Who am I?

- PhD student at Manchester working on LHCb
- Supposed to be working on analysis and velo alignment
- Generally interested in computing
- Before starting this work I tried a few things
  - Including packaging ROOT/XRootD with conda
Why this started?

• This started with analysis preservation in mind
  • Post-DaVinci environments can be tricky to share/preserve
  • Docker is great, but can’t be used in most places
  • Must be something better
• Looked at various options, settled on Nix
• Nix could be more generally useful everywhere
What is Nix?
What is Nix?

• Nix is a “purely functional package manager”
  • Works with Linux and macOS
  • Can be used alongside other package managers
  • There is also a Linux distribution, NixOS
What is Nix?

- Nix is a “purely functional package manager”
- Source-based
  - *Binary caches* can be used to avoid compiling everything
What is Nix?

• Nix is a “purely functional package manager”
• Source-based
• Packages are built from Nix expressions
  • Typically $O(10)$ lines long
  • Defined using a custom functional language
  • $\sim$14,000 package definitions available in nixpkgs
What is Nix?

- Nix is a “purely functional package manager”
- Source-based
- Packages are built from *Nix expressions*
- Builds aim to be portable, reproducible and deterministic
What is Nix?

- Nix is a “purely functional package manager”
- Source-based
- Packages are built from *Nix expressions*
- Builds aim to be portable, reproducible and deterministic
- Lots more features available when used fully
  - NixOS
  - Single and multi user modes
  - Transactional approach to updates and configuration
How does Nix work?

- Everything is stored in /nix/ (by default)
- Packages are kept in /nix/store
- Each package lives in a directory named by hash of it’s dependencies
  - gcc6: /nix/store/6d2zqb3ms49xqqcz459ypkqgv67sqr14-root-6.10.04/
  - gcc7: /nix/store/h082fjwa5wqzcbq6qz83d221j1fv6khc-root-6.10.04/
-Optionally packages can have multiple outputs
  - bin, lib, python-lib, ...
Nix expressions

• A collection of nix expressions is known as a *channel*
• Nixpkgs is the most common: https://github.com/NixOS/nixpkgs

There are also release channels at: https://github.com/NixOS/nixpkgs-channels
A collection of nix expressions is known as a channel

Nixpkgs is the most common: https://github.com/NixOS/nixpkgs

Nixpkgs also provides helper functions
  - buildEnv: Makes a meta package of symlinks
  - `fetchurl`/`fetchgit`/`fetchpatch`/`fetchcvs`/`fetchipfs`
  - `stdenv.mkDerivation`
    - Uses the standard environment to run a genericBuild
    - Sets up linker flags and `RUNPATH`
    - Rewrites the interpreter paths of shell scripts to `/nix/store/...`
    - Also uses test suites for many packages

