Efficient unpacking of required software from CERNVM-FS

AUGUST 2018

AUTHOR:
P. Samuel M. Teuber

EP-SFT
CernVM-FS

COLLABORATORS:

Nicholas Hazekamp

SUPERVISORS:

Jakob Blomer
Gerardo Ganis

CERN openlab

Project Specification

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

- Limited or no Internet access
- The lack of privileged local system rights
- 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 [7] CernVM-FS is optimized for high-throughput resources. Some successful results on HPC resources where achieved using a combination of techniques [5], including a variety of ways of pre-populating a shared file system or container caches with the required files from CernVM-FS. However, some of these solutions are still not optimal. In particular, the cache pre-population techniques use a very conservative set of files which may result in huge container images [15]. From the analysis of the effective cache population, one can expect space for improvement (or size reduction) of one or two orders of magnitude. Building on the above considerations, the project aims to prototype and implement an extension of a tool allowing for an optimal population of the CernVM-FS caches for a given application workflow. The task includes a detailed study of the cache population to capture the essential needs of a given workflow, and the prototype/implementation of an algorithm to be used, for example, in the preparation of the relevant container images. The student will also get involved in testing and validating the new tool on a set of representative supercomputers.

Abstract

In recent times a tool for efficient unpacking of software work-flows from CernVM File System (CVMFS) into standalone images has become necessary.
There are two types of use cases for such images: On the one hand they can be used to deliver HEP software to compute nodes which do not support the traditional CVMFS delivery architecture (in particular HPC compute nodes). On the other hand the images can be used for benchmarks in which the network connectivity should not influence the benchmark results.
We present a new software utility which allows CVMFS-sourced image creation and synchronization. The resulting standalone images can run HEP applications independently from an internet connection. We further provide a methodology for automatically extracting the resource requirements from a given software work-flow through file access tracing.
In first experimental evaluations the new tool outperformed mechanisms traditionally used for this task like rsync and provides finer grained unpack functionalities than uncvmfs.

Contents

Contents iv

1 Introduction 1

2 Current state of CVMFS in HPC environments 2

3 Towards a workflow for exporting software images from CVMFS 3
3.1 Requirements analysis: Tracing . . . 3
3.2 Specification building . . . 3
3.3 Image production . . . 4

4 Design and Implementation 5
4.1 Tracer . . . 5
4.2 Shrinkwrap utility . . . 5
4.2.1 Image architecture . . . 5
4.2.2 File System Traversal . . . 5
4.2.3 IO Thread Pool . . . 7
4.3 Docker Injection . . . 7

5 Use Cases 8
5.1 ATLAS at NERSC . . . 8
5.2 Benchmarking Containers . . . 8

6 Performance evaluation 9

7 Conclusions 12
7.1 A software solution for shrinkwrapping . . . 12
7.2 Future work . . . 12

Bibliography 14

Appendix 16

A Reproducibility 16

Acknowledgements

First and foremost, I would like to thank Nicholas Hazekamp, my co-intern on this project, for the incredibly productive work in the past weeks. I really enjoyed our collaborative work on this project!

Further I would like to thank both of my supervisors, Jakob Blomer and Gerardo Ganis, for their great support during my entire stay at CERN and for always being there to answer any questions that came up in the development process.
I would also like to thank them for selecting me in the first place and putting the trust in me to work on this project.

I would like to thank both the IT Department (in particular Domenico Giordano) as well as the US-ATLAS collaboration (in particular Doug Benjamin and Wei Yang) for providing use cases to test our software utility.

I would also like to thank Radu Popescu, my office colleague, for enduring the endless project discussions between Nick and me.
In general, I would like to thank the entire EP-SFT department as well as the CERN Openlab team for allowing me to spend my summer at CERN in Geneva working on this exciting project and for their support during the stay.

A final word of thank you goes to the fellow Openlab Summer Students (in particular my room mates) for making this summer such a unique experience.

1. Introduction

The CernVM File System (CVMFS) [4] is a distributed global scale network file system that is broadly used for software delivery in the High Energy Physics (HEP) community (especially at CERN). It is mainly optimized for typical software package workloads which consist of many relatively small files. While exact user numbers are unknown due to its proxy hierarchy it is estimated that CERN's CVMFS installation is used to access software distributions on some 64k nodes [3] in 160 sites of the Worldwide Large Computing Grid (WLCG) [2]. CVMFS can therefore be considered one of the main software distribution systems inside the HEP community around CERN.

