Making Scientific Software Installation Reproducible On Cray Systems Using EasyBuild

Petar Forai
Research Institute of
Molecular Pathology
Dr Bohrgasse 7
A-1030 Vienna, Austria
petar.forai@imp.ac.at

Kenneth Hoste
Ghent University
Krijgslaan 281, S9
B-9000 Ghent, Belgium
kenneth.hoste@ugent.be

Guilherme Peretti-Pezzi
Swiss National
Supercomputing Centre
Via Trevano, 131
6900 Lugano, Switzerland
peretti@cscs.ch

Brett Bode
National Center for
Supercomputing Applications
University of Illinois
1205 W. Clark St.
Urbana, IL 61801
brett@illinois.edu

ABSTRACT

Cray provides a tuned and supported OS and programming environment (PE), including compilers and libraries integrated with the modules system. While the Cray PE is updated frequently, tools and libraries not in it quickly become outdated. In addition, the amount of tools, libraries and scientific applications that HPC user support teams are expected to provide support for is increasing significantly. The uniformity of the software environment across Cray sites makes it an attractive target for to share this ubiquitous burden, and to collaborate on a common solution.

EasyBuild is an open-source, community-driven framework to automatically and reproducibly install (scientific) software on HPC systems. This paper presents how EasyBuild has been integrated with the Cray PE, in order to leverage the provided optimized components and support an easy way to deploy tools, libraries and scientific applications on Cray systems.

We will discuss the changes that needed to be made to EasyBuild to achieve this, and outline the use case of providing the Python distribution and accompanying Python packages on top of the Cray PE, to obtain a fully featured 'batteries included' optimized Python installation that integrates with the Cray-provided software stack. In addition, we will outline how EasyBuild was deployed at the Swiss National Supercomputing Centre CSCS and how it is leveraged to obtain a consistent software stack and installation workflow across multiple Cray (XC, CS-Storm) and non-Cray HPC systems.

1. INTRODUCTION

The job of High-Performance Computing (HPC) support teams is to enable the productive and efficient use of HPC resources. This ranges from helping out users by resolving issues like login problems or unexpected software crashes, providing detailed answers to both simple and deeply technical questions, installing requested software libraries, tools and applications, to providing services such as performance analysis and optimization of scientific software being developed.

One particularly time-consuming task for HPC support teams is installing (scientific) software. Due to the advanced nature of supercomputers (i.e., multiple multi-core processors, high performance network interconnect, need for most recent compilers and libraries, etc.), compiling software from source on the actual operating system and for the system architecture that it is going to be used on is typically highly preferred, if not required, as opposed to using readily available binary packages that were built in a generic way.

Support teams at HPC sites worldwide typically invest large amounts of time and manpower tackling this tedious and time consuming task of installing the (scientific) software tools, libraries, and applications that researchers are requesting, while trying to maintain a coherent software stack.

EasyBuild is a recent software build and installation framework that intends to relieve HPC support teams from the ubiquitous burden of installing scientific software on HPC systems. It supports fully automating the (often complex) installation procedure of scientific software, and includes features specifically targeted towards HPC systems while providing a flexible yet powerful interface. As such, it has quickly grown to become a platform for collaboration between HPC sites worldwide. We discuss EasyBuild in more detail in Section 2.

The time-consuming problem of getting scientific software installed also presents itself on Cray systems, despite the extensive programming environment that Cray usually provides. Both HPC support teams and end users of Cray systems struggle on a daily basis with getting the required tools and applications up and running (let alone doing so in a somewhat optimal way). While a policy for providing software installations in a consistent way to end users is a common goal, the reality more often than not is that shortcuts are being taken to speed up the process, resulting in a software stack that is organised in a suboptimal (and sometimes outright confusing) manner, with little consistency in the software stacks provided on different systems beyond what is provided by Cray. Hence, it is clear that EasyBuild could also be useful on Cray systems.

Originally, the main target of EasyBuild was the standard GNU/Linux 64-bit x86 system architecture that is commonplace on today's HPC systems. On these systems there typically is an abundant lack of a decent (recent) basic software stack, i.e., compilers and libraries for MPI, linear algebra, etc., to build scientific applications on top of. EasyBuild was designed to deal with this, by supporting the installation of compilers and accompanying libraries, and through the toolchain mechanism it employs (see Section 2.2.4). As such, it was not well suited to Cray systems, where a well equipped programming environment is provided that is highly recommended to be used.

In this paper, we present the changes that had to made to EasyBuild to provide stable integration with the available programming environment on Cray systems (see Section 3). In addition, we outline the use case of installing Python and several common Python libraries using EasyBuild on Cray systems (Section 4), and discuss how EasyBuild deployed at the Swiss National Supercomputing Centre CSCS (Section 5).

2. EASYBUILD

EasyBuild [17–19] is a tool for building and installing scientific software on HPC systems, implemented in Python. It was originally created by the HPC support team at Ghent University (Belgium) in 2009, and was publicly released in 2012 under an open-source software license (GPLv2). Soon after, it was adopted by various HPC sites and an active community formed around it. Today, it is used by institutions worldwide that provide HPC services to the scientific research community (see Section 2.4).

2.1 Goals

The main goal of EasyBuild is to fully *automate* the task of building and installing (scientific) software on HPC systems, including taking care of the required dependencies and generating environment module files that match the installations. It aims to be an expert system with which all aspects of software installation procedures can be performed autonomously, adhering to the provided specifications.

In addition, EasyBuild serves as a *platform for collaboration* where different HPC support teams, who likely apply different site policies concerning scientific software installations, can efficiently work together to implement best practices and benefit from each others expertise.

