Maintaining Large Software Stacks in a Cray Ecosystem with Gentoo Portage

Colin A. MacLean
National Center for Supercomputing Applications
University of Illinois
Urbana, Illinois
Email: cmaclean@illinois.edu

***Abstract*—Building and maintaining a large collection of software packages from source is difficult without powerful package management tools. This task is made substantially more difficult in an environment where many libraries do not reside in standard search paths and the where loadable modules can drastically alter the build environment. The need to maintain multiple Python interpreters with a large collection of Python modules is one such case of having a large and complicated software stack and is the use case described in this paper. To address limitations of current tools, Gentoo Prefix was ported to Blue Waters, giving the ability to use the Portage package manager to track fine-grained software dependencies. This infrastructure allows for fine-grained dependency tracking, consistent build environments, multiple Python implementations, and easily customizable builds. This customized Gentoo Prefix infrastructure is used to build and maintain over 400 packages for Python support on Blue Waters.**

***Keywords*-package management; Python; Gentoo; Cray; HPC; Blue Waters;**

I. Introduction

Building and maintaining a large collection of software packages from source is difficult without powerful package management tools. Unlike binary package dependencies, full source-based dependency trees must also take into account configuration option dependencies, and build order. There are also multiple dependency types: build time, run time, and occasionally post-build dependencies. A fully configurable source-based package management system must therefore be able to account for the fine granularity of build options and be robust enough to detect the circular dependencies which can arise when building some packages. The package manager should also be able to handle package updates without rebuilding the entire tree and should be able to force packages depending on an updated package to be rebuilt if necessary. Complicating matters further, in an environment such as found on Crays, the package manager must also be capable of installing packages to a prefix and be aware of dependencies also installed in nonstandard locations. The build system must also be able to work around the occasional package configuration problems resulting from the compiler environment on Cray systems and easily apply additional patches.

The full complexity of this problem is particularly difficult to overcome when trying to maintain large Python stacks. In addition to the difficulties above, there are also multiple versions and implementations of Python interpreters, many Python packages may only be compatible with a subset of the available interpreters, and these packages may have shared dependencies. This means that the package management system should ideally be able to handle multiple python installs and be aware of python package compatibility restrictions in order to eliminate redundancy in dependencies which would occur if the build system were to be installed separately for each Python implementation.

II. Software on HPC Systems

Providing users a wide variety of software to use on HPC systems is a far from trivial challenge. The core operating system software often lags behind considerably from the bleeding edge releases of software many users wish to employ. Users may also want to build against specific versions of software for compatibility and consistency reasons. For this reason, it is customary to provide users with a selection of software packages and versions each installed into their own prefix directory. A tool such as Environment Modules[3, 9, 2] can then be used to simplify the setting of environment variables which make these non-standard software install locations visible. At the most basic, this involves adding the prefix's executable directory to `PATH` and its library path to `LD_LIBRARY_PATH`. Some may also set up `CPATH` and `LIBRARY_PATH` variables for compiler header and library search paths. It is also common for environment modules to set other variables unique to the module containing information such as the base prefix or version.[4]

III. Potential Package Managers and Build Systems

The focus for this project was primarily to produce the tooling required to build and maintain a feature-rich Python stack. However, the The first task was to examine the package managers available and to deter-

mine which could meet the necessary requirements with minimal patching and package repository maintenance.

A. Pip

Because the original scope was limited to the installation of Python, Pip[10] was examined as the first candidate. However, `pip` was quickly rejected due to its inability to build non-python dependencies which were either missing or too old on Blue Waters. Another problem was that some packages require the Cray compiler wrappers—`cc`, `CC`, and `ftn`—to build while other packages are unable to identify the compiler type using the wrappers and must invoke the compiler directly. Needing to adjust the environment variables for these situations would have made automating rebuilds and updates difficult.

B. SWTools

SWTools[8] is a simple system to automate the building and rebuilding of software modules for HPC systems. The build steps it runs are all manually written scripts. Installing, maintaining, and manually determining the dependency tree of a large collection of software packages using only hand-written scripts is beyond the scope of what SWTools was designed to support.

