter holiday)
 - Depends on the actual usage of the filesystem by other users
 - No filesystem throughput history available to compare. (admins do not store it)
 - Aggregated filesystem BW on the Goethe-HLR website: 216 Gbit/s (27 GB/s)
 - It does not pose any kind limitation of filesystem according to admins
 - Reduction of the filesystem bandwidth with higher number of clients (for # of client>~60)

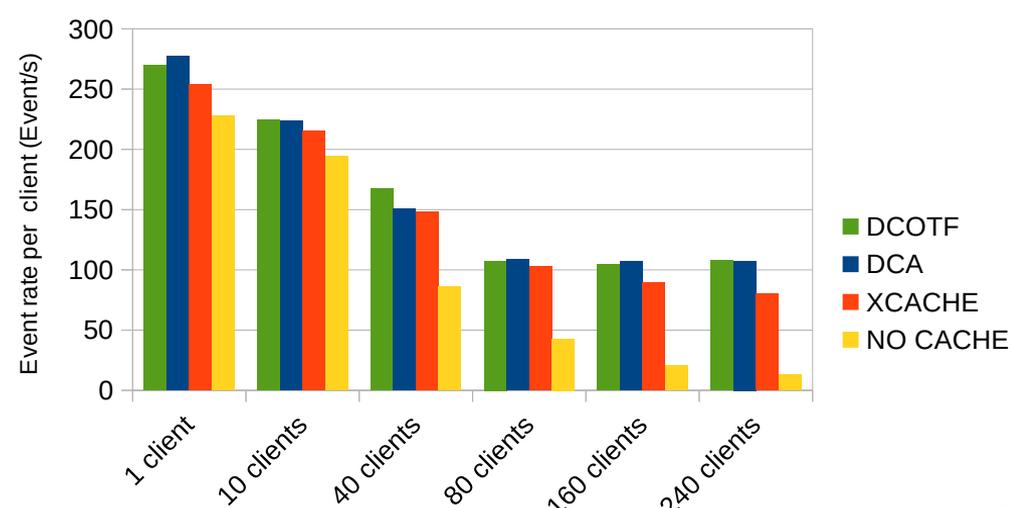
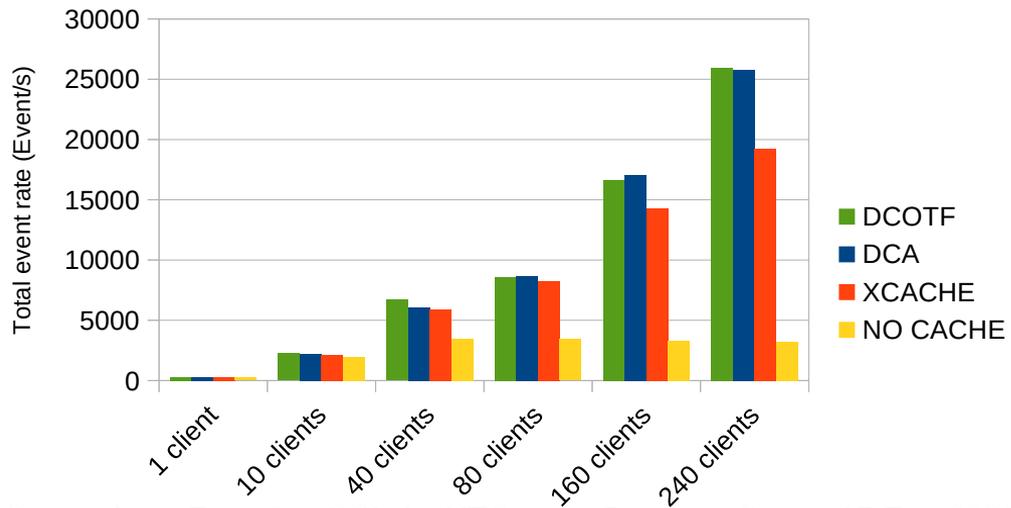
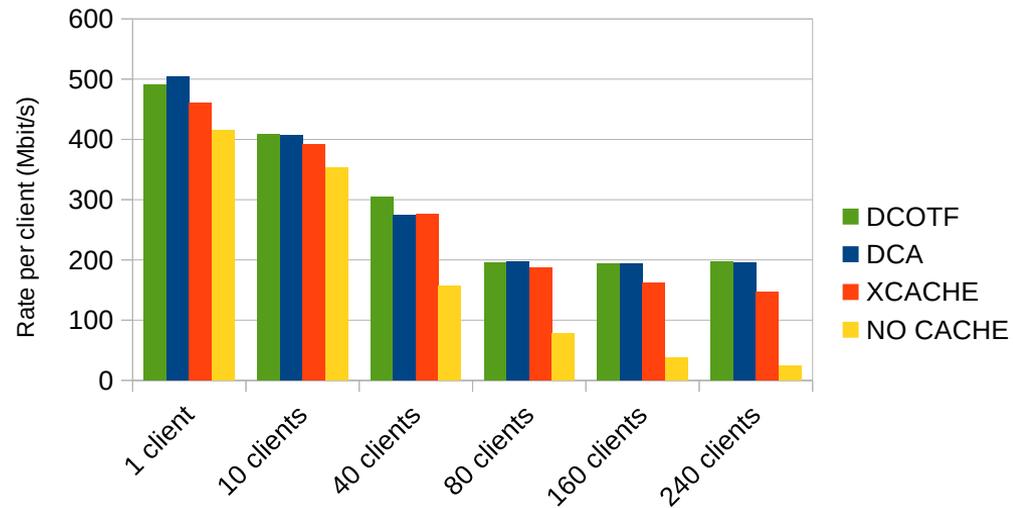
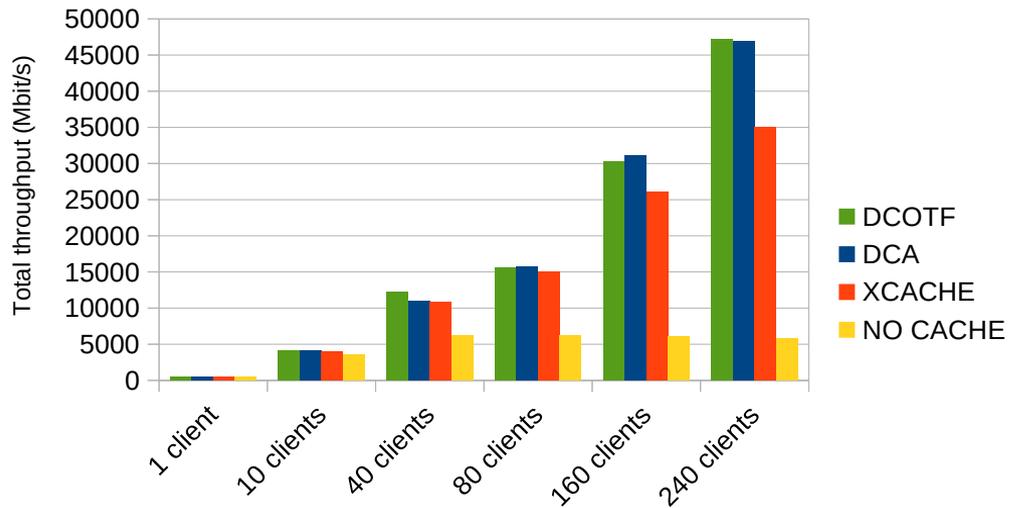
Analysis Benchmarks (only in Singularity)