It aims to support achieving reproducibility in science, by enabling to easily*reproduce* software installations that were performed previously. It allows sharing of installation recipes in order to enable others to perform particular software installations. Several features are available in EasyBuild to facilitate that; see also Section 2.3.5.

EasyBuild intends to be *flexible* where needed so that HPC support teams can implement their own site policies, ranging from straightforward aspects like the installation prefix for software and module files, to the module naming scheme and the particular compiler and libraries being employed; see also Section 2.3.6.

2.2 Terminology

Before going into more detail, we introduce some terminology specific to EasyBuild that will be used throughout this paper.

2.2.1 EasyBuild framework

The *EasyBuild framework* is the core of the tool that provides the functionality that is commonly needed for building and installing scientific software on HPC systems. It consists of a collection of Python modules that implement:

- the `eb` command line interface
- an abstract software installation procedure, split up into different steps including configuring, building, testing, installing, etc. (see Section 2.3.1)
- functions to perform common tasks like downloading and unpacking source tarballs, applying patch files, autonomously running (interactive) shell commands and capturing output & exit codes, generating module files, etc.
- an interface to interact with the modules tool, to check which modules are available, to load modules, etc.
- a mechanism to define the environment in which the installation will be performed, based on the compiler, libraries and dependencies being used (see also Section 2.2.4)

2.2.2 Easyblocks

The implementation of a particular software installation procedure is done in an *easyblock*, a Python module that defines, extends and/or replaces one or more of the steps of the abstract procedure defined by the EasyBuild framework. Easyblocks leverage the supporting functionality provided by the EasyBuild framework, and can be viewed as 'plugins'.

A distinction is made between *software-specific* and *generic* easyblocks. Software-specific easyblocks implement a procedure that is entirely custom to one particular software package (e.g., OpenFOAM), while generic easyblocks implement a procedure using standard tools (e.g., CMake, `make`).

Each easyblock *must* define the configuration, build and install steps in one way or another; the EasyBuild framework leaves these steps purposely unimplemented since their implementation heavily depends on the tools being used in the installation procedure. Since easyblocks are implemented in an object-oriented scheme, the step methods implemented by a particular easyblock can be reused in others through inheritance, enabling code reuse across easyblocks.

For each software package being installed, the EasyBuild framework will determine which easyblock should be used, based on the value of the `easyblock` parameter, or the name of the software package if no easyblock is specified.

2.2.3 Easyconfig files

Easyconfig files contain sets of key-value definitions for the parameters that are supported by the EasyBuild framework, which are also referred to as *easyconfig parameters.*

They specify what software package and version should be installed, which compiler toolchain should be used to perform the installation (see also Section 2.2.4), and which specific versions of the required dependencies should be made available. In addition, they provide some basic metadata (short description, project homepage, . . .), allow to specify custom settings for the build, and so on.

An example of an easyconfig file is shown in Listing 3.

2.2.4 Toolchains

A *compiler toolchain*, or simply *toolchain* for short, is a set of compilers (typically for C, C++, & Fortran) typically combined with special-purpose libraries to support specific functionality, for example an MPI library for distributed computing, and libraries that provide heavily tuned routines for commonly used mathematical operations (e.g., BLAS, LAPACK, FFT).

For each software package being installed with EasyBuild, a particular toolchain is being used. The EasyBuild framework prepares the build environment for the different toolchain components, by loading their respective modules and defining environment variables to specify compiler commands (e.g., via `$CC`), compiler and linker options (e.g., via `$CFLAGS` and `$LDFLAGS`), etc. Easyblocks can query which toolchain components are being used and steer the installation procedure to be performed accordingly.

2.3 Features

We briefly present the main features of EasyBuild below; for a more detailed overview we refer to [17–19] and the EasyBuild documentation [2].

2.3.1 Step-wise installation procedure

Each installation is performed in several steps that are defined in the EasyBuild framework, see Figure 1. The whole chain of steps is executed for each of the provided easyconfig files and, if necessary and desired, also for each of the missing dependencies (see Section 2.3.3). If a problem occurs during one of the steps, the remaining steps are cancelled.

After parsing the easyconfig file, the build directory is created by obtaining and unpacking the source files, and applying specified patch files (if any). Next, the build environment is set up based on the toolchain and dependencies being used. Subsequently, the common configure-build-install cycle is performed; if a test mechanism to verify the build is supported (e.g., `make check`) it is run before the 'install' step. For software packages that support the notion of 'extensions', e.g., Python (packages), Perl (modules), R (libraries), etc., the listed extensions are also installed. The 'sanity check' step performs a couple of small checks to verify the installation, i.e., checking whether prescribed files and directories are present, and whether simple check commands (e.g., importing a Python package) can be executed successfully. If the sanity check passes, the build directory is cleaned up and an environment module file that matches the installation is generated. Finally, a couple of final (optional) steps are performed: changing of permissions (e.g., to protect the installation for a particular group of users, or the make it read-only), creating a package (e.g., an RPM) for the installed software, and running test cases (if any are specified) just like a user would.

Important to note is that each installation is done in a separate installation prefix, enabling multiple versions and builds of the same software package to be installed side-by-side.

2.3.2 Generating module files

For each successful software installation the EasyBuild framework generates a corresponding module file, which specifies the changes to the environment that are required to use the software. In addition to being the canonical way of giving users access to the installed software, these modules are also employed by EasyBuild itself to resolve dependencies (see Section 2.3.3).