C. EasyBuild

EasyBuild[7, 4] is another framework to build and install system for HPC platforms. It is designed However, there are limitations when it comes to resolving dependencies, where it decides to install packages, making minor adjustments to the build process, and has a relatively small repository.

D. Anaconda

`conda`[1] is a package manager used by the Anaconda Python distribution to manage Python installations. Although the Anaconda system primarily uses binary packages, it is possible for `conda` to build packages from source. The most significant problem with using Anaconda, however, is that the build recipes make a lot of assumptions regarding compilers, library locations, and what libraries are already available on the system.

E. Gentoo Prefix

Gentoo Prefix[6] is a version of the Portage package manager and Gentoo repositories modified to to support prefixed installation. Gentoo is a highly customizable GNU/* metadistribution. Its primary usage is for installing and maintaining GNU/Linux systems from source but can also be used for GNU/FreeBSD and prefixed installations of a GNU userland on top of a variety of Linux and Unix kernels such as Solaris. It consists of bootstrapping scripts, patched Portage package manager, an auto-patched ebuild repository, and maintenance utilities. The package manager, Portage, is written in Python and controls the underlying build system which is written in Bash. The underlying Ebuild system includes an extensive Bash API used for writing ebuild files.

Portage has fine-grained dependency calculation. Each package can be represented as an package *atom*. The most basic package atom is represented as `category/package`. Optional version constraints can be added to the atom by prefixing with an operator (`>`, `>=`, `=`, `<=`, or `<`) and appending a version string. The version can contain a wildcard operator, `*`. There are also extended operators to match any revision of a version (`~`) and to prevent the installation of a specific version (`!`). These atoms are used to specify packages to install on the command line with `emerge`, specify a package in the per-package configuration files, and within ebuild files in the specification of dependency strings.

However, when building a program from source, the package name and version is insufficient to describe what is to be built. Source packages usually come with a variety of different configuration options. In Gentoo, these configuration options are abstracted using `USE` flags. A `USE` flag represents a functionality concept. For instance optional MPI functionality is controlled via the `mpi USE` flag. This abstraction allows for the inclusion or exclusion of different features independent of how those features are actually passed to whatever method of configuration a package uses. It is up to the ebuild files to translate those `USE` flags into actual configuration script arguments. `USE` flags can be specified in package atoms in brackets after the package name. This allows dependency calculations to take into account required features, forbidden features, or specify that the status of a feature in a dependency must be the same.

ebuild files for a particular package may also specify different `SLOT`s and optionally specify a `SUBSLOT` . Packages with ebuilds of different `SLOT`s can have one of each `SLOT` installed simultaneously unless a dependency string disallows it. `SLOT`s are also used frequently to indicate an API change. A package atom can specify a slot by appending `:SLOT` after the version. In an ebuild file's dependency strings, `:SLOT=` or `:=` can be added to a dependency atom. When calculating the dependency tree for an upgrade, if such a dependency is to be upgraded, the package will be rebuilt.

The main configuration of Gentoo is in `etc/portage/make.conf`[5]. This file can be used to specify variable settings such as `CC`, `CXX`, compiler flags, `USE` flags, and Python implementation targets. This provides the default build configuration for all packages. However, Portage supports modifying these

options on a per-package basis. These per-package settings are stored in `package.*` files or directories. If a directory is used, it is treated as if all files in the directory are concatenated together. The `package.accept_keywords` file is used to allow the installation of testing packages or packages with unknown status on the architecture. The `package.use` file is used to change `USE` flags from the defaults specified in `make.conf` on a per-package basis. `package.mask` is used to prevent the installation of package atoms specified within it. Finally, miscellaneous variables specified in `etc/portage/env/*` files may be applied to the build environment for particular packages by adding `package-atom file` to the `package.env`Portage also supports a `etc/portage/patches` directory. This allows for the user to apply patches to packages without needing to modify the ebuild.

