HEP Application Delivery on HPC Resources

August 2016

Author:
Tim Shaffer

Supervisor(s):
Jakob Blomer, Gerardo Ganis

CERN openlab Summer Student Report 2016

Project Specification

High-performance computing (HPC) contributes a significant and growing share of resource to high-energy physics (HEP). Individual supercomputers such as Edison or Titan in the U.S. or SuperMUC in Europe deliver a raw performance of the same order of magnitude than the Worldwide LHC Computing Grid. As we have seen with codes from ALICE and ATLAS, it is notoriously difficult to deploy high-energy physics applications on supercomputers, even though they often run a standard Linux on Intel x86_64 CPUs. The three main problems are:

1. Limited or no Internet access;

2. The lack of privileged local system rights;

3. The concept of cluster submission or whole-node submission of jobs in contrast to single CPU slot submission in HEP.

Generally, the delivery of applications to hardware resources in high-energy physics is done by CernVM-FS [1]. CernVM-FS is optimized for high-throughput resources. Nevertheless, some successful results on HPC resources where achieved using the Parrot system[2] that allows to use CernVM-FS without special privileges. Building on these results, the project aims to prototype a toolkit for application delivery that seamlessly integrates with HEP experiments job submission systems, for instance with ALICE AliEn or ATLAS PanDA. The task includes a performance study of the parrot-induced overhead which will be used to guide performance tuning for both CernVM-FS and Parrot on typical supercomputers. The project should further deliver a lightweight scheduling shim that translates HEP’s job slot allocation to a whole node or cluster-based allocation. Finally, in order to increase the turn-around of the evaluation of new supercomputers, a set of "canary jobs" should be collected that validate HEP codes on new resources.

[1] http://cernvm.cern.ch/portal/filesystem

[2] http://cernvm.cern.ch/portal/filesystem/parrot

Abstract

On high performance computing (HPC) resources, users have less control over worker nodes than in the grid. Using HPC resources for high energy physics applications becomes more complicated because individual nodes often don't have Internet connectivity or a filesystem configured to use as a local cache. The current solution in CVMFS preloads the cache from a gateway node onto the shared cluster file system. This approach works but does not scale well into large production environments. In this project, we develop an in-memory cache for CVMFS, and assess approaches to running jobs without special privilege on the worker nodes. We propose using Parrot and CVMFS with RAM cache as a viable approach to HEP application delivery on HPC resources.

Table of Contents

1 Introduction

CernVM-FS (CVMFS) is a central component of CernVM, developed to address the needs of large-scale, high-throughput computing for high-energy physics (HEP) experiments. CVMFS facilitates software deployment onto heterogenous computing resources by making experiment sofware and tools available without the need for installation or maintenance on numerous compiler-OS platforms [1]. Since CVMFS is freely available for use both inside and outside CERN, it is difficult to determine the number of active users. Based on web server logs, which show accesses during CernVM boot, there are an estimated 1.2 million virtual machine reboots per month, with about 200,000 of these being fresh boots. The worldwide userbase, including grids, clouds, and HPC resources, is estimated at 100,000 clients. Based on feedback from users, CVMFS is in use across the majority of the LHC experiments. By using CVMFS, researchers can deploy a small (on the order of 100MB) image and transparently load the current experiment software, greatly simplifying software distribution and version management.

2 Design of CVMFS

CVMFS was developed to distribute large amounts of content for HEP research at a global scale. To handle the enormous amount of data produced by the LHC, hundreds of thousands of machines spread across potentially thousands of worldwide sites must obtain and update experiment software stacks via CVMFS. CERN makes use of a layered hierarchy of distribution servers to achieve global scale [2].

Unlike a traditional local filesystem, which balances reading and writing, CVMFS assumes that content, once published, is immutable. This approach greatly simplifies caching and consistency issues. CVMFS internally uses CAS (content addressable storage); data and metadata are identified by the cryptographic hashes of their contents. Any changes will produce a different hash value, so multiple versions can coexist in the same repository without conflict. Content changes are released as new snapshots, so clients always have a single, consistent view of the repository. In addition, this approach gives clients the options of using the most recent snapshot, or any previous version. In this way, CVMFS simplifies software management and enables reprocucibility, since all previous versions of an experiment’s software can be accessed and used at any point in the future.

CVMFS was also designed for compatibility with existing web infrastructure, and as such uses HTTP to transport content and metadata. Since CVMFS serves read-only content that never expires, the existing web caching infrastructure is well-integrated with the design of CVMFS. Sites can use commodity web servers running freely available software to add a local cache layer. This is particularly important for computing centers, where a proxy cache on a fast local network can greatly improve performance and decrease load on the public servers.