The generated module file specifies updates to environment variables like `$PATH`, `$LD_LIBRARY_PATH`, etc. to make binaries and libraries available. The EasyBuild framework does this automatically by checking for standard sub-directories like `bin`, `lib`, etc. Additional non-standard paths to be considered can be specified in easyblocks or easyconfig files. Likewise, additional environment variables that should be manipulated by the generated module file can be specified.

The module naming scheme to be used can be implemented in a simple Python module that prescribes how a module name should be constructed based on a set of easyconfig parameters. A couple of common module naming schemes are provided out-of-the-box, including the one used by default (`<name>/<version>-<toolchain>-<versionsuffix>`).

2.3.3 Dependency resolution

The dependency resolution mechanism that is part of the EasyBuild framework makes it trivial to install a software application and all requires dependencies (including compiler toolchain) *with a single command*.

For each dependency, EasyBuild will check whether a corresponding module file is available to resolve that dependency. If not, and the 'robot' mode is enabled, it will search for a matching easyconfig file for each missing dependency, in a set of locations that can be controlled by the user. The easyconfig files for the missing dependencies are then processed in the order prescribed by the directed graph that was constructed based on the dependency specifications, to perform the required installations.

2.3.4 Logging

EasyBuild keeps a thorough log of how an installation was performed. This can be used in case of problems during the installation to debug the problem at hand, or to re-evaluate a successful installation procedure later, for example when problems emerged with the installed software. During the installation, the log file is kept in a temporary location that is clearly mentioned in the output of the '`eb`' command. After a successfull installation, the log file is copied to a sub-directory of the installation directory.

EasyBuild log files are well structured: each log message starts with a specific prefix tag (to distinguish it from output of shell commands being executed), and includes a time stamp and the location in the code where the log message was sent from. Additional useful information is provided in particular cases, like the location where shell commands were run, environment variables that were defined, exit codes of shell commands, etc.

2.3.5 Reproducibility

One of the design goals of EasyBuild is to allow for easily reproducing an installation that was performed earlier, a base requirement for reproducible science. Although there are potential external influences that EasyBuild does not control (yet), all major aspects of the installation procedure are strictly defined somewhere. Global settings like the prefix

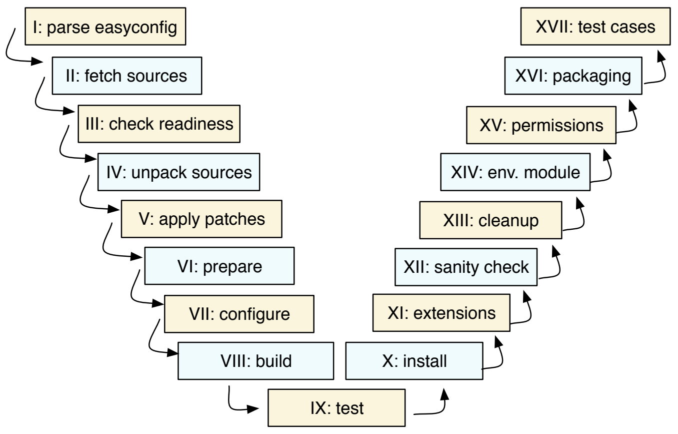

Figure 1: Step-wise installation procedure performed by EasyBuild.

path for software installations are specified in the EasyBuild configuration, while aspects specific to the particular software being installed and its installation procedure are encoded in the easyblock and easyconfig file. The easyconfig file may in addition also define parameters that are specific to the particular software version or toolchain being used.

With this approach it becomes very easy to reproduce a particular installation, since it basically comes down to using the same easyconfig file as was used before. For this reason, the easyconfig file is copied to the installation directory for each successful installation. In addition, it implies that sharing easyconfig files and updates to easyblocks is typically sufficient to enable others to replicate an installation. This has been a major factor influencing the growth of the EasyBuild community (see also Section 2.4).

2.3.6 Flexibility

Another important design goal of EasyBuild is to provide ample flexibility so HPC sites can implement their own site policies. This is reflected in several ways: how EasyBuild can be configured, by the dynamic nature of the EasyBuild framework, and in the different features that provide control over the installation procedures that will be performed.

First of all, EasyBuild can be configured in different ways: via system-level and user-level configuration files, through environment variables, and using the command line interface (CLI). These different configuration levels precede one another in an intuitive way, i.e., environment variables overrule settings defined in configuration files, while settings specified using the CLI get preference over the value of the corresponding environment variables. This allows putting a central EasyBuild configuration in place, without limiting the ability to reconfigure EasyBuild easily on a per (shell) session basis for testing or experimental purposes.

Next, the EasyBuild framework is very dynamic in the sense that it can be easily extended or enhanced at runtime, by providing additional or customized Python modules that can overrule existing ones. Through this mechanism, specific easyblocks, compiler toolchains, module naming schemes, etc. can be dynamically included, which will be picked up as needed by the EasyBuild framework, regardless of whether they are part of the EasyBuild installation being used or not.

Finally, EasyBuild supports different ways to tweak its default behaviour. This ranges from providing a wealth of configuration settings to control different aspects of the installation procedure, to being able to specify alternate paths that should be considered (first) when searching for easyconfig files, to even redefining or altering particular functionality of the EasyBuild framework using custom implementations; for example, the module naming scheme being used.

2.3.7 Transparency

A common concern is that EasyBuild may be too much of a ‘black box’. To address this, the EasyBuild framework provides a number of features to improve transparency. In particular, it supports the notion of performing a ‘dry run’ installation, i.e., reflecting what EasyBuild would do given a current configuration and (set of) provided easyconfig file(s).

