EPJ Web of Conferences **214**, 05005 (2019) https://doi.org/10.1051/epjconf/201921405005
CHEP 2018

Software packaging and distribution for LHCb using Nix

Chris Burr[1,*], *Marco* Clemencic[2], and *Ben* Couturier[2]
on behalf of the LHCb Collaboration

[1]The University of Manchester
[2]CERN

Abstract. Software is an essential and rapidly evolving component of modern high energy physics research. The ability to be agile and take advantage of new and updated packages from the wider data science community is allowing physicists to efficiently utilise the data available to them. However, these packages often introduce complex dependency chains and evolve rapidly introducing specific, and sometimes conflicting, version requirements which can make managing environments challenging. Additionally, there is a need to replicate old environments when generating simulated data and to utilise pre-existing datasets. Nix is a "purely functional package manager" which allows for software to be built and distributed with fully specified dependencies, making packages independent from those available on the host. Builds are reproducible and multiple versions/configurations of each package can coexist with the build configuration of each perfectly preserved. Here we will give an overview of Nix followed by the work that has been done to use Nix in LHCb and the advantages and challenges that this brings.

1 Introduction

Computationally intensive areas of modern research, such as high energy physics, provide unique challenges for software packaging. Software is used at a massive scale for processing large datasets using heterogeneous resources, such as the Woldwide LHC Computing Grid [1]. Simulating and reprocessing data can continue to use software for decades after the software was originally written. For example, the Large Electron Positron collider (LEP) continued to publish results for over 20 years after the start of data taking and the Large Hadron Collider (LHC) will have an even longer lifetime.

In order to facilitate this use, software must be stable for long periods; much longer than even Long Term Support operating systems are available. Additionally, the software should reproduce any and all bugs which were present in the original version to ensure the accuracy of the final results. Builds should be reproducible to allow for patches to be carefully introduced.

Contradictorily, analysts of data often want to experiment with using modern or even prerelease software to make analysing data easier or to improve final results. However, once a method has been finalised, the environment is expected to stay stable for the remainder of the analysis which often takes multiple years. Even after a result is published, it can still be

---

[*] e-mail: christopher.burr@cern.ch

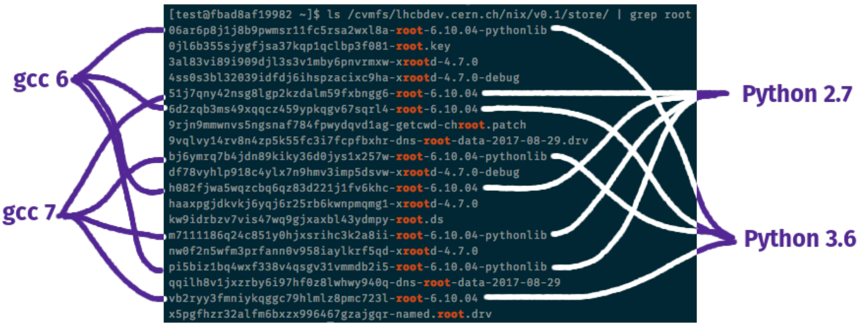
necessary to rerun the code to combine older results with newer ones and to ensure the best possible combined result is obtained.

Finally, most analysts are physicists with little training in computer science best practices and should not be expected to build and preserve complex software stacks.

2 Nix

Nix [2] is a “purely functional package manager” that was started as a research project in 2003 [3]. It has since grown to become both a full Linux based operating system (NixOS) as well as an independent package manager that supports both Linux and macOS. It can build and run software for the `i686`, `x86_64` and `arm64` architectures, either directly or with the use of cross-compilation. Nix is used for a wide range of use cases including managed hosting [4], high performance computing (HPC) [5, 6], financial services companies and embedded systems [7].