On top of an HTTP transport, CVMFS presents repository content via POSIX filesystem interface. When a client reads a file hosted in CVMFS, a locally running program receives the request and fetches the data over the network. Once the content is downloaded and verified, the local CVMFS program can pass the data through the filesystem. Client programs can operate on this filesystem interface as if the entire CVMFS repository were locally installed. This approach makes frequent updates to HEP software stacks manageable without rebuilding virtual machine/container images or introducing software version inconsistencies. It also obviates the need to push large updates to each and every client node, since the CVMFS program running on each client can fetch new versions of individual files (or parts of large files) as needed.

A final design consideration to discuss here is ease of deployment. Since the LHC experiments make use of computing resources at sites across the globe, it is unreasonable to obtain administrator privileges on every machine. This rules out certain possibilities, such as kernel modifications or virtual machine deployment. CVMFS uses FUSE (Filesystem in USErspace), a software interface that allows non-privileged users to create filesystems. CVMFS presents a virtual filesystem that serves data from remote CVMFS repositories rather than local storage.

CVMFS makes use of extensive caching to improve client performance and reduce load on servers. Repositories have very fine-grained control over how content is distributed [3]. A single Release Manager Machine is responsible for updating a CVMFS repository, which can then be served via web servers, content delivery networks (CDNs), caching proxies, load balancers, etc. The read/write server, the Stratum 0 server, does not directly serve client requests. Instead, the Stratum 0 server feeds content to the Stratum 1 servers, a handful of geographically distributed web servers. As long as the content is duplicated across the Stratum 1 servers, CVMFS offers a robust content delivery network. In the event of an outage, clients can fail over to another available server.

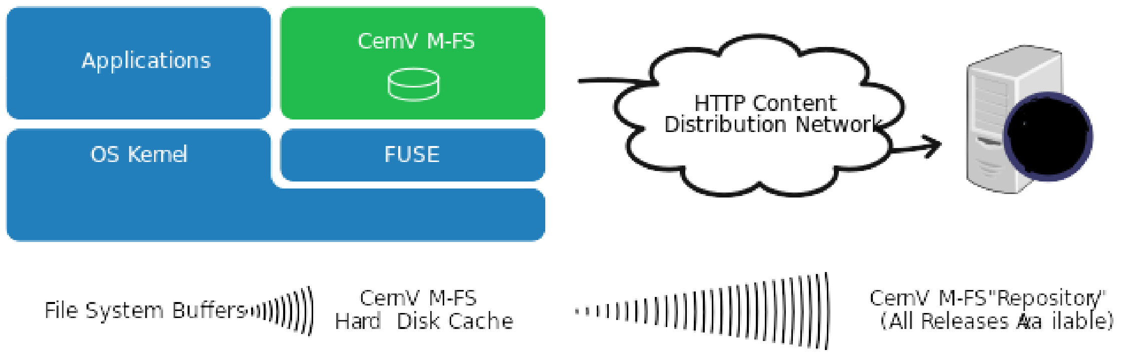

Applications
CernVM-FS
OS Kernel
FUSE
HTTP Content Distribution Network
File System Buffers
CernVM-FS Hard Disk Cache
CernVM-FS "Repository" (All Releases Available)

Figure 1: The overall architecture of CVMFS

Repositories can fine-tune their distribution systems to achieve optimal performance for their workloads. Closer to clients, it is recommended to run a cluster-wide or network-wide proxy cache both to reduce external traffic and to decrease the time to load content. In addition, each client maintains a local, filesystem-backed cache so that previously accessed files can be handled without waiting for a network request. There is some overhead in passing filesystem requests to a FUSE module, so even directly serving files from the local CVMFS cache is slower than direct disk access. The operating system kernel maintains a cache of very recently accessed data and can handle cached requests directly without consulting the FUSE module. The kernel cache allows some filesystem operations to approach the speed of local disk operations. The overall architecture of the FUSE module is shown in Figure 1. From the client's perspective, the Stratum 1 servers and cache layers make up a single CVMFS "repository". The client is responsible for fetching, assembling, verifying, and caching the data served by the CVMFS repository.

3 CVMFS in HPC

High-performance computing (HPC) resources present some additional obstacles to the deployment of HEP applications. These include restrictions on network access, limited control of the execution environment, and a different job scheduling scheme.

Since CVMFS typically requires internet access as well as some local storage for the cache, a modified approach is necessary. The currently recommended setup requires a shared filesystem for the worker nodes [4]. The logon node, which has internet access, can preload the necessary content onto the shared cluster filesystem. Worker nodes can then treat the shared filesystem as another cache directory and retrieve content without connecting to the internet. When spinning up workers in a cluster, however, there is a risk of drowning the shared filesystem. During startup or heavy use, a shared cluster filesystem might not be able to directly handle the large number of synchronized metadata operations (millions per node within a few seconds) such as path lookups generated during cluster operation. Clients must make use of local caches to resolve most operations locally rather than risk overwhelming the shared infrastructure.