- In Frankfurt (Goethe-HLR)
 - SLURM scheduler
 - Local cache location under BeeGFS filesystem at Goethe-HLR with infiniband fabric
- Caching node: node48-004, client nodes: node48-[001-003]
 - 2x (20 core, 40 thread) CPUs, 192GB RAM, EDR link (100Gbit/s)
- “No cache” measurements with old router for 10Gbit/s link
 - router at GSI → [router in Frankfurt](#) → worker nodes: ~6Gbit/s due to CPU limitation of router in Frankfurt
- Analysis setup
 - ALICE AOD and ALICE ESD datasets
 - 10530MB / 25990 events AOD data (CPU intensive analysis tests)
 - 31457MB / 17290 event ESD data (data intensive analysis tests)
 - 1, 10, 40 and 80 client on single node (node48-001)
 - 160 clients on 2 nodes, 240 clients on 3 nodes
- Sample analysis reads a branch from data and fill a histogram
 - removed filling histogram to increase throughput
 - now only reads events from a branch and does nothing further (>80% of each file)

ALICE AOD Data in Singularity



ALICE ESD Data in Singularity



Results for Analysis Benchmark

- Similar performance DCOTF and XCache DCA
 - Comparable performance with XCache up to # of total threads on the caching node (80 clients)
- AOD data
 - ~17Gbit/s achieved (240 tasks)
 - ~47k event/s achieved (240 tasks)
 - Maximum ~240Mbit/s per client (1 client) and ~72Mbit/s after CPU saturation
 - Maximum ~580 event/s per client (1 client) and ~190 event/s after CPU saturation
- ESD data
 - ~47Gbit/s bandwidth achieved with ESD data (240 tasks)
 - ~26k event/s achieved (240 tasks)
 - Maximum ~490Mbit/s per client (1 client) and ~200Mbit/s after CPU saturation
 - Maximum ~280 event/s per client (1 client) and ~110 event/s after CPU saturation
- “No Cache” is limited to 6Gbit/s, thus has the worst performance among all