IV. Package Manager Selection

After researching the available tools for package management, a decision was made to alter Gentoo Prefix for better compatibility with HPC software environments. There were several reasons for this decision.

First, the Python-specific package managers did not have the functionality to easily handle the complex build environment. Some configuration scripts are unable to determine the type of compiler the Cray wrappers invoke. This can lead to fatal configuration errors or misconfiguration. For these packages, the extra arguments passed to the compiler by the wrappers can be extracted, added to the compiler flag environment variables, and the unwrapped compiler invoked directly. However, other packages, particularly those using MPI, the compiler wrappers must be used. Thus, the build system must set up the environment variables differently for certain packages, preferably without having to manually edit the build specifications.

The HPC build and package management systems integrate well into the complex build environment. However, these are not well suited for the more monolithic deployment of a Python stack. The previous Python stack on Blue Waters was a collection of several dozen different Environment Modules managed by SWTools. This method of deployment would not scale well to hundreds of packages used at once. Even with several dozen modules, the need for Python to search each path on every import did not work well with the Lustre filesystem. SWTools also requires manual creation of build scripts. While this is not troublesome for single software package installs, the maintenance requirements are too high to build a comprehensive Python stack. EasyBuild has EasyConfigs and EasyBlocks to automate the build process. However, it lacks strong dependency calculation abilities.

While Gentoo Prefix lacked awareness of environment modules, attempting to ignore as much of the host system as possible, its robust source-based package management features are ideal for managing a large collection of software with a highly customized build process. Thus, it was determined that it would be easier to port Gentoo Prefix to a Cray host and add support for Environment Modules than to add Portage-level dependency resolution features to an HPC-centric package manager. Gentoo also has large repositories of ebuilds already available. While some of those ebuilds would require modification, the per-package environment modification features was determined likely be able to handle most of these cases.

V. Porting Gentoo Portage to Blue Waters

A. Compiler Environment and Library Paths

There are several ways in which a compiler can be made aware of a prefix. The method used by the unpatched Gentoo Prefix is to build its own GCC configured with `--sysroot=$EPREFIX`. This hard-codes the prefixed library directories and include as if they were `/lib64`, `/usr/lib64`, and `/usr/include`. However, on a Cray, Gentoo Prefix must be able to use a host system compiler, which is not built specifically for the prefix. There is also a problem with adding the prefixed locations as `-I` and `-L` options due to directory search ordering errors when building some packages. The three possibilities left for Gentoo Prefix are to supply GCC with a customized `specs` file, set `CPATH` and `LIBRARY_PATH`, or build a prefixed libc and manage a prefixed `ld.so.conf`. The `CPATH` and `LIBRARY_PATH` method was chosen as the easiest alternative.

B. Bootstrapping Script

Gentoo Prefix has two bootstrapping scripts. The first bootstraps Bash for host platforms lacking the shell and isn't necessary on a Linux host. The second bootstraps Portage and its dependencies into the specified `EPREFIX`. Because Gentoo Portage is designed to be run with as few host dependencies as possible—the kernel and a handful of core libraries—it needed to be modified in order to expose host system `PATH`, libraries, and compilers. These dependencies provided by the system or modules also had to be removed from the boostrapping procedure. The bootstrapping script also sets an initial configuration for Portage and repositories and these defaults also had to be modified for use in the Cray environment and with host-provided compilers.

C. Adding a Blue Waters profile

A profile in Portage contains the default settings for a platform. The Blue Waters profile forces on the `cray`

USE flag, which is used to make Cray-specific changes to the build process in a few ebuilds. A profile.bashrc is used to make a few minor adjustments to the build environment for a handful of core packages installed during the boostrapping process. package.provided is used to ignore dependencies which are already available on the host system.

D. Patching Portage

Portage was patched to add support the Environment Modules system and to expose the host environment to the build process, which starts from a non-interactive non-login Bash environment.