The dependency resolution mechanism can be queried to get an overview of the full dependency graph, to see where the various dependencies come into play and for which of them matching module files are already available. Additionally, the installation procedure that would be performed by EasyBuild for a given easyconfig file can be consulted in detail, in a matter of seconds. Not only does this provide a way to gain confidence in EasyBuild itself and in the employed workflow using the tool, it is also useful when developing, debugging and evaluating easyblocks and easyconfig files.

2.4 EasyBuild Community

One other, non-technical, feature of EasyBuild is its active and quickly growing community. Shortly after the first public release in 2012, EasyBuild was picked up by HPC user support teams desperately looking for a better way to deal with their daily struggles of getting scientific software installed.

Today, EasyBuild is used by HPC sites around the world, ranging from small sites all across Europe, large European sites like Jülich Supercomputing Centre (Germany) and the Swiss National Supercomputing Centre (CSCS, see also Section 5), several institutions outside of Europe including Stanford University (US), Ottawa Hospital Research Institute (OHRI, Canada), New Zealand eScience Infrastructure (NeSI), etc., to even large commercial companies like Bayer (Germany).

Not only is the EasyBuild userbase already substantial and ever expanding, a lot of the HPC sites using it are also active in the community by participating in discussions, attending meetings, etc., and are actively contributing back in various

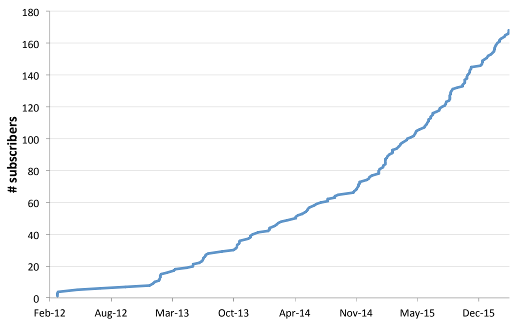

Figure 2: Number of subscribers to the EasyBuild mailing list, over time.

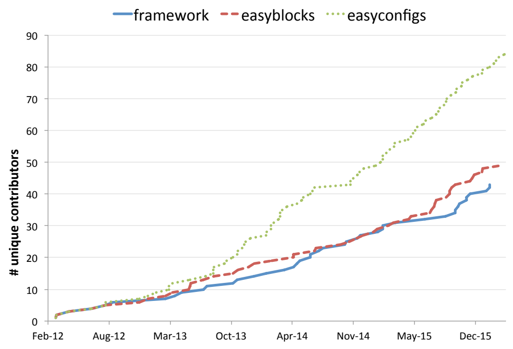

Figure 3: Number of (unique) contributors to the different central EasyBuild repositories, over time.

ways, by reporting and fixing bugs, sending back enhancements to easyblocks/easyconfigs and adding new ones, and also sharing ideas for (and implementations of) additional features in the EasyBuild framework.

Figures 2 and 3 give a quantified view of the growth of the EasyBuild community, in terms of subscribers to the EasyBuild mailing list and unique contributors to the central EasyBuild repositories where resp. the framework, easyblocks and easyconfig files are being hosted.

2.4.1 Features for and by the community

The community-oriented aspect of EasyBuild is also reflected in some of its features. As already discussed in Sections 2.1 and 2.3, various features were included to provide amply control over EasyBuild's behaviour and to support implementing local site policies.

Additionally, there are also several features that were implemented as a reaction to the many contributions being sent back. In particular, there is a tight integration with the GitHub platform where the EasyBuild code repositories are hosted. Support is available to obtain easyconfig files from GitHub pull requests, to send back test reports for those pull requests, review pull requests, and even to create and update pull requests straight from the '`eb`' command line interface, in an attempt to lower the bar to entry and streamline the workflow for contributing back.

Many of the features in place today were also either implemented by popular request from the community, or even by members from the community themselves. This confirms that EasyBuild has grown beyond a tool for the community, into a tool *by* the community and truly serving as a platform for collaboration.

3. EASYBUILD ON CRAY SYSTEMS

EasyBuild supports bringing up an entire software stack from the ground up, including compilers and accompanying libraries. However, this is not the approach recommended by the system vendor on platforms where a well equipped and optimized software stack of basic tools and libraries is readily provided.

On Cray systems, the *Cray Programming Environment* (Cray PE) provides a robust software environment that is tuned to the underlying hardware, including different compiler suites, the Cray MPT library, and various additional tools and libraries that were optimized by Cray.

In this section, we will discuss in detail what changes needed to be made in EasyBuild to provide stable support for installing scientific software on top of the readily available software stack provided on Cray systems.

3.1 Cray Programming Environment

Before going into detail, we outline the aspects of the Cray Programming Environment (or Cray PE, for short), sometimes also referred to as Cray Application Developer's Environment (CADE), that are relevant to the subsequent discussion. For more details, we refer to the Cray documentation [20].

3.1.1 Modules interface

The Cray PE is implemented through the well-established concept of environment module files [14,15,21,24]. Although the logic encoded in these modules can be quite complex, end users are not exposed to this thanks to the simple interface provided by the `module` command. Except maybe through the large number of modules that are loaded by default when logging in to a Cray system; unlike other HPC systems, having at least 20 modules loaded at any given time is very common.

Even though the set of module files provided by the Cray PE is well structured and reasonable in size, it can be daunting to navigate this environment, for a number of reasons. First of all, the provided modules form a large combinatorial space since multiple versions are typically available for most of them. Since different (sets of) modules also interact with each other in complex ways, extensive manual loading, unloading and swapping of these modules sometimes leads to unexpected problems. Also, as the Cray PE release gets updated as often as monthly, additional modules will be added and older ones may be removed (depending on site policies). Cray does a good job of providing a sane, coordinated default set of modules, but users are on their own piecing together the components for a non-default release, and are expected to constantly re-evaluate which modules they employ, as sites typically recommend recompiling software after a new Cray PE release is rolled out.