In recent years there has been growing interest in using High Performance Computing (HPC) installations for HEP calculations. However, deploying HEP software to HPC resources like Cori, NERSC [1] (USA) still poses a big challenge to the current CVMFS delivery architecture.
Three of the main challenges for bringing HEP software to HPC resources are limited access to internet from the compute nodes inside the HPC center, lack of a local hard disk which could be used for file caching and the lack of system access necessary to use technologies like FUSE [9], [5].
While there are some ways of avoiding the need for an active internet connection or the need for a local hard disk [5], [12], the task of HEP software deployment becomes notoriously difficult once the underlying system does not allow mounting through FUSE. Due to these circumstances there has been a growing interest in exporting software stored in CVMFS into stand-alone software images.

Another reason for the development of the export utility has been a growing interest in building benchmark containers for system evaluations. In these cases, it is desirable to have a container of minimal size which runs exactly the same work-flow on every system without relying on external factors like the internet connection for its execution. The export utility therefore perfectly fits this use case with its ability to export previously examined workflows into standalone images.

The presented software utility (cvmfs_shrinkwrap) allows the export of software stacks from CVMFS into standalone images. The software functionality represents a super set of the uncvmfs [14] utility, which is currently used in HEP for exporting images from CVMFS repositories, since it allows the export of repository subsets (instead of an entire repository).
The utility currently permits exports to POSIX folders but also to tarballs, SquashFS images or even Docker Images stored in OCI registries [6] through a POSIX export followed by a packaging step which produces the desired image format. Just like uncvmfs the cvmfs_shrinkwrap utility uses hardlink deduplication for the image building to minimize storage needs for the resulting read-only images.

2. Current state of CVMFS in HPC environments

A more detailed analysis on CVMFS in HPC environments so far can be found in [5]. While typical HPC applications are carefully optimized for a given environment and often statically linked, High Throughput Computing (HTC) applications for HEP usually should be executable on a multitude of different Linux flavors provided within the WLCG. For this reason, HEP software on CVMFS has a long tradition of being delivered as a bundle containing all software components, libraries, and compilers necessary to run a given software work-flow. This results in software releases with sizes of some Gigabytes and a large number of small files in the 10-100 kilobyte range.

Due to security concerns the FUSE module of the Linux kernel is deactivated in quite a few HPC systems (especially in HPC centers operated by the US DOE). As an alternative to FUSE, CVMFS can be used through a Parrot [13] interface on such compute nodes. This only has a small performance impact on application execution which is negligible for many HEP applications. However, this method can be difficult in use cases which use multithreading or GPU calculations. The performance of CVMFS further heavily relies on caching at compute node level. The lack of a dedicated disk for each compute node makes this difficult in the case of HPC systems. As an alternative a shared directory can be used as a common CVMFS cache for all compute nodes. This often leads to a very high number of small files which are difficult to handle for typical HPC file systems. Alternatively, one large image file per compute node can be used that is mounted in the compute nodes file system as a loop back device. Furthermore, RAM caches in HPC compute nodes have been explored as an alternative to local file system caches [12].

To circumvent the need for an active internet connection, a preloader can make use of an internet-connected login node to produce a full mirror of the necessary CVMFS repositories in a shared directory of the HPC storage system. This is more time efficient than the use of applications like rsync due to the use of Merkle Trees in the synchronization of the repository. Another approach used so far is the unpacking of the entire CVMFS tree into a fat image [15] which can be used as a normal mounted directory or container image using a tool called uncvmfs [14]. Even when making use of file system level deduplication through hardlinks such images easily amount to some 100GB of storage which makes the build process tedious and the resulting images difficult to deliver with the tooling currently used.

3. Towards a workflow for exporting software images from CVMFS

The proposed work-flow for CVMFS-sourced image exports consists of 3 steps which were all implemented within the scope of this project:

1. Requirements analysis
2. Specification building
3. Image production

For certain cases (i.e. cases where the required files are well known) the first step can be omitted, and one can start with the creation of a (then handwritten) specification in step 2 instead.

While the specification building is neither strictly necessary (the export of an entire repository is supported as well), it typically makes a lot of sense not to export the entire repository, but to pre-select the files that are actually needed. This is because most CVMFS repositories contain software packages for a variety of platforms and architectures of which most are not actually needed for a software run in a specific, known environment.