A strong focus of Nix is on the purity, reproducible and portability of the builds. Packages are built as deep stacks, with every dependency being defined within Nix down to the `libc` and `ELF` interpreter. Installed packages are kept in a unique subdirectory of the *store*, typically `/nix/store/`. This subdirectory is named using a cryptographically secure hash of all inputs to the build, including the build sources, configuration and dependencies to allow for an unlimited number of versions and configurations to be available simultaneously, without any risk of conflicts between installations. For example if ROOT and XRootD are each built with different Python and `gcc` versions they each end up in a different directory as shown in Figure 1.



```
[test@fbad8af19982 ~]$ ls /cvmfs/lhcbdev.cern.ch/nix/v0.1/store/ | grep root
06ar6p8j1j8b9pwmsr11fc5rsa2wxl8a-root-6.10.04-pythonlib
0jl6b355sjygfjsa37kqp1qclbp3f081-root.key
3al83vi89i909djl3s3v1mby6pnvrmxw-xrootd-4.7.0
4ss0s3bl32039idfdj6ihspzacixc9ha-xrootd-4.7.0-debug
51j7qny42nsg8lgp2kzdalm59fxbngg6-root-6.10.04
6d2zqb3ms49xqqcz459ypkqgv67sqrl4-root-6.10.04
9rjn9mmwnvs5ngsnaf784fpwydqvd1ag-getcwd-chroot.patch
9vqlvy14rv8n4zp5k55fc3i7fcpfbxhr-dns-root-data-2017-08-29.drv
bj6ymrq7b4jdn89kiky36d0jys1x257w-root-6.10.04-pythonlib
df78vyhlp918c4ylx7n9hmv3imp5dsvw-xrootd-4.7.0-debug
h082fjwa5wqzcbq6qz83d221j1fv6khc-root-6.10.04
haaxpgjdkvkj6yqj6r25rb6kwnpmqmg1-xrootd-4.7.0
kw9idrbzv7vis47wq9gjxaxbl43ydmpy-root.ds
m7111186q24c851y0hjxsrihc3k2a8ii-root-6.10.04-pythonlib
nw0f2n5wfm3prfann0v958iaylkrf5qd-xrootd-4.7.0
pi5biz1bq4wxf338v4qsgv31vmmdb2i5-root-6.10.04-pythonlib
qqilh8v1jxzrby6i97hf0z8lwhwy940q-dns-root-data-2017-08-29
vb2ryy3fmniykqggc79hlmlz8pmc723l-root-6.10.04
x5pgfhzr32alfm6bxzx996467gzajgqr-named.root.drv
```

Figure 1: Example contents of a Nix store directory containing ROOT and XRootD built using gcc 6 and gcc 7 against Python 2.7 and Python 3.6 in a build matrix. This results in 4 unique builds of ROOT/XRootD each with a different dependency hash in their installation directory.

Source files, such as tarballs and patches, are defined using a hash of their content. These are downloaded and imported into the store directory to ensure that all required inputs are available indefinitely, or until they are explicitly deleted.

To ensure builds remain pure and do not have dependencies which have not been explicitly specified, Nix typically uses a sandbox [8] to isolate builds. This uses modern Linux kernel features, such as user namespaces, to restrict the build to only accesses the directories within the Nix store directory which have been specified as a dependency. Additionally, downloaded inputs such as source tarballs and patches are downloaded to the store directory and network access is restricted to prevent builds from downloading files which may change or be removed in future. Builds aim to be bit-for-bit reproducible, though this is an ongoing effort with the wider community to remove non-deterministic elements from the build process [9].

The primary source of nix expressions is the `nixpkgs` git repository [10], which contains definitions for $O(14\,000)$ packages. The git commit hash of a particular revision can be used to pin a snapshot of this repository. Simple modifications can be made when installing by overriding attributes on the deviation which is to be installed. Additionally, “overlays” can be used to modify any part of `nixpkgs`; from a minor configuration change when building, to replacing a low level dependency such as the `gcc` version and flags used for building all known packages which triggers a rebuild of the entire system.

3 Hydra

One of the disadvantages of building deep stacks is that it is time consuming and inconvenient for many use cases. To mitigate this issue Nix can query static web servers using the package’s hash to download a signed tarball of the build products. The servers hosting this content are known as binary caches.

Binary caches can be managed using Hydra [11], a continuous build system which can be used to build software after every change, after releases or periodically. It has deep integration with Nix and is primarily built for the testing and deployment of the official Nix binary cache, though it can also be used to provide build and continuous integration for any project. Private instances are used by several organisations that build the entirety of `nixpkgs` either to apply low level customisations, such as changing the default compiler, or out of security concerns when using externally provided binaries. Hydra is also used to provide continuous integration for Nix projects such as Nix, Hydra, patchelf [12] as well as some GNU projects.