In our experience, significant care must be taken when interacting with these modules. We found out the hard way that using `module purge` to start afresh does not work as expected, in the sense that it is close to impossible to manually reconstruct a properly working environment afterwards. Additionally, reloading modules that are already loaded may have unexpected side effects, so this should be avoided at all times.

3.1.2 The programming environment module `PrgEnv`

The most prominent set of modules that is provided, at least for the sake of this discussion, are the `PrgEnv` modules.

For each of the compiler suites supported by Cray, a corresponding `PrgEnv` module is available:

- `PrgEnv-cray`, for Cray Compiling Environment (CCE)
- `PrgEnv-gnu`, for GNU Compiler Collection (GCC)
- `PrgEnv-intel`, for Intel compiler suite
- `PrgEnv-pgi`, for PGI compiler suite

Each of these modules prepare the environment for using the respective compiler suite, by configuring the generic Cray compiler wrappers and loading additional modules that either provide additional software components or further tweak the environment as needed. For example, the `craype` module that corresponds to that particular version of the `PrgEnv` module gets loaded, which results in (re)defining a set of `$PE`-prefixed environment variables that influence the semantics of other modules being loaded afterwards.

Depending on the particular `PrgEnv` module being loaded, a module for the corresponding compiler suite will also be loaded. For example, loading `PrgEnv-gnu` will result in the Cray-provided `gcc` module being loaded as well; likewise for the other supported compiler suites. In addition, the `cray-libsci` module will also be loaded, which provides optimized numerical routines (BLAS, LAPACK, etc.), among others. Note that this is fairly similar to the EasyBuild toolchain mechanism explained in Section 2.2.4; however there are some key differences, see Section 3.2.3.

It is important to note here that although multiple versions of the different `PrgEnv` modules are available, the compiler and `cray-libsci` modules they load are always loaded without specifying a particular version, i.e., whichever version is the default version for those modules (as indicated by the site maintainers) will be loaded. As we will discuss in Section 3.3.4, this raises concerns w.r.t. reproducibility, since the default module versions tend to change over time as Cray PE updates are being made available.

3.1.3 Additional tools and libraries

Next to the basic programming environment, additional modules are available for libraries and 3rd party software. A prominent one is the `cray-mpich` module for the Cray MPT library providing support for distributed computations. Other examples include the `fftw` module that provides libraries with optimized FFT routines, and the `cray-hdf5` and `cray-netcdf` modules providing tuned versions of the established HDF5 and netCDF I/O libraries, respectively.

Not only do these modules make additional tools and libraries available, they also further configure the compiler wrappers provided via the `PrgEnv` module by defining various environment variables.

3.2 Integrating EasyBuild with the Cray PE

In order to integrate EasyBuild with the Cray PE, a number of issues needed to be dealt with. We briefly outline them in this section, before presenting how these were resolved in Section 3.3.

3.2.1 Leveraging the Cray PE modules

The most significant required feature was to enable EasyBuild to leverage the modules provided by the Cray PE. As mentioned in Section 2.3.2, EasyBuild not only generates module files for the installations being performed, it also heavily relies on those modules for resolving dependencies and preparing the build environment. The EasyBuild framework needed to be enhanced to allow for using the so-called 'external' modules provided by the Cray PE, see Section 3.3.1.

3.2.2 Better awareness of session environment

Some minor enhancements needed to be made to the mechanism for generating module files and the way in which EasyBuild interacts with the modules tool, in order to properly deal with the sensitive nature of the Cray-provided modules briefly discussed in Section 3.1.1.

This consists of taking into account already loaded modules and unloading/swapping conflicting modules if required, eliminating the use of `module purge`, and preventing Cray PE modules from being loaded again if they are already loaded.

These issues are discussed in Sections 3.3.2 through 3.3.4.

3.2.3 Version pinning of toolchain components

Traditional toolchains employed by EasyBuild have explicit versions specified for each of the toolchain components, exactly like dependencies of software packages. This stands in stark contrast with the closely related `PrgEnv` module files provided by the Cray PE, where the version of modules corresponding to what would be toolchain components (compiler, numerical libraries) are deliberately *not* specified (see also Section 3.1.2).

To adhere to the EasyBuild goal of enabling reproducible installations, this needed to be dealt with; this is covered in Section 3.3.4.

3.3 Enhancements to EasyBuild

In this section, we present the changes that were made to EasyBuild to deal with the concerns outlined in Section 3.2.

The development of these features started early 2015, leading to the experimental Cray support included in EasyBuild version 2.1.0 (Apr'15). Although progress was rather slow due to limited initial interest from Cray sites, the support from the CSC team in Finland, the feedback and contributions by CSCS staff members, and the testing of EasyBuild on Blue Waters at NCSA helped significantly in working towards stable support for using EasyBuild on top of the Cray PE.

The features discussed here are all part of the most recent EasyBuild release to date, version 2.7.0 (Mar'16). All together about 500 extra lines of code (LoC) were required to implemented these changes, on top of the existing codebase of about 50, 000 LoC. Hence, in hindsight the required effort in terms of implementation was very limited, illustrating the maturity of the EasyBuild framework.