All features described in the following section will be released with CVMFS 2.6.

3.1 Requirements analysis: Tracing

To allow for an easy analysis of the necessary requirements a tracing utility was implemented inside the regular CVMFS client. This tracing utility, once activated, will write a per-repository log on all system calls performed that were redirected to CVMFS (i.e. all CVMFS paths accessed). Running one or various similar software work-flows with an activated tracer will offer a detailed view on the paths required by the given work-flow in an exported image for a successful software run. Each trace will contain both the accessed path and the executed operation so that different ”depths” of file use are distinguishable. To obtain a stable, reliable image for a given work-flow later on it is highly preferable to run a few variations of the same work-flow to cover a larger number of the possible software behaviors (and files necessary to support these).

3.2 Specification building

Given the result of a trace log we implemented two basic algorithms for automated specification building. These algorithms will parse a trace log file and produce a specification containing all the paths necessary for a successful software run. Depending on the use case different levels of additional coverage can be used: The specification can for example either contain exactly the files that were accessed by the software or the directories from which the software read. This way we can approximate different levels of variability in the software’s work-flow and account for this variability with further extended or very precise specifications.

3.3 Image production

The image export is at the core of the described work-flow and can be invoked through the cvmfs_shrinkwrap command. The utility is based on the official CVMFS library libcvmfs and allows the multithreaded export of files from CVMFS to an abstract interface. Currently the utility only contains an interface for POSIX which can then be extended through packaging the POSIX directory into various other image formats (tar, SquashFS, ext4 loopback, Docker images, ...).

It is important to note that an image can also be updated once a first version has been built. This is also true for the case of Docker images in OCI registries. That is, the utility can also be used to synchronize an earlier exported image with the current version of the software inside the CVMFS repository. This should permit faster update times of the images in comparison to a full reexport, once a first version of the image has been created since only the files that actually changed need to be updated.

4. Design and Implementation

4.1 Tracer

The tracing utility has been implemented in a way that minimizes the overhead if the tracer is deactivated and also minimizes the measurable overhead in the FUSE calls for an activated tracer by delegating the actual log writing to a separate thread to avoid blocks during the accesses to CVMFS.
In fact, for a deactivated tracer the entire overhead consists of a single inline if for some of the FUSE calls.

4.2 Shrinkwrap utility

The shrinkwrap utility has been designed with the objective to allow fast exports even for meta data intensive workloads (i.e. a large number of small files) as this is a typical use case for CVMFS [4].

4.2.1 Image architecture

The architecture of an exported image is very similar to the file system architecture of CVMFS repositories in general. The image will typically contain a directory named cvmfs which in turn will contain any exported repositories as directories.

The cvmfs directory further contains a data directory which is used for the creation of content addressable hardlinks to all files inside this image. Since all images produced are supposed to be used as read only images, this allows us to deduplicate all files stored in the image by saving them once under its content addressable name and then hardlinking to this file from the directory structure as it is also done in uncvmfs [14]. It has previously been shown that this greatly helps to reduce the total number of files in CVMFS and this functionality can therefore also be very useful for images exported from CVMFS.

During file comparisons for synchronization the content addressable hardlinks can also be used to allow for a more efficient comparison in the POSIX case without a need for hashing the files in the destination file system. For this a file entry including its content hash is retrieved from the source file system (i.e. CVMFS) and the stat information for both the content addressable file as well as the entry in the repository are fetched in the destination file system. Afterwards the inode numbers of the two destination entries can be compared. If the 2 files have the same inode it can be assumed that the file has not changed and still corresponds to the file stored in the source file system. However, if the 2 files have different inodes the file has changed in a recent update on the source side and needs to be updated.

4.2.2 File System Traversal

A large amount of the work has been spent on the design of the file system traversal, various speed ups for it, as well as the design of an abstract interface which hides the implementation details of a specific destination file system. The following section describes the design decisions

made in this process as well as various optimizations that were implemented to improve the system's performance.

File System Interface

The objective for the file system interface had been to come up with a generic interface which can be used to integrate the software with various export formats. For this reason, the interface does not only include methods necessary for the file system traversal and entry creation/removal, but also a dedicated read/write interface that offers an abstract interface to a file's sequential read-/write operations.
Keeping in mind the image design described above the interface makes a clear distinction between (unique) file creation (and writing) and the hard linking of these files into the repositories. The traversal further delegates the file comparison (apart from basic stat information) to the interface allowing for file system dependent optimization's like the inode comparison for POSIX described above.