The existing local cache stores data and metadata objects in a cache directory by their hash IDs. As new objects are retrieved, they are verified and written into the cache directory. Subsequent accesses are served directly from the local cache. If the local cache is configured with a size limit, CVMFS can periodically remove unused objects from the cache to fit the cache size. Thus to use CVMFS, it is necessary to configure a cache directory with read and write access. For HPC resources, however, this is not always feasible. Worker nodes may not have writable scratch space available for use as a cache for CVMFS.

HPC clusters may also give users limited privileges on the workers nodes. Jobs may not make system-wide changes, such as modifying the kernel or the root filesystem. In addition, FUSE is often not available on the worker nodes. Most HEP applications stacks are written with the assumption that CVMFS repositories will be available under /cvmfs, so FUSE may not be a viable solution in HPC environments.

4 Parrot

Parrot is a tool developed at the University of Notre Dame for attaching existing programs to remote I/O systems through a filesystem interface. The predecessor to CVMFS, GROW-FS, was developed as part of Parrot to deliver simulation software for the (now concluded) Collider Detector at Fermilab (CDF) experiment [2]. CERN later began developing CVMFS to address performance and scalability issues in the original GROW-FS. Today, Parrot supports CVMFS as a remote service.

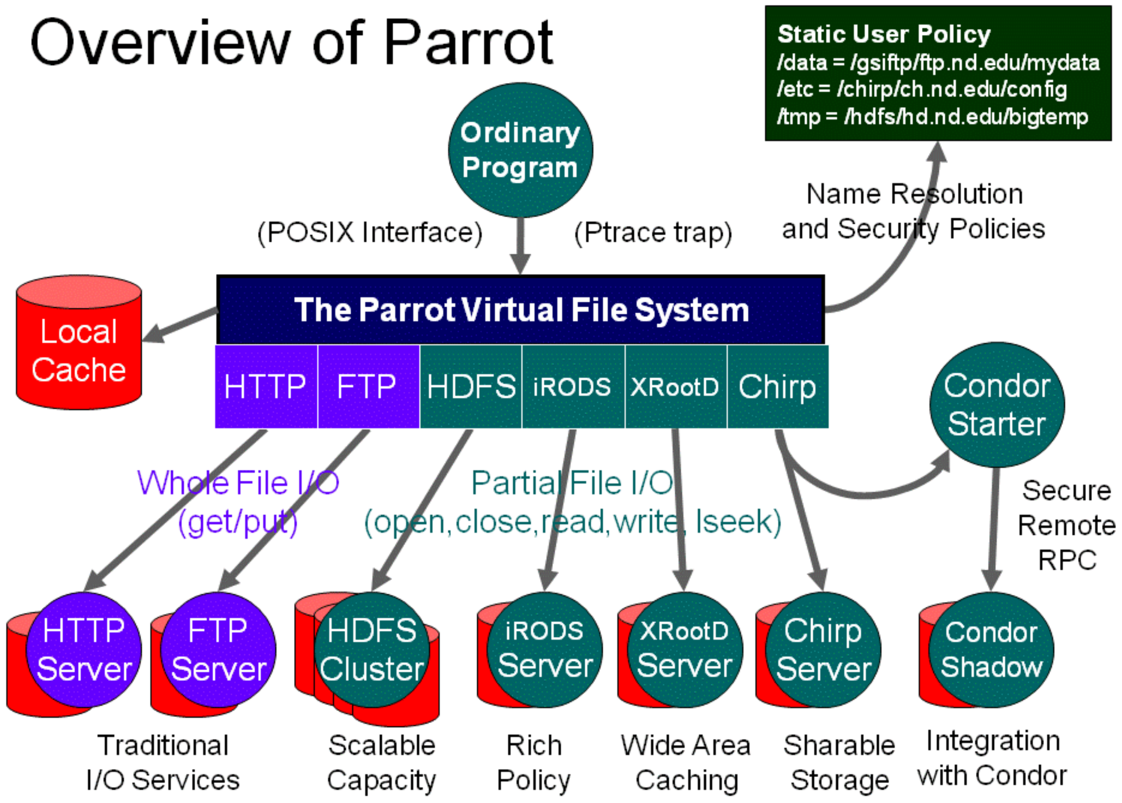


Figure 2: The overall architecture of Parrot

Rather than interacting with the Virtual Filesystem Switch within the kernel as FUSE does, Parrot makes use of the ptrace interface [5]. Profiling and debugging programs such as gdb use ptrace to monitor and interact with other programs as they run. The ptrace interface allows a tracer process to attach to a running process and intercept syscalls, signals, and other system-level events. Parrot uses this functionality to intercept and rewrite I/O requests such that remote services appear to be installed in the filesystem. As shown in Figure 2, Parrot can serve as a central connector for multiple remote services. This is a good fit for scientific applications, as Parrot allows access to remote services without requiring every program to link against and configure every service. ptrace does not require kernel modifications, root access, setuid helpers, etc. ptrace allows much more general changes to a process's view of the system (rearranging the filesystem, changes to uid/gid, etc.) and does not require any special privileges, but Parrot intercepts all syscalls, not only filesystem-related interactions. Thus processes suffer additional overhead on non-filesystem operations, compared to processes accessing a FUSE mount.