3.3.1 Support for external modules

As mentioned earlier, EasyBuild relies on the module files it generated during subsequent installations, see Section 2.3.3. When resolving dependencies, EasyBuild will check whether the corresponding module files are already available, and

```
dependencies = [
    ('pandas', '0.17.1'),
    ('cray-hdf5/1.8.13', EXTERNAL_MODULE),
]
```

Listing 1: Example list of dependencies illustrating the use of external modules.

load them before initiating the installation; if not, it will search for easyconfig files that can be used to install the missing dependency. An important detail here is that the modules generated by EasyBuild contain information about the installed software they correspond to, that may be used during subsequent installations. In particular, the installation prefix and software version are provided through EasyBuild-specific environment variables, respectively named '`$EBROOT*`' and '`$EBVERSION*`'.

In order to let EasyBuild leverage modules that were provided some other way, for example as part of the Cray PE, the notion of '*external modules*' was introduced. Basically, this comes down to supporting that a dependency can be specified using only a module name and a marker indicating that it should be resolved directly via the specified module, as opposed to a standard dependency specification that includes the software name and software version, and optionally an additional label (a.k.a. 'version suffix') and the (sub)toolchain it should be installed with. Marking a dependency as an external module instructs EasyBuild that it should *not* attempt to search for an easyconfig file that can be used to install that dependency; if a module with the specified name is not available, it should simply report an error stating that the dependency could not be resolved.

The example list of dependencies extracted from a (fictional) easyconfig file shown in Listing 1 illustrates the difference between a standard dependency specification and one marked to be resolved by an external module. While the dependency on version 0.17.1 of the pandas Python package will be resolved the standard way, i.e., by letting EasyBuild determine the module name, checking whether the module is available and falling back to searching for an easyconfig file if needed, only the specified version of the `cray-hdf5` module will be considered to resolve the dependency on the HDF5 library.

Additionally, support was added to supply metadata for external modules, so that EasyBuild can be made aware of what they provide: the software name(s), version(s) and installation prefix corresponding to external modules of interest can be specified in one or more configuration files.

An example of metadata for external modules is shown in Listing 2. It specifies that:

- the `cray-hdf5/1.8.13` and `cray-hdf5-parallel/1.8.13` modules both provide HDF5 version 1.8.13 [1], and that the installation prefix can be obtained via the `$HDF5_DIR` environment variable that is defined by these modules;
- the `cray-netcdf/4.3.2` module provides both netCDF 4.3.2 and netCDF-Fortran version 4.3.2, and their common installation prefix can be determine via `$NETCDF_DIR`;

[1]EasyBuild makes the distinction between serial and parallel builds of HDF5 differently, i.e., through compiler toolchain that was used and whether or not it includes MPI support.

```
[cray-hdf5/1.8.13]
name = HDF5
version = 1.8.13
prefix = HDF5_DIR

[cray-hdf5-parallel/1.8.13]
name = HDF5
version = 1.8.13
prefix = HDF5_DIR

[cray-netcdf/4.3.2]
name = netCDF, netCDF-Fortran
version = 4.3.2, 4.3.2
prefix = NETCDF_DIR

[fftw/3.3.4.3]
name = FFTW
version = 3.3.4.3
prefix = FFTW_INC/..
```

Listing 2: Example metadata for external modules.

- the `fftw/3.3.4.3` module provides FFTW version 3.3.4.3, and the top-level installation directory is one level up from the path specified in `$FFTW_INC`, i.e. `$FFTW_INC` should be combined with the relative path '`..`'

The installation prefix can also be hardcoded with an absolute path, in case it can not be derived from an environment variable set by the external module.

Note that EasyBuild will not attempt to interpret the names of external modules in any way; it solely relies on the provided metadata. Also, EasyBuild will only interact with external modules via the modules tool, to load them and check for their availability; it will not try to parse or interpret their contents.

Since EasyBuild version 2.7.0, a file containing metadata for selected modules provided by the Cray PE is included. This is sensible only because the Cray-provided module files are known to be identical across Cray sites, if they are present. Moreover, it enables Cray sites to quickly get started with EasyBuild.

It is worth noting that the support in EasyBuild for using external modules is in no way specific to Cray systems. The feature was deliberately implemented in a generic way, in order to also make it useful on other platforms where the need for integrating with existing module files arises. For more details on using external modules, we refer to the EasyBuild documentation [2].

3.3.2 Generating more sophisticated module files

Motivated by the sometimes unpredictable behaviour of the modules files provided by the Cray PE, induced by the complex logic that is encoded in them, the mechanism in the EasyBuild framework for generating module files needed to be extended slightly.

Support for including '`module unload`' and '`module swap`' statements, optionally guarded by a condition that checks whether a particular module is loaded or not, had to be implemented. We explain the need for this in more detail in Section 3.3.4.

[2]`http://easybuild.readthedocs.org/en/latest/Using_external_modules.html`

3.3.3 Eliminating the use of `module purge`

EasyBuild controls the environment in which installations are being performed, in order to increase the likelihood of being able to reproduce it later. This was done by recording the environment in which EasyBuild is running, frequently purging the modules that were loaded by the EasyBuild toolchain mechanism (for dependencies, toolchain, etc.), and restoring the initial environment in the sanity check step and before initiating a new installation.

Since running '`module purge`' in an environment defined using Cray PE modules can not be done reliably, the purging of modules was replaced by simply restoring the initial environment instead, which is a better approach anyway of controlling the environment.

3.3.4 Cray-specific compiler toolchains

Defining and supporting Cray-specific compiler toolchains was another major enhancement that had to be made. This involved changes to the toolchain mechanism of the EasyBuild framework, and a custom easyblock to consume easyconfig files that specify particular instances of Cray-specific toolchains.