Hydra can either be ran using a single machine or use SSH to distribute builds over a cluster of build machines and has mitigations built to fix common issues, such as misbehaving workers, network issues or random failures. A web interface is provided for configuring Hydra, managing builds and viewing build logs. Binaries can be served directly, or uploaded using a plugin system (most commonly to a `S3` compatible endpoint).

4 Defining packages

Nix packages are defined using a custom functional language though knowledge of this language is not needed for almost any use cases. The (`nixpkgs`) repository contains many helper functions to simplify defining packages, while also performing actions to help ensure the builds are pure. Package definitions already exist with support for most build systems as well as binary releases. Adding new packages can generally be done by creating a file containing an URL and hash for the source, listing the packages’s dependencies and then adding one line to `pkgs/all-packages.nix` to make Nix aware of the new package. The default build script hides almost all of the complexity of correctly building packages for Nix. It is highly configurable and splits the build into phases:

- **`unpackPhase`**: Unpack the archives from the `src` variable.
- **`patchPhase`**: Apply any patches that are required, taken from the `patches` variable.
- **`configurePhase`**: Prepare the source tree for building. By default this assumes an Autoconf script and runs `./configure.sh` however including dependencies like `cmake` overrides this as appropriate.

- `buildPhase` Compile the package, by default this simply calls `make` provided a suitable `Makefile` is present.
- `checkPhase`: Run tests against the build output to avoid broken builds. Defaults to being disabled.
- `installPhase`: Install software to the default store directory, typically by running `make install`.
- `installCheckPhase`: Similar to `checkPhase` except test against the installed binaries. Also disabled by default.
- `fixupPhase`: Perform Nix specific post-processing. This involves stripping or splitting debug information, patching interpreter paths, minimising runtime dependencies by simplifying the `RPATH` in `ELF` files and splitting the output into multiple parts. Much of this is achieved using patchelf which is also a Nix project.

An example derivation which is used to build the base `LHCb` software package is shown in Figure 2.

```nix
{ stdenv, fetchurl, boost, cmake, python, ninja, root, gaudi
, clhep, xercesc, cppunit, libxml2, openssl, relax, gsl, eigen, aida, graphviz
, qt5, mysql57, sqlite, hepmc, cool, coral, libgit2, pkgconfig, vdt, cpp-gsl
, oracle-instant-client, xrootd
# Data packages
, det-sqldddb, fieldmap, gen-decfiles, paramfiles, prconfig, raweventformat
, tck-hlttck, tck-l0tck }:

stdenv.mkDerivation rec {
  name = "LHCb-${version}";
  version = "v44r0";

  src = fetchurl {
    url = "https://gitlab.cern.ch/lhcb/LHCb/repository/${version}/archive.tar.gz";
    sha256 = "0h5wph3p3ha7h34byyamd1dlvb27hs5xpjbfff363y8r43dsk4pa";
  };

  buildInputs = [
    cmake ninja boost gaudi clhep xercesc cppunit libxml2 openssl relax eigen
    gsl aida graphviz qt5.qtbase mysql57 sqlite hepmc cool coral libgit2
    pkgconfig vdt cpp-gsl oracle-instant-client xrootd root
    (python.withPackages (ps: with ps; [ xenv pyqt5 lxml ]))
    det-sqldddb fieldmap gen-decfiles paramfiles prconfig
    raweventformat tck-hlttck tck-l0tck
  ];

  propagatedBuildInputs = [ python ];

  cmakeFlags = [
    "-GNinja"
    "-DMYSQL_INCLUDE_DIR=${mysql57}/include/"
    "-DGRAPHVIZ_INCLUDE_DIR=${graphviz}/include/"
    "-DCOOL_PYTHON_PATH=${cool}/python"
    "-DCORAL_PYTHON_PATH=${coral}/python"
  ];

  checkPhase = ''
    ninja test
  '';
  doCheck = true;

  postInstall = ''
    for fn in $out/lib/lib*.so; do \
      ${gaudi}/bin/listcomponents.exe $fn >> "'${fn%.so}.components"
    done
  '';

  enableParallelBuilding = true;

  meta = {
    homepage = http://lhcbdoc.web.cern.ch/lhcbdoc/lhcb/;
    description = "General purpose classes used throughout the LHCb software.";
    platforms = stdenv.lib.platforms.unix;
  };
}
```

Figure 2: Nix expression of defining `LHCb` the base library of the LHCb experiment's software stack.