Specification based entry selection

With the introduction of automated specification building we expect specification files which might be multiple magnitudes of order larger than current specifications in the dirtab format [8]. In addition to this there was a need for other types of requirement specifications than previously supported (e.g. the inclusion of a folder without filled sub directories).
Therefore, a new system for specification parsing and matching was implemented in order to reduce lookup times and support these new types of specifications.

Specification language The specification language contains 3 types of inclusion and one type of exclusion:

- **Inclusion of a single file/empty folder**
  Will make a simple copy of the file/directory and can be specified through: `/abc/def/foo.txt`
- **Inclusion of a directory with all entries, non-recursive**
  This will include a directory as well as all entries inside the directory without recursing into other subdirectories. Can be specified through: `~/abc/def/dir/*`
- **Inclusion of a directory with all entries, recursive**
  This will include the entire directory and recurse into subdirectories. Can be specified through: `/abc/def/*`
- **Exclusion of an entry** This will exclude the entry (and any child directories) from the export. Can be specified through `!/abc/def/entry`

Specification parsing During the parsing of the specification the software builds a hash table based in-memory tree of the entries inside the directories labeled with the corresponding inclusion/exclusion mode. During parsing the software can make use of partially sorted specification trees through caching of the last entries used during parsing in a stack. This is useful because the result of the automated specification tool is usually partially sorted.
For a file containing 10.000 lines of path specifications this software will parse the file in roughly $30ms$.

Specification lookup For the lookup the algorithm traverses the hash table tree up to the tree entry which represents the longest prefix of the given path and checks for the inclusion mode of this entry. This allows lookup times dependent on the longest path in the specification instead of lookup times dependent on the number of specification entries. Indeed, our microbenchmark showed lookup times of around $3\mu s$ independent of the specification size.

In memory directory listings Making use of the hash tables inside the tree structure described above, the software can produce directory listings for non-wildcard directories in the specification without the need to rely on the source directory listings. This can be done entirely in memory without libcvmfs access The file system traversal then makes use of this functionality thereby also avoiding the treatment of not specified directories by dropping them from the directory listings from the start on.

4.2.3 IO Thread Pool

The export of files from CVMFS to images is a very IO-intensive task. During the export a lot of time is spent on the actual data transfer from CVMFS into the corresponding image. This makes an entirely sequential program very inefficient at this task and makes the use of a thread pool essential in achieving good export performance.

The parallelization model employed for the image export consists of one Master thread which traverses the file system and a thread pool doing the file copying work. The Master thread will traverse the file system, compare the found files and - if necessary - submit copy jobs to a pipe from which the threads can then read the jobs to do the actual copying.

We also experimented with parallelizing the file system traversal itself however it seems that this would not significantly improve the performance as the traversal is typically done faster than the copying.

4.3 Docker Injection

Once a repository subset has been exported to a POSIX directory we can - among other formats - inject the software into a Docker image. The Docker injection further allows to later update the software inside the image to a newer revision by replacing the corresponding files in the docker image.

For the first Docker injection a new tarball layer is added to the Docker container and the tar and gz hashes of the layer are stored inside the image as labels. For subsequent updates the utility will then identify the cvmfs layer through the image labels, retrieve the layer through an OCI registry and replace the ”old” tarball in the image by a new, updated, version.

With the upcoming CVMFS 2.6 release which also features a notification system for updates of a CVMFS repository, the described Docker image update functionality could allow for a continuous deployment architecture in which Docker images are automatically updated once there is a new software revision available.

5. Use Cases

5.1 ATLAS at NERSC

The US collaboration of the ATLAS project is currently taking advantage of computing resources at various super computers in the United States including Cori at NERSC. For the deployment of their software the uncvmfs utility is used to unpack entire CVMFS repositories into standalone images (usually in ext4 or squashfs format) which in turn can be used to build Shifter [10] or Singularity [11] images. While these images can very well be scaled out onto a large number of nodes inside the NERSC infrastructure, the image build process takes around 24 hours which makes it difficult to deploy up-to-date versions of the software on a regular basis [10, p. 3].