5 RamCache

As part of this work, we refactored the existing local cache infrastructure to support alternative caching implementations in CVMFS. In the future, additional caches can easily be added. The existing cache implementation is now available as the POSIX cache, and CVMFS includes a new cache implementation, the RAM cache.

To ensure that a cache entry is not evicted mid-operation, cache accesses are mediated by handles. To begin using an existing cache entry, the caller must open the entry to obtain a handle. Subsequent operations make use of this handle to identify the cache entry, and when the entry is no longer needed, the handle must be closed. Only entries with no outstanding handles are subject to eviction from the cache. To add new cache entries, callers write their content into a memory region allocated for the transaction, and then atomically commit the entry to the cache. If a commit would exceed the configured cache size, the cache manager tries to free sufficient space by evicting entries. Entries with open handles are not considered, but other entries are evicted in LRU (Least Recently Used) order. CVMFS caches also support the notion of pinned and volatile entries. Pinned entries are never evicted and only explicitly removed from the cache. The primary use case for pinned entries is file catalogs. To ensure that metadata operations can be handled locally, the catalog(s) for a repository are always kept in-cache. Data needed to start analysis but not needed thereafter, such as conditions data, can be marked volatile to indicate that it should be evicted from the cache before regular entries are evicted. The RamCache provides a file descriptor-like interface and transactional semantics, but the actual data storage is handled by a key/value store implementation. This separation is present so that a cache manager can make use of other (or multiple) storage backends.

6 KvStore

The newly implemented RamCache for CVMFS depends on a thread-safe key/value store to manage the actual object data. CVMFS objects are identified by the cryptographic hashes of their contents. Thus a simple key/value store is well suited to storing CVMFS data and metadata objects. The KvStore implementation written for RamCache adds reference counting to protect entries from deletion. By default, the key/value store uses the malloc() implementation provided by libc to allocate memory. An alternative arena-based allocator is also available. This allocator acquires a sequence of large blocks of memory (arenas) from the operating system and tries to fill each as completely as possible. For some usage patterns, these memory allocators may exhibit suboptimal performance due to fragmentation. Another allocator that supports compaction is, at the time of writing, being developed and integrated into the KvStore.

7 Benchmarks

To assess the performance and viability of the new cache system in CVMFS, we performed measurements on some of the standard experiment benchmarks. These benchmarks are based on simulation jobs for several of the LHC experiments (ATLAS, ALICE, and CMS are used here). The benchmarks load and initialize their full experiment software stacks, and generate some small data. To measure performance on larger jobs, we added another benchmark that simulates a larger number of events from the CMS benchmark. Measurements were taken on a dedicated CernVM test machine.

The four benchmarks were measured with every combination of cache implementations (POSIX, RAM-malloc, RAM-arena), filesystem interface (FUSE, Parrot), and kernel caching behavior (enabled, disabled). As preliminary measurements, we took 7 or more samples for each of these 36 benchmark configurations and computed means and standard deviaitions of job running times.