An old patch for Gentoo Prefix was resurrected in order to add prefix chaining support to Portage. Prefix chaining allows the creation of a light weight Gentoo Prefix which is able to the utilities and dependencies of parent prefixes (See Section VI-C. This mostly consisted in manually re-applying the patch due to movement and minor changes of the affected code. The patch was also improved by adding support for a user to use a prefix installed by another user. The prefix chaining setup utility was also updated and now has Environment Modules support.

E. Patching *etc/profile*

The Gentoo Prefix maintenance shell and Portage both rely on the prefix's etc/profile to set up their environments. This file was patched to source the host /etc/profile, expose host files in the environment variables, and to load a default set of Environment Modules. This file is generated by the baselayout-prefix ebuild and does not require manual editing of full paths.

An environment variable containing RPATH arguments for the compiler/linker is generated when etc/profile it sourced. Gentoo Prefix was patched to use RUNPATH the method for finding library locations to stop Prefix libraries from interfering with system binaries while the module is loaded. The unpatched Gentoo Prefix used its own compiler built with --sysroot, which would not work with the compiler wrappers. --enable-new-dtags is passed to the linker in order to allow LD_LIBRARY_PATH to override RUNPATH if necessary. Prefix locations for CPATH and LIBRARY_PATH are used to make the compiler and linker aware of the prefix, ensuring that these paths are always searched after -I and -L arguments generated by package configuration scripts.

etc/profile can be optionally patched to enable prefix chaining support (See Section VI-C). When prefix chaining support is enabled, the etc/profile is patched to recursively load the environment from the parent prefixes.

VI. Features

A. Environment Modules support

Environment Modules support is primarily intended to be used on a per-package basis through package.env to modify a default set of Environment Modules loaded via etc/profile. The variable ENVMOD can be used to load, unload, and swap modules:

Listing 1. ENVMOD usage

```
#loads module
ENVMOD="module"
#unloads a module
ENVMOD="-module"
#swaps module1 for module2
ENVMOD="%module1:module2"
```

These operations are invoked from left to right as a space separated list. Ebuild files may use the variables ENVMOD_REQUIRE and ENVMOD_RESTRICT to list modules which must be loaded by the configuration or cannot be loaded. In all cases module may either be used as a specific version or for all version.

B. Environment Modules module for *eselect*

Gentoo uses the utility eselect to manage its configuration. A module for eselect was written to give a command line interface for generating the default environment for the prefix. This utility allows the prefix maintainer to select a base set of modules to load. The environment that Environment Modules generates is saved to etc/env.d. The env-update command can then be used to update etc/profile.env, which gets sourced by etc/profile.

C. Prefix Chaining

Prefix chaining support was added to Portage as a means of supporting multiple package configurations. This feature constructs a lightweight Gentoo Prefix consisting of several configuration files. The prefix environment makes use of the software installed in parent prefixes in its PATH, compiler search directories, and dependency calculations. If a package dependency in the parent prefix doesn't exist, is too old, or needs to be built with different configuration options, then Portage will install these dependencies in the chained prefix. The chained environment is configured to allow these packages to override those provided by the parent prefixes. The package python-chaining adds a sitecustomize.py file to each of the installed Python implementations, allowing paths and eggs in chained prefixes to be visible to Python.

The chained prefix has its own make.conf, allowing packages to be built with different options such as USE flags, compiler flags, and compiler. This feature is

used on Blue Waters to create the bwpy-mpi environment on Blue Waters. The base Gentoo Prefix, bwpy offers a collection of Python interpreters and packages built for use in a single-node environment where MPI is unavailable, such as on the login nodes. bwpy-mpi sets the mpi USE flag and the packages which have this flag, such as h5py, are rebuilt. This provides MPI-enabled Python stack for use on compute nodes. This feature could also be used to provide a choice between packages linked against scientific library implementations.

Prefix chaining is able to distinguish between the different types of dependencies: DEPEND (build-time dependencies), RDEPEND (run-time dependencies), HDEPEND (host dependencies for cross compiling), and PDEPEND (post-merge dependencies). Choosing not to inherit run-time and post-merge dependencies from the parent prefix is necessary for building a child prefix with a different CHOST and is also useful for keeping Portage and its utilities separate from the target software bundle.