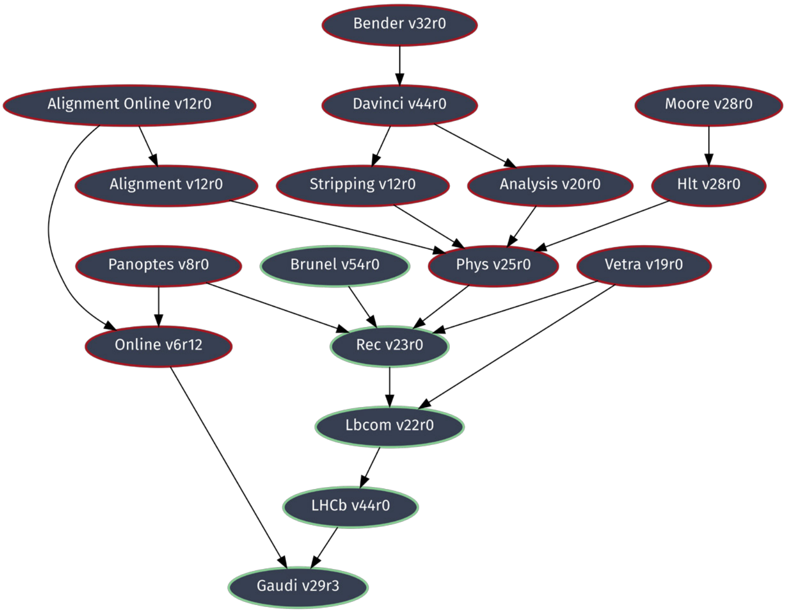


Figure 3: Dependency graph containing most of LHCb's software packages. The packages built as part of this work are shown in green.

5 Defining environments

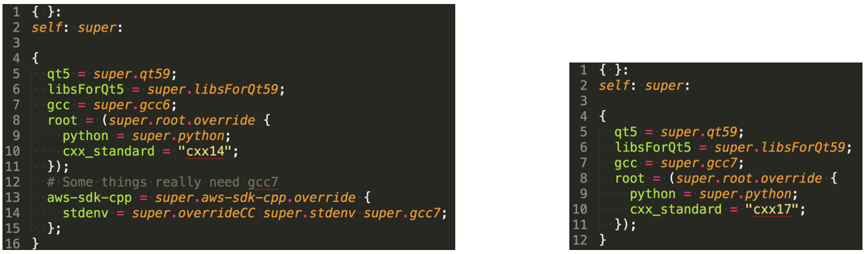
Environments can be defined using Nix as meta packages which are "built" by creating a directory of symlinks. nixpkgs contains several helpful functions to help with this, the most important of which is `buildEnv`. Nix includes an executable (`nix-shell`) which can be used to setup environments, including non standard environment variables such as `ROOTSYS` and `CMAKE_MODULE_PATH`. See the HSF packaging group's testdrives for an example of using `buildEnv` to define a deep stack [13].

6 Tests building LHCb software

The LHCb software stack, shown in Figure 3, is made up of around 20 separate packages which are typically distributed as binary releases on CVMFS [14]. For testing Nix it was decided to build up to the reconstruction package (Brunel) which depends on 4 other LHCb packages, several "data packages" which contain non-executable dependencies like the magnetic field map, as well as many external packages.

To ensure Nix is suitable for use with the current distribution model, the store directory was changed to a mocked directory representing CVMFS, `/cvmfs/lhcbdev.cern.ch/nix/`. This can be done by setting environment variables which override the install directory. As this contributes to the hash which is used to define a package this results in all packages having to be rebuilt.