We are currently discussing options to integrate our shrinkwrap utility into the HPC deployment processes of ATLAS as we believe that the proposed software solution can help to reduce the overhead of HPC software delivery significantly.
On the one hand the actual uncvmfs export process can be optimized as the current export workflow loads the entire CVMFS repository in a first step and later on removes certain files in the exported image which were not in fact needed. This is something that can be entirely avoided through the definition of exclusion rules passed to the shrinkwrap utility.

On the other hand, we believe that an export client which relies on the libcvmfs library might reduce the total overhead and therefore improve the export performance in comparison to both uncvmfs and rsync (for uncvmfs this is something that we haven't had yet the chance to evaluate in an experiment).
As a longer-term goal, it might be useful to start tracing ATLAS simulation workflows on a few machines to collect data that might allow further export optimizations in a later stage through automated specification building.

5.2 Benchmarking Containers

For benchmarking purposes, the CERN IT-Department is interested in a "Benchmark in a Container" solution which allows the distribution of standalone container images for system performance evaluations. For such benchmarks the IT-Department typically relies on simulation workflows provided by CERN's experiment collaborations. To execute these benchmarks in an easily portable and reproducible manner, it is of high interest to package these software workflows into containers. Since the workflows' software stacks are usually stored inside CVMFS repositories, this is a perfect use case for the shrinkwrap utility and it's Docker injection functionality.

For the image export the workflow's resource-needs would be examined through the CVMFS tracing feature in a first step. This would then result in a log containing detailed insights on the paths which were accessed by the benchmarking workload. In a second step we can then produce a specification based on the tracing log which will allow an easy, efficient export of the necessary files from CVMFS. Afterwards the Docker injector can be used to inject the exported files into a prepared benchmarking container stored in a registry. Later, the same utility can be used to update the current container to an up-to-date software version or - depending on the exact use case - the utility could even be integrated into a continuous deployment mechanism which produces new benchmarking containers on every software update.

6. Performance evaluation

To evaluate the performance of our system we executed some preliminary benchmarks to check on its performance for the export of a sample directory. This is because experiments with real use cases are still due.

Setup For the benchmarks a dedicated machine with 16 cores, 64GB of RAM and a Samsung Evo SSD RAID5 storage system was used. During the benchmark execution no other tasks ran on the machine. Each benchmark evaluation consisted of 16 identical runs with a preloaded warm cache. More information for reproducibility can be found in appendix A.

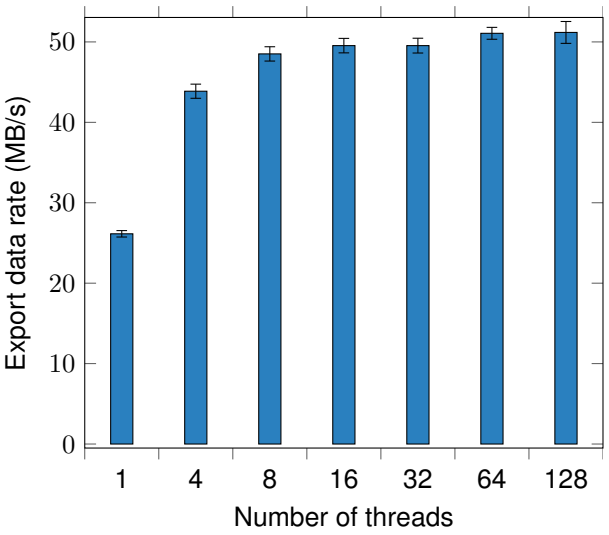

Figure 6.2: Data rate with standard deviation of export through shrinkwrap utility with various thread numbers

Evaluated example For the benchmark a subset of the cernvm-prod.cern.ch repository was exported into a POSIX folder. The subset consisted of 405,904 entries of which 358,488 were files (including 321,683 unique files) while the other entries were directories or symbolic links. The entire subset amounted to an exported image size of 13.39 GB (8.16 GB when deduplicated). A histogram of the file sizes (including duplicates) can be found in figure 6.1.

Experimental results on cvmfs_shrinkwrap The first test run pictured in Figure 6.2 indicates that our software's multithreading capability helps significantly in speeding up the export process by keeping the disk busy at all times. It is visible however that the export rate saturates at around 50MB/s.