Overview of Cray-specific toolchains. In the current implementation, three different Cray-specific toolchains have been defined. Each of them corresponds directly to one of the `PrgEnv` modules we discussed in Section 3.1.2:

- `CrayCCE` for `PrgEnv-cray`
- `CrayGNU` for `PrgEnv-gnu`
- `CrayIntel` for `PrgEnv-intel`

The equivalent toolchain for the `PrgEnv-pgi` module was missing at the time of writing because the PGI compiler is not supported yet as a toolchain component in EasyBuild version 2.7.0.

In each of these, the toolchain consists of the respective compiler suite (CCE, GCC, Intel) together with the Cray MPT library (provided by the `cray-mpich` module) and Cray LibSci libraries (`cray-libsci`).

Framework support for Cray toolchains. The toolchain mechanism in the EasyBuild framework needed to be made aware of the different components that are part of the `Cray*` toolchains, in order to support the provided toolchain introspection functionality that is used in various easyblocks, and to properly define the build environment based on the selected toolchain.

For the compiler component of the toolchains, we leverage the compiler wrappers provided by the Cray PE. The compiler commands to are used by EasyBuild are identical regardless of which specific `Cray*` toolchain is used, i.e. `cc` (C), `CC` (C++) and `ftn` (Fortran). The readily available support for the GCC and Intel compilers was reused where relevant for the `CrayGNU` and `CrayIntel` toolchains, e.g., for the compiler flags to control floating-point precision. A limited set of toolchain options was defined to steer the compiler wrappers where needed, for example to disable dynamic linking which is enabled by default and is controlled by defining the `$CRAYPE_LINK_TYPE` environment variable, or to produce verbose output (which is useful for debugging).

For the Cray MPT library, the existing support for the MPICH MPI library was leveraged to inform the toolchain mechanism how to deal with the MPI component of the `Cray*` toolchains. Again, we leave the heavy lifting up to the CrayPE compiler wrappers.

Finally, for the Cray LibSci library a stub implementation of a library providing BLAS/LAPACK support was added. Just like for the compiler and MPI toolchain components, the EasyBuild framework relies mostly on the CrayPE compiler wrappers, other than making sure that the installation prefix of LibSci is obtained via the `$CRAY_LIBSCI_PREFIX_DIR` environment variable, so it can be used if required.

Custom easyblock for Cray toolchains. EasyBuild requires a module file to be available for every compiler toolchain to be used. For traditional toolchains, these typically only include straightforward '`module load`' statements for each toolchain component; the logic for actually using the toolchain is implemented in the toolchain support of the EasyBuild framework.

To obtain a module file for a particular version of a Cray-specific toolchain, a custom `CrayToolchain` easyblock was implemented that redefines part of the module generation step of the install procedure. This is required since the environment in which these toolchains will be employed is significantly more tedious to operate in than usual. Care must be taken that already loaded conflicting modules are unloaded, or that they are swapped with an equivalent module that is part of the toolchain being used. This comes back to the issues discussed in Section 3.1.1.

More specifically, the module file for a `Cray*` toolchain must:

- make sure that the correct `PrgEnv` module is loaded, after making sure that any other `PrgEnv` module is unloaded; it is worth noting that the seemingly straightforward option of swapping whatever `PrgEnv` module is loaded with the one that should be loaded was found to be unreliable
- load the specific module version for each toolchain component that is specified in the easyconfig file for the toolchain (see below), or swap whichever version is loaded with the required one

Note that this module is not only used by EasyBuild itself during the installation to prepare the build environment, but also whenever a module corresponding to a software installation is loaded by users, since these include a '`module load`' statement for the toolchain module that was used to install the software.

Definition of Cray toolchains. The actual definition of a particular version of a Cray-specific toolchain is expressed in an easyconfig file, just like for traditional toolchains.

An example is shown in Listing 3. The toolchain name and version are specified to be `CrayGNU` and `2015.11`, respectively. EasyBuild is instructed to process this easyconfig file using the `CrayToolchain` easyblock (see previous section), and some basic information is provided through the `homepage` and `description` parameters; this will be included in the generated module file to populate the output of `module help`.

While specifying a `toolchain` in each easyconfig file is strictly enforced by the EasyBuild framework, it is irrelevant in the case of installing a module file for a compiler toolchain, since only the step of the install procedure where the module is being generated is performed by the `CrayToolchain` easyblock; all other steps are skipped. Thus, a so-called 'dummy' toolchain is specified here.

The remainder of the easyconfig file defines the actual toolchain composition. First, the `PrgEnv-gnu` module is listed as the base of the compiler toolchain; the EasyBuild framework will verify whether this aligns correctly with the toolchain name. The specified module name is deliberately left versionless here, as was recommended to us by Cray support. The `PrgEnv` modules are intended to be mostly backwards compatible in terms of functionality, and the latest available version should be loaded at all times.

The subsequent entries of the `dependencies` list specify a particular version of a toolchain component. These versions collectively define this particular version of the `CrayGNU` compiler toolchain.

Note that the date-like format of the toolchain version, `2015.11` in this example, is strongly connected to the CrayPE release it was created for, indicated by the year and month. The actual versions of the toolchain components that define a particular toolchain version are selected based on the recommendations made by Cray for the corresponding CrayPE release. Our evaluation has shown that there is no need to discriminate between different types of Cray systems (e.g. XC vs XE/XK).

The intention of version pinning the toolchain components is to allow for reproducible software installations on top of the Cray PE. If the toolchain definitions can be decided upon in mutual consensus between different Cray sites, it provides a solid base for collaboration. This stands in stark contrast with today's lack of efficiently sharing expertise in a structured way on getting scientific software installed on Cray systems.