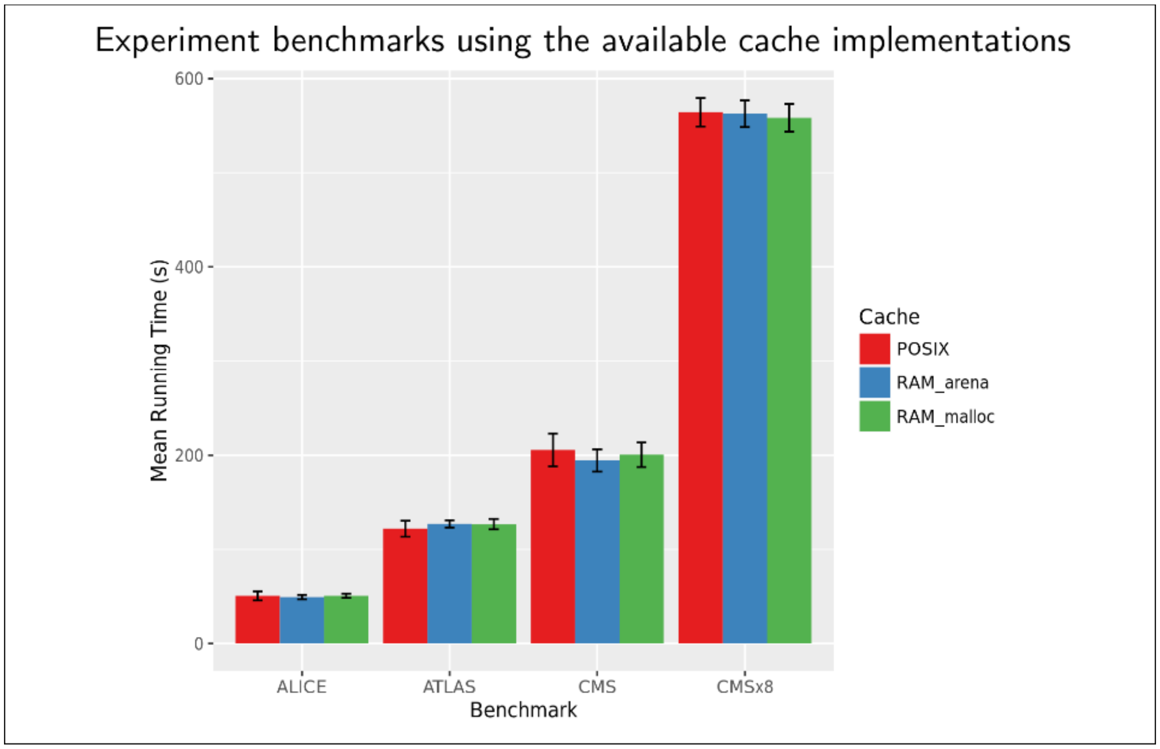


Figure 3: Benchmark running times on FUSE with kernel caching disabled

Figure 3 gives a comparison of cache implementations under FUSE with kernel caching disabled. The results under Parrot are quite similar. The performance of all three cache configurations is comparable. Additional measurements would be necessary to discern any performance difference.

The two shorter benchmarks, ALICE and ATLAS, do relatively little simulation work, so these measurements are more representative of startup times. With further optimization, we hope that the RAM cache could outperform the POSIX cache, but these preliminary measurements suggest that the new cache implementation can match current overall performance.

Especially for shorter tasks, the kernel cache has a much larger performance impact. When the FUSE module or Parrot connector handle filesystem requests, there is significant cost in context switches, IPC, etc. If kernel caching is enabled, the kernel can handle some of the requests without involving the userspace components at all.