1There are also release channels at: https://github.com/NixOS/nixpkgs-channels
A Nix expression for Gaudi

```nix
{ stdenv, fetchurl, fetchpatch, boost, clhep, cmake, cppunit, gperftools
, heppdt, jemalloc, libunwind, python, tbb, utillinux, xercesc, zlib
, ninja, root, gdb, aida, gsl, libpng }

stdenv.mkDerivation rec {
  name = "gaudi-${version}";
  version = "v29r0";

  src = fetchurl {
    url = "https://gitlab.cern.ch/gaudi/Gaudi/repository/${version}/archive.tar.gz";
    sha256 = "1ijdq1l8rscwij9hgyzrlvgalqg7b0csx76wcd76x3yli8bc766b";
  };

  buildInputs = [
    cmake python gdb aida ninja root boost clhep cppunit gperftools heppdt
    jemalloc libunwind tbb utillinux xercesc zlib gsl libpng
  ];

  patches = [ ./fix-profiling.patch ];

  cmakeFlags = [ 
    "-GNinja"
  ];

  enableParallelBuilding = true;

  meta = {
    homepage = https://gaudi.web.cern.ch/gaudi/;
    description = "A basis for HEP experiment frameworks";
    platforms = stdenv.lib.platforms.unix;
    maintainers = with stdenv.lib.maintainers; [ chrisburr ];
  }
}
What have I done?
Moving the nix store directory

- Installed Nix inside docker without cvmfs mounted
- Built Nix changing /nix/ to /cvmfs/lhcbdev.cern.ch/nix/
- Built Nix again....
- And it works!!!!
- But the official binary cache can't be used anymore...
- Have since created a gitlab group: https://gitlab.cern.ch/lhcb-nix/
- bootstrap: Use GitLab CI to build nix with a custom store directory
- Also contains forks of hydra, nix, and nixpkgs
Moving the nix store directory

• Installed Nix inside docker without cvmfs mounted
• Built Nix changing `/nix/` to `/cvmfs/lhcbdev.cern.ch/nix/`
• Built Nix again....
Moving the nix store directory

- Installed Nix inside docker without cvmfs mounted
- Built Nix changing `/nix/` to `/cvmfs/lhcbdev.cern.ch/nix/`
- Built Nix again....
- And it works!!!!
- But the official binary cache can’t be used anymore...
• Installed Nix inside docker without cvmfs mounted
• Built Nix changing /nix/ to /cvmfs/lhcbdev.cern.ch/nix/
• Built Nix again....
• And it works!!!!
• But the official binary cache can’t be used anymore...
• Have since created a gitlab group: https://gitlab.cern.ch/lhcb-nix/
  • bootstrap: Use GitLab CI to build nix with a custom store directory
  • Also contains forks of hydra, nix and nixpkgs
Set up an instance of Hydra\(^2\) on openstack: \(http://lhcb-hydra.cern.ch:3000/\)

- Took less than an hour to get my first build Including setting up PostgreSQL!
- Uses the local machine for builds
- Since moved to using DBoD and GitLab CI to build a container

\(^2\)https://nixos.org/hydra/
Hydra build “farm”

- Set up an instance of Hydra\(^2\) on openstack: \(\text{http://lhcb-hydra.cern.ch:3000/}\)
- Took less than an hour to get my first build Including setting up PostgreSQL!
  - Uses the local machine for builds
- Since moved to using DBoD and GitLab CI to build a container
- Setting up an extra worker was trivial
  - Just need to be able to SSH to a machine with Nix
  - Docker container on \(\text{lblhcbpr3}\) with my build of Nix installed

\(^2\)\(\text{https://nixos.org/hydra/}\)
Hydra build “farm”

- Set up an instance of Hydra\(^2\) on openstack: \url{http://lhcb-hydra.cern.ch:3000/}
- Took less than an hour to get my first build Including setting up PostgreSQL!
  - Uses the local machine for builds
- Since moved to using DBoD and GitLab CI to build a container
- Setting up an extra worker was trivial
  - Just need to be able to SSH to a machine with Nix
  - Docker container on \texttt{lblhcbpr3} with my build of Nix installed
- Support for slaves with different architectures or extra features (AVX?)

\(^2\url{https://nixos.org/hydra/}\)
nixpkgs overlays

- nixpkgs has a concept of overlays that are applied to the main nixpkgs
nixpkgs overlays

• nixpkgs has a concept of overlays that are applied to the main nixpkgs

```
{lhcbsoftware.nix} 69 Bytes
1 { }
2 | self: super:
3 |
4 | { gaudi = super.callPackage ../pkgs/gaudi { }; }
5 |
6 
```

• Can also override existing packages or package arguments

```
{gcc-6.nix} 235 Bytes
1 { }
2 | self: super:
3 |
4 | { 
5 |     qt5 = super.qt59;
6 |     libsForQt5 = super.libsForQt59;
7 |     gcc = super.gcc6;
8 | # Some things really need gcc7
9 |     aws-sdk-cpp = super.aws-sdk-cpp.override { 
10 |         stdenv = super.overrideCC super.stdenv super.gcc7;
11 |     };
12 |
```
LHCb version of \texttt{nixpkgs}

- Use first overlay to add packages that are unsuitable for upstream
- Second overlay is an argument to \texttt{nixpkgs} to set the environment

![Diagram showing the relationship between upstream versions and overlays]
Creating environments

- **nixpkgs** can be used to create environments using `buildEnv`
  - Symlinked to the store directory, similar to an LCG view
  - To give a short but comprehensive example:

```nix
{nixpkgs ? builtins.FatchGit { url = https://gitlab.cern.ch/lhcb-nix/nixpkgs.git; ref = "master-lhcb"; } , name = "user_environment" , extraOverridePath = "gcc-7.nix" , extra_packages ? [] };

let
  user_environment = (buildEnv { name = name; paths = (builtins.concatLists []
    pkgs.coreutils
    pkgs.bash
    pkgs.gcc
    # Python 2
    (pkgs.root.override { python = python27; }).pythonlib
    pkgs.python27Full.withPackages(ps: [
      ps.matplotlib
      ps.pandas
    ])
    # Python 3
    (pkgs.root.override { python = python36; }).pythonlib
    pkgs.python36Full.withPackages(ps: [
      ps.matplotlib
      ps.pandas
      ps.snake
    ])})

  (builtins.map (s: pkgs.$s) extra_packages)
in user_environment
```
Creating environments

- You can then define multiple versions with different arguments
- I’ve created three as an example:

```nix
{nixpkgs}:
  with import nixpkgs;{}
let
  jobs:={
    # Some example environments for now
    example_environment_gcc6 = callPackage ./make_user_environment.nix {}
    .
    .
    .
    example_environment_gcc7 = callPackage ./make_user_environment.nix {}
    .
    .
    .
    gaudi_environment_gcc7 = callPackage ./make_user_environment.nix {}
    .
    .
    .
    # Make a replacement nixpkgs channel
    lhcb_nixpkgs = pkgs.releaseTools.channel {}
    .
    .
    .
  }
in jobs
```

- Full example stored at: https://gitlab.cern.ch/lhcb-nix/lhcb-environments
- Built in the lhcb-environments project on hydra
As most of the work is done upstream adding packages is easy

As these examples are designed to replace PATH entirely they contain:

• Shells: bash/zsh/tcsh/dash
• Standard utilities: coreutils/man/grep/tar/findutils/rsync/...
• Text editors: nano/vim/neovim/atom
• Version control: git/svn/hg

• Building: gcc/cmake/ninja/boost/libxml2/tbb/gperftools/...
• Debugging: gdb/lldb/valgrind

• TexLive 2017
• Python 2.7 with matplotlib/numpy/pandas/nose/jupyter/...
• Python 3.6 with matplotlib/numpy/pandas/snakemake/...
• XRootD with Python 2.7 and 3.6 bindings
• ROOT with Python 2.7 and 3.6 bindings
The example environments

• As most of the work is done upstream adding packages is easy
• As these examples are designed to replace PATH entirely they contain:
  • Shells: bash/zsh/tcsh/dash
  • Standard utilities: coreutils/man/grep/tar/findutils/rsync/...
  • Text editors: nano/vim/neovim/atom
  • Version control: git/svn/hg
  • Building: gcc/cmake/ninja/boost/libxml2/tbb/gperftools/...
  • Debugging: gdb/lldb/valgrind
  • TexLive 2017
The example environments

- As most of the work is done upstream adding packages is easy
- As these examples are designed to replace PATH entirely they contain:
  - Shells: bash/zsh/tcsh/dash
  - Standard utilities: coreutils/man/grep/tar/findutils/rsync/...
  - Text editors: nano/vim/neovim/atom
  - Version control: git/svn/hg
  - Building: gcc/cmake/ninja/boost/libxml2/tbb/gperftools/...
  - Debugging: gdb/lldb/valgrind
  - TexLive 2017
  - Python 2.7 with matplotlib/numpy/pandas/nose/jupyter/...
  - Python 3.6 with matplotlib/numpy/pandas/snakemake/...
The example environments

- As most of the work is done upstream adding packages is easy
- As these examples are designed to replace PATH entirely they contain:
  - Shells: bash/zsh/tcsh/dash
  - Standard utilities: coreutils/man/grep/tar/findutils/rsync/...
  - Text editors: nano/vim/neovim/atom
  - Version control: git/svn/hg
  - Building: gcc/cmake/ninja/boost/libxml2/tbb/gperftools/...
  - Debugging: gdb/lldb/valgrind
  - TexLive 2017
  - Python 2.7 with matplotlib/numpy/pandas/nose/jupyter/...
  - Python 3.6 with matplotlib/numpy/pandas/snakemake/...
  - XRootD with Python 2.7 and 3.6 bindings
  - ROOT* with Python 2.7 and 3.6 bindings
Try it for yourself in docker!
Try it for yourself in docker! (CERN only due to firewall)

• Install Nix:

1. docker run --rm -it centos:7 bash
2. useradd test
3. yum install -y bzip2
4. mkdir -p -m 0755 /cvmfs/lhcbdev.cern.ch/nix
5. chown test /cvmfs/lhcbdev.cern.ch/nix
6. cd /home/test
7. su test bash -c "curl -LO https://chrisburr.me/lhcb-nix-2.0/nix-2.0-2018_03_20-x86_64-linux.tar.bz2"
8. su test bash -c "curl https://chrisburr.me/lhcb-nix-2.0/install | sh"

• Install one (or more) of the environments in any directory:

example_environment_gcc6/example_environment_gcc7/gaudi_environment_gcc7

1. su test
2. . /home/test/.nix-profile/etc/profile.d/nix.sh
3. export LC_ALL=en_US.utf-8
4. export LANG=en_US.utf-8
5. mkdir -p "/cvmfs/lhcbdev.cern.ch/nix/environments/"
6. export LHCB_NIX_ENV_DIR="/cvmfs/lhcbdev.cern.ch/nix/environments/analysis_environment_gcc7"
7. nix-env -ir analysis_environment_gcc7 --profile "$LHCB_NIX_ENV_DIR" -Q -j8

• Set PATH and run!

1. su test
2. export LHCB_NIX_ENV_DIR="/cvmfs/lhcbdev.cern.ch/nix/environments/analysis_environment_gcc7"
3. export PATH="$LHCB_NIX_ENV_DIR;/bin"
4. export CMAKE_PREFIX_PATH="$LHCB_NIX_ENV_DIR"
5. export NIX_SSL_CERT_FILE=/etc/ssl/certs/ca-bundle.crt
6. bash
A few limitations of this setup

- Downloads are slow:
  - The binary cache is currently compressed on the fly by hydra
  - There is a setting to copy them to a directory/AWS/..
  - This can then be hosted on any web server
  - Plus packages are then signed automatically