If the prefix-chain-setup script detects that the owner of the parent prefix and the chained prefix are the same, it will generate a modulefile for Environment Modules and install it into the parent prefix's usr/modulefiles directory.

Multiuser prefix chaining support was also added. This allows other users to use dependencies from Gentoo Prefix environments created by other users. This makes it possible for a user to create a chained prefix in their home directory and install software using Portage. This could be useful for the easy install of specific versions and configurations of packages like PETSc.

D. Prefix Support for revdep-rebuild.sh

revdep-rebuild.sh is a script which checks the linking of every dynamic library in the prefix and finds the packages those libraries belong to. It is used to rebuild packages which have broken linkage or link to a specified library. This is useful to perform checks and rebuilds after system updates. This utility was not written to include support for a prefixed Portage and search directory, so the necessary changes were made to make it work on systems with a Linux host.

E. Package Retesting

Preliminary work has been started to enable a prefix maintainer to rerun ebuild test phases to and report packages which have broken due to system updates. This requires setting FEATUES="noclean" to keep the build directories. By setting a number of Bash environment variables, Portage's ebuild.sh can be invoked to rerun the test phase of the emerge process.

F. Creating New Releases

It is often unnecessary to re-bootstrap Gentoo Prefix to create a new release of the prefix. Gentoo Prefix contains a script, usr/lib/portage/bin/chpathtool.py, which can be used to change the prefix for all files in a directory provided that the new prefix directory contains the same or fewer character in the path. If the new prefix is of shorter length than the old prefix, the path is padded with repeated slashes when adjusting binary files. Due to this restriction, it is recommended to use a versioning scheme for the prefix directory which either includes padding zeros or will always be the same length. This script is primarily used internally to adjust prefixed binary packages, but is also useful for copying a prefix to create a new release with updated packages. The script is invoked as chpathtool.py location old-prefix new-prefix and will update all files in location.

It may also be desirable to create a minimal prefix containing only the packages necessary for Portage to run. The new prefix could then be created by copying this minimal prefix and adjusting it or by using prefix chaining. The new release can then be created quickly by using binary packages from the old prefix. To create binary packages from the old prefix, the old prefix can either run quickpkg "*/*" or the buildpkg feature could have been enabled in old prefix:

Listing 2. old-prefix/etc/portage/make.conf

```
FEATURES="${FEATURES} buildpkg"
```

Then by using the --usepkg for emerge, the binary packages in PKGDIR are taken into consideration during dependency calculations. Thus, recompilation of most packages without updates can be avoided.

VII. Example of Porting a Broken ebuild

One example of a package which needs a modified ebuild is h5py. This ebuild was modified as follows:

Listing 3. h5py-2.5.0.ebuild excerpt

```
...
IUSE="cray doc test examples mpi"
...
pkg_setup()
  if use mpi; then
    if use cray; then
      export CRAY_ADD_RPATH=yes
      export CRAYPE_LINK_TYPE=dynamic
      export CC=cc
    else
      export CC=mpicc
    fi
  fi
```

```
}
...
python_configure() {
  esetup.py configure $(usex mpi --
      mpi '')
}
...
```

For this package, the problem was that the `CC` variable was being set to `mpicc` when MPI functionality is enabled. Because Crays use `cc` instead of `mpicc`, this package was failing to build. The fix was straight forward. First, `cray` was added to the IUSE variable. This `USE` flag is forced on by the Blue Waters Portage profile. The ebuild is then modified to conditionally export the necessary environment settings if the `cray` `USE` variable is enabled. This way, it is possible for this ebuild to be pushed upstream to the Gentoo Prefix or Science overlay without breaking the package for other systems.