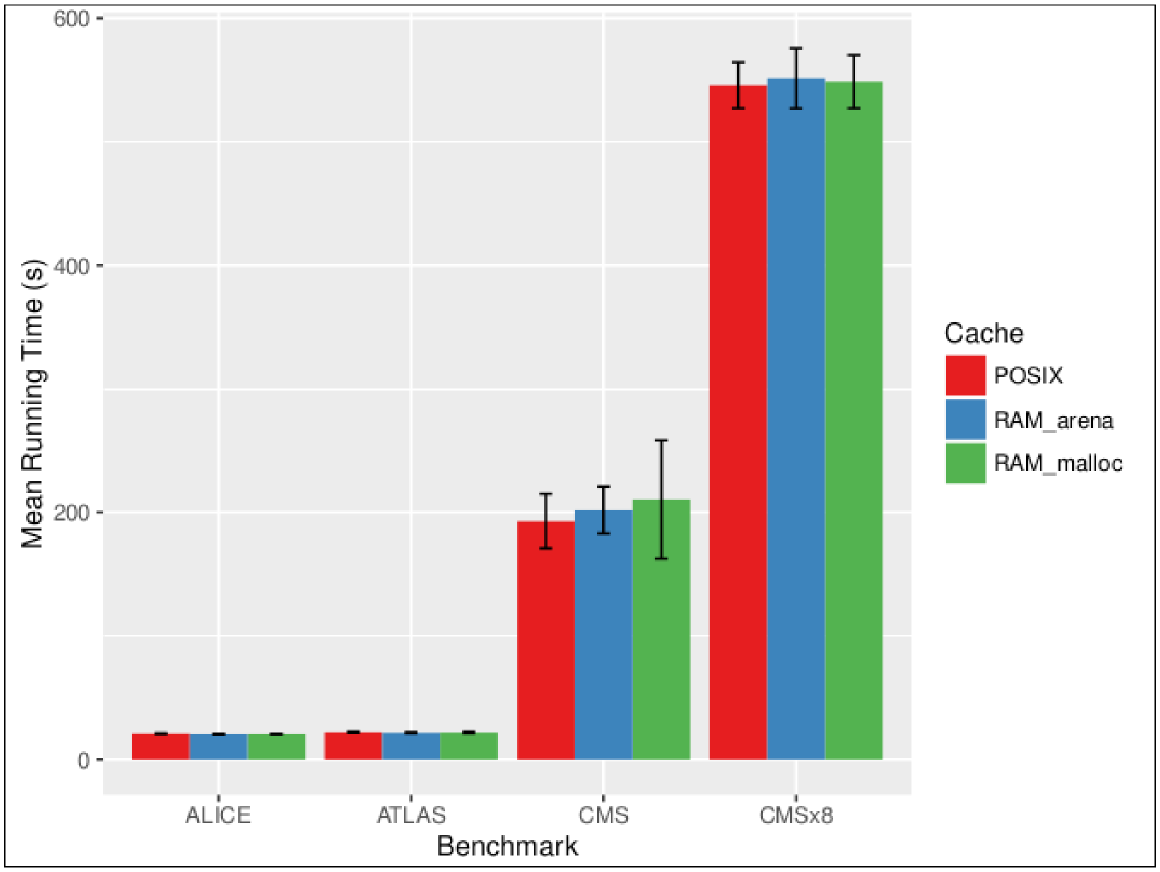


Figure 4: Benchmark running times on FUSE with kernel caching enabled

In Figure 4, we observe that the running times for the ALICE and ATLAS benchmarks decrease significantly with kernel caching enabled. In this case, the FUSE module must fetch each file on the first access, but the kernel can directly handle subsequent activity. Thus we see that these two benchmarks do little work beyond loading their application stacks. The CMS and CMSx8 benchmarks, on the other hand, show less improvement from kernel caching. This is likely because these benchmarks spend more time doing simulation work, so I/O performance is not as significant.

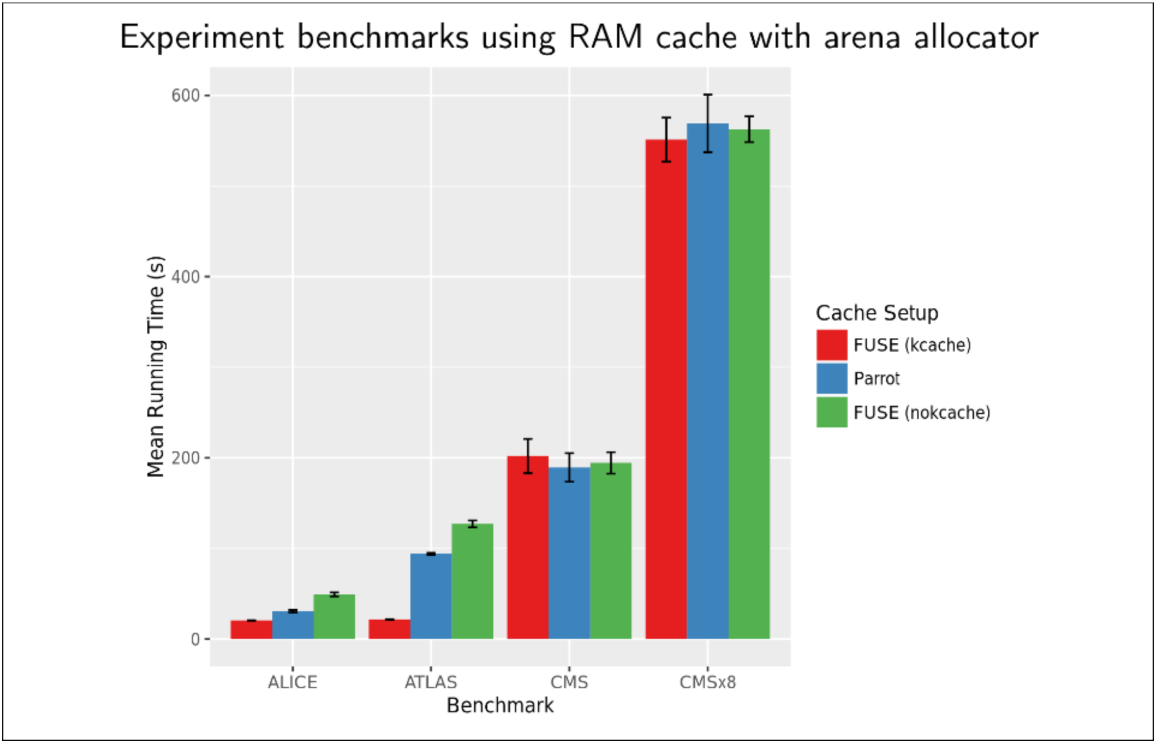
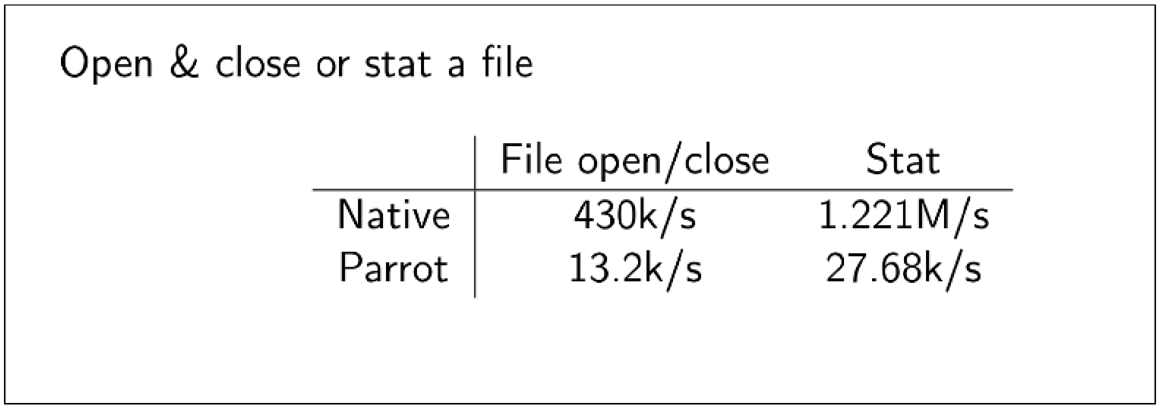