- Package signatures aren’t checked (see above)
- Some packages have issues being built inside docker containers
My thoughts...
Why use Nix?

- Software built should be able to run on “any” flavour of Linux
  - Example works with CentOS 6, 7 and Ubuntu
  - Darwin should be fairly easy to add
  - Experimental support for AArch64

- Simpler environments
- Huge number of packages definitions already written
- Adding new package definitions is straightforward
- Active community, lots of very helpful experts on IRC
Why use Nix?

• Software built should be able to run on “any” flavour of Linux
• Simpler environments
  • No more (ab)use of LD_LIBRARY_PATH or PYTHON_PATH
  • Software with conflicting dependencies can be used at the same time

14,000

• Active community, lots of very helpful experts on IRC
Why use Nix?

- Software built should be able to run on “any” flavour of Linux
- Simpler environments
- Huge number of packages definitions already written ~14,000
  - Adding new software to an environment is a one line change
Why use Nix?

- Software built should be able to run on “any” flavour of Linux
- Simpler environments
- Huge number of packages definitions already written ~14,000
- Adding new package definitions is straight forward
  - The standard builder already works with most build systems
  - RUNPATH and other paths are set automagically
- Building Gaudi was trivial
  (once I had written definitions for all of it’s HEP specific dependencies...)
  (and fixed a bug? in the CMake config of the profiling module...)
Why use Nix?

- Software built should be able to run on “any” flavour of Linux
- Simpler environments
- Huge number of packages definitions already written ~14,000
- Adding new package definitions is straight forward
- Active community, lots of very helpful experts on IRC
What isn’t so good?

- Documentation is lacking some places
  - But it’s rapidly improving
  - Figuring things out from the source isn’t too difficult
What isn’t so good?

- Documentation is lacking some places
- The Nix expression language has a steep learning curve
  - I had never used a functional language like Haskell
  - Might have been easier if I had
- Doesn’t matter simple things like writing packages
What isn’t so good?

- Documentation is lacking some places
- The Nix expression language has a steep learning curve
- Independence from the host system isn’t perfect
  - I’ve read about issues with OpenGL/graphics drivers
  - Kernel
- Can’t be worse than what already exists
What isn’t so good?

- Documentation is lacking some places
- The Nix expression language has a steep learning curve
- Independence from the host system isn’t perfect
- Sometimes reproducible builds aren’t reproducible
  - Only seen this happen due to remote files being removed/changed
  - So long as the original nix store is kept there is always a copy
Conclusions

- Nix is awesome!
- I can see a lot of benefits and potential uses
  - Could avoid issues with missing or conflicting dependencies
  - Defining extra environments is easy (per analysis?/distributable?)
  - Can update old environments where needed (XRootD?)
- Useful resources and some other details in backup
Conclusions

- Nix is awesome!
- I can see a lot of benefits and potential uses
  - Could avoid issues with missing or conflicting dependencies
  - Defining extra environments is easy (per analysis?/distributable?)
  - Can update old environments where needed (XRootD?)
- Useful resources and some other details in backup

Any Questions?
Useful resources

Documentation:

- Introduction to Nix: https://nixos.org/nixos/nix-pills/
- Nix manual: https://nixos.org/nix/manual/
• **PYTHON_PATH** isn’t ideal as it is used by all Python versions
• **sitecustomize.py** is aimed for this purpose
  • Uses `$LHCB_NIX_ENV_DIR/lib/pythonX.Y/site-packages/`
• **ROOT** can’t be built with simultaneous Python 2 and 3 support
  • Instead make the Python library a separate package
  • Each is then loaded from `lib/pythonX.Y/site-packages`
• Using **TPython** from the **root** REPL uses Python 2
Debug symbols

- Stripped and deleted by default
- `stdenv.mkDerivation` has an option `separateDebugInfo`
- Makes a `-debug` package containing `lib/debug/.build-id/XX/YYYY`
- Can be loaded in GDB by modifying `~/.gdbinit` to contain:
  - `set debug-file-directory ENV_DIR/lib/debug`
  - There are probably other methods available