Initial developments relied upon building all software from source at install time, however it was soon found that setting up a custom Hydra instance to serve a binary cache is simple and dramatically improves the experience of using Nix. This instance is hosted on CERN's OpenStack cloud and is backed with a Postgres DataBase on Demand (DBoD) instance. Hydra is installed inside a minimal Docker container running Alpine Linux and uses



```
{ }:
self: super:

{
  qt5 = super.qt59;
  libsForQt5 = super.libsForQt59;
  gcc = super.gcc6;
  root = (super.root.override {
    python = super.python;
    cxx_standard = "cxx14";
  });
  # Some things really need gcc7
  aws-sdk-cpp = super.aws-sdk-cpp.override {
    stdenv = super.overrideCC super.stdenv super.gcc7;
  };
}
```

(a) `gcc6.nix`

```
{ }:
self: super:

{
  qt5 = super.qt59;
  libsForQt5 = super.libsForQt59;
  gcc = super.gcc7;
  root = (super.root.override {
    python = super.python;
    cxx_standard = "cxx17";
  });
}
```

(b) `gcc7.nix`

Figure 4: Example overlay definition files used to change the default qt and compiler version as well as the c++ standard. For `aws-sdk-cpp` it was necessary to override the compiler back to `gcc7` as `gcc6` is not supported.

SSH to connect to a Docker container running on a powerful build machine. Additional build machines were easy to add at times of high load.

The unstable branch of the upstream `nixpkgs` repository was forked to allow easy experimentation with building entirely custom stacks on top of Nix, such as rebuilding all packages under different `gcc` versions. Maintaining this fork was simple, with Hydra automatically monitoring for changes and making new builds as appropriate. An even simpler method was later found known as pinning `nixpkgs` which allows a upstream git revision to be specified along with a list of `patch` files.

While most dependencies of the LHCb software are already included in Nix; `CatBoost`, `COOL`, `CORAL`, `CLHEP`, `frontier`, `pacparser`, `RELAX`, `REFLEX` and `VDT` were missing. Most were trivial to define with only two requiring notable effort:

- **`CatBoost`** has a closed source build system which depends on glibc, once this was identified it was simple to fix using `patchelf`.
- **Oracle Instant Client** is included in `nixpkgs` however licensing issues prevent Nix from automatically downloading and distributing the source binaries. This required manually downloading/importing the source and enabling non-free builds in Hydra.

LHCb's software is typically built for a selection of platforms which are defined according to the HSF platform naming convention [15]. This defines a string of the form `architecture-OS-compiler-buildtype` such as `x86_64+avx2-centos7-gcc7-opt` and `x86_64-slc6-gcc49-dbg`. A similar system was achieved within Nix with the use of *overlays*, with exception of the `OS` component which is redundant when using Nix as binaries can be used on any Linux distribution. This allows for modifications to `nixpkgs` to be defined in an external file for purposes such as replacing the default `gcc` version. Examples are shown in Figure 4.

7 Containers

Container technologies, such as Docker [16] and Singularity [17], are seen as a likely solution to many software preservation problems as they provide a simple way to ensure self contained and system independent binaries. Despite this, the problem of how to build a container remains. Typically the build process is effectively a shell script and often downloads dependencies using a package manager with no guarantee that the same script will continue to work indefinitely. Nixpkgs provides a solution to this in the form of functions that can build images according to `v1.2.0` of the Docker Image Specification [18] from a Nix expression. This ensures the reproducibility of the build as the configuration and source dependencies will have been fully cached in the Nix store of the build machine.

Additionally, containers are often relatively large binary blobs which add overhead when starting jobs and significantly increase the amount of storage required. This situation can be improved by using layers in the container to share a common basis between containers. This basis is not ideal however as each layer in the image is dependent on the previous layer leading to duplication between layers.

Further improvements are possible with Nix thanks to the fact that each directory within the Nix store is immutable after installation and has an exactly known set of dependencies which are also store directories. This, combined with the fact the dependencies are defined by images rather than the layers themselves means that each store directory can be placed into a separate layer. Docker images can then be created that depend on arbitrary combinations of these layers to give maximal caching between images using the `pkgs.dockerTools.buildLayeredImage` function from `nixpkgs`. A more advanced algorithm can be used to work around the limit on the number of layers that can be used by an image [19].