Figure 5: Benchmark performance under FUSE and Parrot

When comparing the performance of the filesystem interfaces in Figure 5 for a representative cache configuration, we see the previously discussed performance increase with the kernel cache on the ATLAS and ALICE benchmarks. For these two, Parrot's performance is intermediate between the kernel caching modes, but appears to be closer to uncached FUSE. As the simulation time increases with CMS and CMSx8, however, the three filesystem configurations show approximately the same performance. This suggests that for compute-bound tasks like simulation, running under Parrot does not introduce an undue performance penalty compared to either FUSE configuration.

8 Microbenchmarks

While not necessarily representative of real-world performance, microbenchmarks are useful for assessing performance limits. Measuring file open/close and stat operations show a substantial slowdown under Parrot.

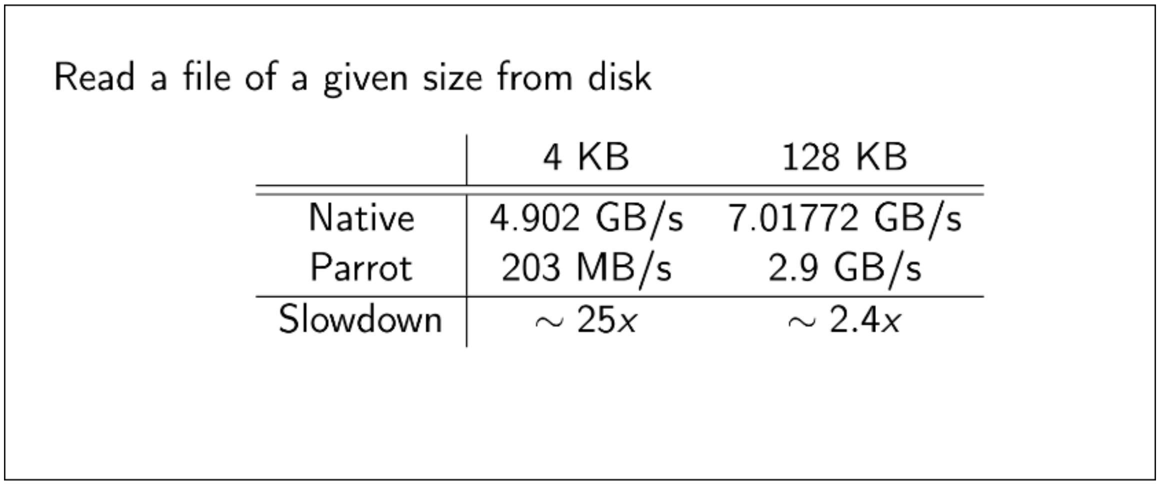


Open & close or stat a file

	File open/close	Stat
Native	430k/s	1.221M/s
Parrot	13.2k/s	27.68k/s

Figure 6: Open/close operations

The microbenchmark in Figure 6 is nearly pure syscalls, so this is something of a worst case for Parrot's performance. In addition to the overhead from the syscall, Parrot has its own processing, and often makes a syscall itself in processing a request. Since stat is such a common operation, Parrot's performance might benefit from better metadata caching.