The original version tested used a 4kB copy buffer per thread. One method that should theoretically increase the export speed would be to enlarge the copy buffer as this would decrease the number of copy iterations necessary per file. To check this the export rates with a 4kB buffer were compared to the export rates with a 1MB buffer as seen in figure 6.3. However, it is quite clear that the additional buffer space did not increase the export rate by a significant margin. The large standard deviation for the 32-thread case is not yet understood and might very well be just due to an anomaly during testing. Due to insufficient improvements given a larger buffer we discarded the buffer size increase.

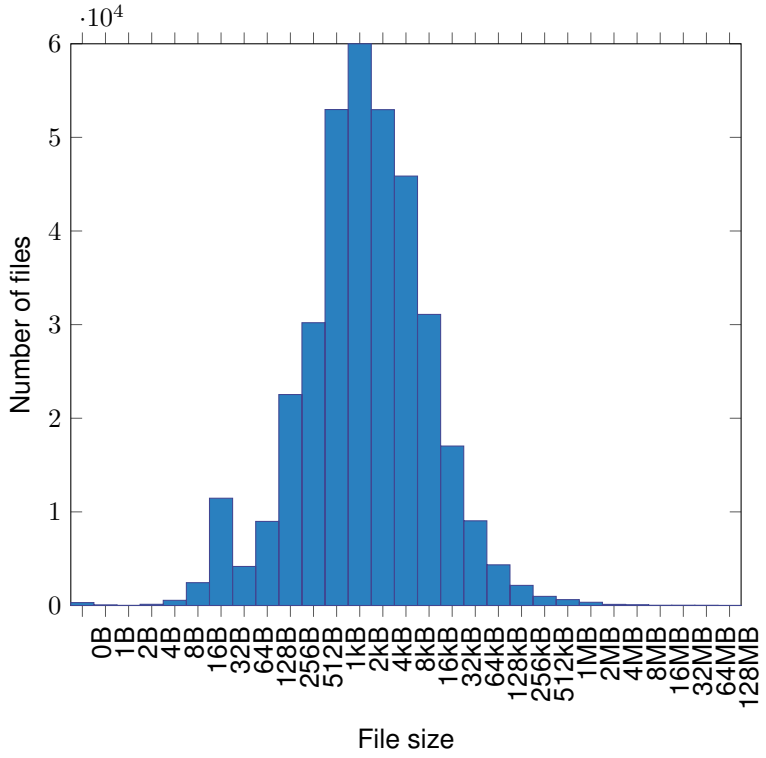

Figure 6.1: File size histogram for the exported image

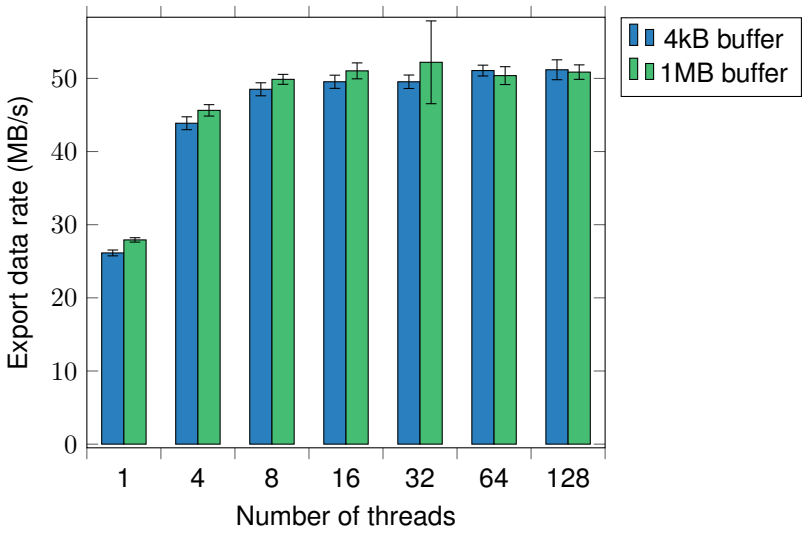

Figure 6.3: Data rate with standard deviation of export through shrinkwrap utility with various thread numbers comparing different buffer sizes.