8 Summary and future work

High energy physics, and HPC in general, requires highly configurable package management which is able to produce efficient and reproducible binaries. Nix is an ideal candidate for this task and there is interest from the wider HPC community in Nix, with several organisations working to improve relevant parts of Nix such as support for InfiniBand networking, the Intel Math Kernel Library and the Intel Compiler Collection. As software continues to become more and more complex shared effort is becoming essential to ensure builds remain up to date and reliable. Especially as many popular ecosystems, such as Python, contain many small packages which can be difficult to distribute in a reliable way. Nix has been used to successfully build part of the LHCb experiment's software stack and this effort with continue, with changes being pushed upstream in collaboration with the wider community [20].

References

[1] I. Bird, P. Buncic, F. Carminati, M. Cattaneo, P. Clarke, I. Fisk, M. Girone, J. Harvey, B. Kersevan, P. Mato et al. (2014), `CERN-LHCC-2014-014 LCG-TDR-002`, `https://cds.cern.ch/record/1695401`

[2] E. Dolstra, *Nix: The purely functional package manager*, accessed on: 01/12/2018, `https://nixos.org/nix/`

[3] E. Dolstra, Ph.D. thesis, Universiteit Utrecht (2006), `https://nixos.org/~eelco/pubs/phd-thesis.pdf`

[4] C. Kauhaus, *Migrating a Hosting Infrastructure from Gentoo*, in *Nixcon* (2018), `https://www.youtube.com/watch?v=5GtOAaqqNGU`

[5] P.A. Bouttier, *Nix as HPC package management system*, in *Nixcon* (2018), `https://www.youtube.com/watch?v=s5iY3CsdSfQ`

[6] B. Oldeman, *Combining CVMFS, Nix, Lmod, and EasyBuild at Compute Canada*, in *FOSDEM* (2018), `https://archive.fosdem.org/2018/schedule/event/computecanada/`

[7] D. Barlow, *NixWRT: purely functional firmware images for IoT*, in *Nixcon* (2018), `https://www.youtube.com/watch?v=0K1qn60X2HI`

[8] J. Thalheim, *About Nix sandboxes and breakpoints*, in *Nixcon* (2018), `https://www.youtube.com/watch?v=ULqoCjANK-I`

[9] H. Levsen, C. Lamb, *Reproducible Buster and beyond*, in *DebConf18* (2018), `https://debconf18.debconf.org/talks/80-reproducible-buster-and-beyond/`

[10] E. Dolstra, the Nixpkgs/NixOS contributors, accessed on: 01/12/2018, `https://github.com/NixOS/nixpkgs`

[11] E. Dolstra, E. Visser, *Hydra: A declarative approach to continuous integration* (2018), `https://nixos.org/~eelco/pubs/hydra-scp-submitted.pdf`

[12] E. Dolstra, *Patchelf*, accessed on: 08/04/2019, `https://nixos.org/patchelf.html`

[13] HEP Software Foundation, accessed on: 01/12/2018, `https://github.com/HSF/packaging`

[14] J. Blomer, Ph.D. thesis, Technical University of Munich (2012), `https://cdsweb.cern.ch/record/1462821/`

[15] B. Hegner, Tech. Rep. HSF-TN-2018-01, HEP Software Foundation (2018), `https://hepsoftwarefoundation.org/notes/HSF-TN-2018-01.pdf`

[16] Docker, Inc., accessed on: 01/12/2018, `https://www.docker.com/`

[17] G.M. Kurtzer, V. Sochat, M.W. Bauer, PLOS ONE **12**, 1 (2017)

[18] Docker, Inc., accessed on: 01/12/2018, `https://github.com/moby/moby/blob/master/image/spec/v1.2.md#docker-image-specification-v120`

[19] G. Christensen, accessed on: 01/12/2018, `https://grahamc.com/blog/nix-and-layered-docker-images`

[20] C. Burr, *Nix for software deployment in high energy physics*, in *Nixcon* (2018), `https://www.youtube.com/watch?v=Ee8k97Rx3DA`