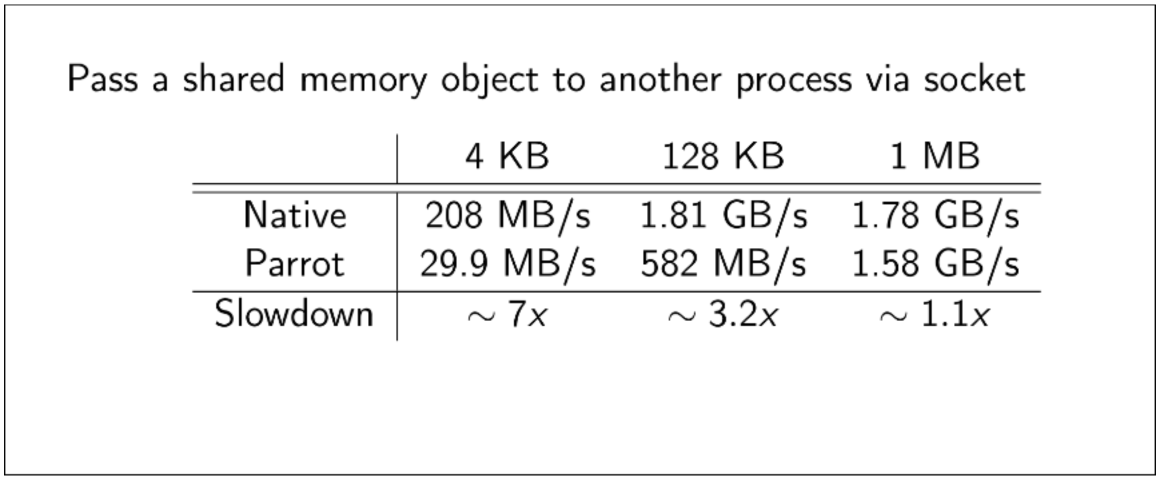


Read a file of a given size from disk

	4 KB	128 KB
Native	4.902 GB/s	7.01772 GB/s
Parrot	203 MB/s	2.9 GB/s
Slowdown	$\sim 25x$	$\sim 2.4x$

Figure 7: File reads

When examining read performance in Figure 7, we again observe a significant slowdown under Parrot. This effect is especially acute on the small 4K reads. With larger reads, more data is read per syscall, so the bandwidth is much improved. Any computation or other processing that a realistic program performs between reads would similarly reduce the slowdown.



Pass a shared memory object to another process via socket

	4 KB	128 KB	1 MB
Native	208 MB/s	1.81 GB/s	1.78 GB/s
Parrot	29.9 MB/s	582 MB/s	1.58 GB/s
Slowdown	$\sim 7x$	$\sim 3.2x$	$\sim 1.1x$

Figure 8: IPC via shared memory

In the microbenchmark in Figure 8, each iteration involved receiving a file descriptor to a shared memory object over a Unix domain socket, mapping it into the test process' address space, making a local copy, and unmapping the shared memory. In this case, the number of syscalls is indepent of the amount of data transferred, so while a slowdown is observed for small memory objects, Parrot's slowdown nearly vanishes as data size increases. As with many of the previous measurements, this suggests that programs that can minimize the number of syscalls relative to computation can achieve acceptable performance under Parrot.

9 Conclusions and Future Work

Running high energy physics applications on high performance computing resources presents some distinct challenges compared to existing deployments. HPC environments often impose restrictions on the network and filesystem access, as well as privileges on worker nodes. By adding an in-memory cache in addition to the existing filesystem-backed cache, CVMFS can now run efficiently without write access to the local filesystem. With an HTTP cluster proxy, a shared cluster filesystem is not necessary either. Finally, measurements running actual experiment software stacks suggest that Parrot is a viable choice for running compute-bound jobs such as simulation without administrative privileges.

For future work, we would like to expand the RAM cache manager to support additional memory allocators, for example a log-structured allocator. Another important addition is support for clients on a single cache manager instance. One possibility is RamCloud, which can provide a low latency, in-RAM key/value store distributed over a cluster. A locally shared cache would allow multiple Parrot instances to efficiently share cache space on a worker node. Work Queue [6], another software component developed at the University of Notre Dame, allows worker nodes to run job instances. This is a promising direction for efficiently running slot allocated HEP jobs on whole-node allocated HPC resources. Work Queue can be leveraged to schedule multiple jobs using the full resources on each node, presenting a grid-like scheduling interface on top of any available resources. Over the course of this project, we updated and expanded the CVMFS benchmark jobs to run under the new cache configurations, and under Parrot. These benchmarks test multiple experiment frameworks, and give an indication of performance under real workloads. Therefore, we propose using these benchmarks to validate both correctness and performance on new computing platforms. Using Work Queue, we can easily run batches of these benchmark jobs on a new cluster, which gives a preliminary indicator of how the cluster will behave under load. At the time of writing, we are discussing the possibility of testing these benchmarks and some realistic workflows using CVMFS on real hardware at NERSC.

10 References

[1] G Ganis et al.; 2015 J. Phys.: Conf. Ser. 664 022018 "Status and Roadmap of CernVM"

[2] J Blomer et al.; Computing in Science and Engineering 17(6) 61-71, "The Evolution of Global Scale Filesystems for Scientific Software Distribution"

[3] https://cvmfs.readthedocs.io/en/latest/cpt-overview.html

[4] https://cvmfs.readthedocs.io/en/latest/cpt-hpc.html

[5] http://ccl.cse.nd.edu/software/parrot/

[6] http://ccl.cse.nd.edu/software/workqueue/