VIII. Evaluation of Gentoo Prefix on Blue Waters

Gentoo Prefix is used to manage the `bwpy` Python stack on Blue Waters. Of the 414 packages installed in `bwpy`, only 18 of the ebuilds required any modification to build and install correctly. Cray-specific Maintenance of these customized ebuilds when new versions are released is usually just a matter of copying the ebuild. 8 new ebuilds were also created for software lacking ebuilds. These new and modified ebuilds are available in the Blue Waters Python Gentoo Portage overlay.

Per-package modification of the environment was not required to be used as often as initially thought. Most of these environment changes were patched into the ebuilds to make redeployment and sharing of build settings easier.

Because most packages can be installed with one command, it has become much easier to support Python users on Blue Waters. In the event that a user has trouble building a Python package that is not provided by `bwpy` , the ease of package management makes it reasonable to simply install the package into `bwpy`. For example, one user was experiencing difficulty with installing the Numba package. This Python package uses LLVM to generate optimized sections of Python code. Gentoo Prefix was able to successfully build LLVM and Numba in a single command without any modifications to the ebuild or build environment.

IX. Conclusion

Building and maintaining large software stacks such as Python in the complex environment of a Cray was difficult with previous tools. Python-specific package managers were lacking in complex from-source build features and dependency calculation ability. HPC-centric build systems were highly capable for individual packages but were also lacking in fine-grained software dependency features. Gentoo Prefix has robust source-based package management, but lacked support for integration into HPC build environments. It was determined that porting Gentoo Prefix to Cray would require less effort than adding Portage-level package management capabilities to a HPC-centric build system.

Gentoo Prefix was patched to open up its environment settings to the Cray compilers and modules. Support for the Environment Modules in the ebuild build environment was added by patching Portage and adding an `eselect` module for Environment Modules. An old patch enabling prefix chaining was revived for use in building alternative configurations of packages and later to allow users access to the Portage management system in their home directories.

The `bwpy` Python stack on Blue Waters is managed by this patched version Gentoo Prefix for use on Crays. It is used to manage over 400 packages. Most of these packages can be built straight from standard ebuild repositories without modification. This allows Blue Waters to support an optimized and comprehensive Python stack with easy maintenance.

The Blue Waters Gentoo Prefix Overlay and bootstrapping scripts can be found at https://github.com/camaclean/bw-python-gentoo-prefix-overlay.

References

[1] Continuum Analytics, Inc. *Anaconda*. Version 4.0.0. 2016. URL: https://www.continuum.io/.

[2] D. Eadlin. *Keeping It Straight: Environment Modules*. URL: http://www.admin-magazine.com/HPC/Articles/Managing-the-Build-Environment-with-Environment-Modules.

[3] J. L. Furla. "Providing a Flexible User Environ". In: *Proceeding of the Fifth Large Installation System Administration (LISA V)* (1991), pp. 141–152.

[4] Markus Geimer, Kenneth Hoste, and Robert McLay. "Modern scientific software management using easybuild and lmod". In: *Proceedings of HUST 2014: 1st International Workshop on HPC User Support Tools* (2014), pp. 41–51. DOI: 10.1109/HUST.2014.8. URL: http://hpcugent.github.io/easybuild/files/hust14%7B%5C_%7Dpaper.pdf.

[5] Gentoo. *make.conf(5)*. 2016. URL: https://dev.gentoo.org/~zmedico/portage/doc/man/make.conf.5.html.

[6] Gentoo Foundation, Inc. *Gentoo*. 2016. URL: https://www.gentoo.org.

[7] Ghent University. *EasyBuild*. Version 2.7.0. Mar. 20, 2016. URL: http://hpcugent.github.io/easybuild/.

[8] ORNL. *SWTools*. Version 1.0.0. Jan. 1, 2011. URL: https://www.olcf.ornl.gov/center-projects/swtools/.

[9] P. W. Osel and J. L. Furla. "Abstract yourself with Modules". In: *Proceeding of the Tenth Large Installation System Administration (LISA '96)* (1996), pp. 193–204.

[10] PyPA. *Pip*. Version 8.1.1. Mar. 17, 2016. URL: https://pip.pypa.io.