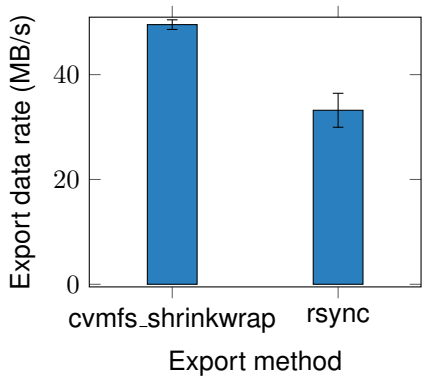

Figure 6.4: Data rate with standard deviations for export through shrinkwrap (32 threads) in comparison with rsync

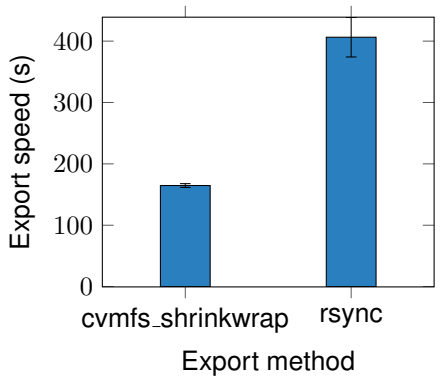

Figure 6.5: Export time with standard deviation for a 13.39GB image using shrinkwrap (32 threads) in comparison with rsync. (Shrinkwrap made use of deduplication)

Comparison with rsync One possible alternative to using the proposed shrinkwrap utility would be the use of rsync for exporting from the FUSE mounted CVMFS repository. As seen in figure 6.4 our experiments showed that an export with cvmfs_shrinkwrap is around 20MB/s faster than an export with rsync. This can be explained by both the use of multithreading in the shrinkwrap case and the avoidance of FUSE-system calls which are necessary for the rsync case and can be avoided by the shrinkwrap utility using libcvmfs. A look at the absolute export time presented in figure 6.5 shows that the total time of a shrinkwrap export is even faster than expected given the data rates due to the use of deduplication which avoided duplicate transfers of some 5GB of data in this case.

Future evaluation For future evaluations it would be interesting to compare the export speed of uncvmfs to cvmfs_shrinkwrap. Further benchmarks with real use cases from ATLAS or the IT-Department could provide more insights into the performance of our utility. Unfortunately, we haven't had the chance to test these things before the end of the internship due to time constraints.

7. Conclusions

7.1 A software solution for shrinkwrapping

In this report we presented software utilities allowing the requirements analysis, specification and production of standalone images from CVMFS. We presented solutions for various problems encountered underway of the development including a methodology for fast(er) specification matching as well as a method for integrating shrinkwrapped images into Docker containers.

We showed that our software's performance exceeds the performance of utilities like rsync both in terms of time (faster) and image size (smaller through deduplication).

The utility is currently under code-review and is planned be released with CVMFS version 2.6 coming end of this year.

7.2 Future work

While generally in good shape there are still a few points which could be improved and fine-tuned.

Specification language extension While it is generally preferable to have a detailed specification containing all files and directories that should be exported, certain cases make a wildcard directory inclusion necessary. In this case those directories might however still contain certain file types which the user desires not to export. An example is the current image build system used by ATLAS, which includes some directories while specifying that certain file extensions should not be exported into the image (e.g. HTML documentation files). For such cases a (preferably short) list of exclusion regular expressions in the specification language would be desirable. This is something that should be added to the utility in order to allow the experiments to efficiently use the software.

Native image export to other formats Currently the only supported export format from CVMFS is a POSIX directory. While this already allows to export into any format which can be (loopback) mounted in write mode on a POSIX system, it could be of big impact to develop export functionalities into other formats such as SquashFS or tar. While a one-time export would be quite easy to achieve this task becomes challenging for the update case where files might be deleted and would therefore have to be removed from such a continuous- segment-based file format (in particular for the tarball case). As the export into a tarball for example, currently consists of unpacking a tar into a temporary directory, updating the corresponding POSIX directory and then repacking the tar, considerable performance gains could be expected from a native tar export functionality. In a case where SquashFS or tarball images become an essential use case of the utility it might be worthwhile to investigate methods allowing for a native export into such formats.

Specification difference reasoning It was already mentioned that it is preferable to use multiple traced software runs (possibly with varying software parameters) in order to obtain stable, reliable specifications. The easiest methodology for this would be to append all trace logs into one single, large trace log and to use this file for the automated specification building.

Another possible methodology would be to first produce one specification per software run and then to intelligently merge these export specifications into a single overall specification. For this one would essentially perform automated, rule-based reasoning about a software's behavior based on the differences observed during varying software executions. In the described case results could be achieved by interpreting the different image specifications under the assumption that a software might branch into different children of some arbitrary directory in the repository tree during different software runs. In such a case it might be more reasonable to include the entire parent directory to obtain a more stable (but in turn slightly larger) image. This would be visible in the specification files through the inclusion of varying children of some directory inside the tree in the runs' different specifications.

Some preliminary tests with this methodology on a ROOT test case showed promising results given some (at this stage) manual tuning and might therefore be worth to pursue.

Optimization of the commandline interface As we are still evaluating the exact workflows for various use cases we have not yet defined the final command line interface of the software utility. In the long-term it would be highly preferable to have one single entry point for the software utility allowing the export into all the different formats possibly calling various subsystems or external functionalities to achieve this.
This optimization is something that should be developed before the first release in CVMFS 2.6 and will be designed in frequent discussions with parties interested in some of the use cases described above to optimize the utility for their workflows. Generally, we believe that a dual interface would be quite worthwhile for this either allowing configuration of the export through pure commandline arguments or allowing the export configuration to be stored inside some configuration file.

Bibliography

[1] Katie Antypas, Nicholas Wright, Nicholas P Cardo, Allison Andrews, and Matthew Cordery. Cori: a cray xc pre-exascale system for nersc. *Cray User Group Proceedings. Cray*, 2014. 1

[2] I Bird, U Schwickerath, R Jones, J Shiers, N Brook, C Grandi, Christoph Eck, I Fisk, Y Schutz, B Panzer-Steindel, et al. Lhc computing grid: Technical design report. Technical report, CERN, 2005. 1

[3] J. Blomer, P. Buncic, R. Meusel, G. Ganis, I. Sfiligoi, and D. Thain. The evolution of global scale filesystems for scientific software distribution. *Computing in Science Engineering*, 17(6):61–71, Nov 2015. 1

[4] Jakob Blomer, Carlos Aguado-Sánchez, Predrag Buncic, and Artem Harutyunyan. Distributing lhc application software and conditions databases using the cernvm file system. *Journal of Physics: Conference Series*, 331(4):042003, 2011. 1, 5

[5] Jakob Blomer, Gerardo Ganis, Nikola Hardi, and Radu Popescu. Delivering lhc software to hpc compute elements with cernvm-fs. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing*, pages 724–730, Cham, 2017. Springer International Publishing. ii, 1, 2

[6] CERN. Open container initiative distribution specification. `https://github.com/opencontainers/distribution-spec`. [online; accessed 14 August 2018]. 1

[7] CernVM. Cernvm file system. `http://cernvm.cern.ch/portal/filesystem`. [online; accessed 11 June 2018]. ii

[8] CernVM. Preloading the cernvm-fs cache. `https://cvmfs.readthedocs.io/en/latest/cpt-hpc.html?highlight=dirtab#preloading-the-cernvm-fs-cache`. [online; accessed 15 August 2018]. 6

[9] FUSE. Filesystem in userspace. `https://github.com/libfuse/libfuse`. [online; accessed 27 August 2018]. 1

[10] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. Shifter: Containers for hpc. In *Journal of Physics: Conference Series*, volume 898, page 082021. IOP Publishing, 2017. 8

[11] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017. 8

[12] Tim Shaffer, Jakob Blomer, and Gerardo Ganis. Hep application delivery on hpc resources. `https://doi.org/10.5281/zenodo.61157`, August 2016. [online; accessed 11 June 2018]. 1, 2

[13] Douglas Thain and Miron Livny. Parrot: An application environment for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005. 2

[14] uncvmfs. uncvmfs. https://github.com/ic-hep/uncvmfs. [online; accessed 14 August 2018]. 1, 2, 5

[15] Wei Yang. Containers and cvmfs for nersc and other hpc centers. https://twiki.cern.ch/twiki/bin/view/AtlasComputing/ContainersAtNERSC. [online; accessed 11 June 2018]. ii, 2

A. Reproducibility

All execution protocols from the benchmarks can be found in a git repository available at `https://github.com/samysweb/cvmfs_shrinkwrap-bench0818`.
For each benchmark there exist 2 log files: One logfile (called run...) containing details on the execution with outputs provided by the benchmarked software and one log file (called bench...) containing the benchmark results summaries as well as the bash scripts and configurations used for execution.
In order to obtain consistent benchmark results, all file system caches were emptied before each software run. After each execution the sync command was executed to account for files that might still be stored in memory caches.