If desired, sites can define local variants of a toolchain by slightly modifying the versions of the toolchain components according to the availability of Cray PE modules. Such a custom toolchain can be tagged with an additional label (via the `versionsuffix` easyconfig parameter) to discriminiate it from the standard toolchain definiition.

Details of the Cray toolchain module file. To summarize, we now take a detailed look at the generated module file for a Cray-specific toolchain, and explain how it is able to reliably modify the complex environment defined by the Cray PE it gets loaded in.

Listing 4 shows a part of the `CrayGNU` module file that is generated by EasyBuild for the easyconfig file shown in Listing 3; other parts of the module file not relevant to the discussion here have been omitted.

First, the `PrgEnv-gnu` module that forms the base of the `CrayGNU` toolchain is dealt with. This is done through i) `unload` statements for all other possible `PrgEnv` modules that

```
easyblock = 'CrayToolchain'

name = 'CrayGNU'
version = '2015.11'

homepage = 'http://docs.cray.com/books/S-9407-1511'
description = """Toolchain for Cray compiler wrapper,
using PrgEnv-gnu see: PE release November 2015)."""

toolchain = {'name': 'dummy', 'version': 'dummy'}

dependencies = [
    ('PrgEnv-gnu', EXTERNAL_MODULE),
    ('gcc/4.9.3', EXTERNAL_MODULE),
    ('cray-libsci/13.2.0', EXTERNAL_MODULE),
    ('cray-mpich/7.2.6', EXTERNAL_MODULE),
]

moduleclass = 'toolchain'
```

Listing 3: Easyconfig file for CrayGNU version 2015.11

may be loaded (note that unloading is always safe to do, even if the specified module is not loaded); ii) a guarded (versionless) `load` statement for the `PrgEnv-gnu` module. The condition on the negation of '`is-loaded PrgEnv-gnu`' is required to ensure that the `PrgEnv-gnu` does not get loaded again, since that may result in unwanted side effects.

Note that when this first section of the module file has been processed when this toolchain module is being loaded, a particular version of some of the toolchain components will already be loaded since the `PrgEnv` module loads the default module for the compiler and `cray-libsci` modules (see also Section 3.1.2).

The remainder of Listing 4 shows the logic used to load the specified version of each of the toolchain components. Depending on whether a module for that particular toolchain component is already loaded or not, a `swap` or `load` operation will be performed to ensure that the specified version gets loaded.

3.3.5 *Cross-compilation and target architecture*

It is quite common on Cray systems that the Cray development and login (CDL) nodes have a different processor architecture than the actual compute nodes of the system, and even that different partitions exist in the system that differ in processor architecture. If so, software compilation should be done via cross-compilation for a different target architecture, to achieve optimal performance. The Cray PE provides a set of `craype-<arch>` modules for this purpose, which configure the compiler wrappers provided via the `PrgEnv` module to generate binary code for a particular architecture.

This is strongly related to the `optarch` configuration option that is available in EasyBuild, which allows for specifying which compiler flags should be used that control the target architecture.

For Cray toolchains specifically, we redefined the meaning of `optarch` to indicate which `craype-<arch>` module must be loaded in the build environment. That is, if `optarch` is defined to be '`haswell`', then EasyBuild will make sure the `craype-haswell` module is loaded; if it is not (potentially because another `craype-<arch>` module is loaded), it will exit with an error. A possible enhancement to this may be to automatically swap to the correct `craype-<arch>` module

```
#%Module

module unload PrgEnv-cray
module unload PrgEnv-intel
module unload PrgEnv-pgi

if { ![ is-loaded PrgEnv-gnu ] } {
    module load PrgEnv-gnu
}

if { [ is-loaded gcc ] } {
    module swap gcc gcc/4.9.3
} else {
    module load gcc/4.9.3
}

if { [ is-loaded cray-libsci ] } {
    module swap cray-libsci cray-libsci/13.2.0
} else {
    module load cray-libsci/13.2.0
}

if { [ is-loaded cray-mpich ] } {
    module swap cray-mpich cray-mpich/7.2.6
} else {
    module load cray-mpich/7.2.6
}
```

Listing 4: Partial generated module file for version `2015.11` of the `CrayGNU` toolchain

instead.

Also, in order to use a Cray toolchain `optarch` *must* be specified, while with other toolchains it is an optional setting that can be used to override of the (usually sensible) default of targeting the host architecture. If it is not specified, EasyBuild will currently exit with a clear error message explaining that it should be defined and correspond to an available `craype-<arch>` module.

3.4 Testing & evaluation

The support in EasyBuild version 2.7.0 for integrating with the Cray PE has been tested and evaluated on various Cray systems, using multiple established scientific applications. This section provides a short overview.

3.4.1 Systems

Piz Daint & co – CSCS, Switzerland. See Section 5 for a detailed discussion on evaluating EasyBuild on Cray (and non-Cray) systems at CSCS.

Sisu – CSC, Finland. Sisu is a 1,688 compute node Cray XC40 series system hosted at the IT Center for Science (CSC) in Finland; more information is available at `https://research.csc.fi/csc-s-servers`.

Swan – Cray. Swan is a Cray XC40 series system hosted at Cray. Swan is an early access test bed system for Cray technologies. As such it is comprised of compute nodes using the Intel Broadwell, Intel Haswell and Intel Ivybridge processors next to hybrid nodes equipped with Nvidia K20X GPUs at time of writing.

Blue Waters – NCSA, US. Blue Waters is the largest Cray XE6/XK7 class system ever built with 26,864 compute nodes. 4,228 of the nodes are XK7 nodes which have half the memory and CPUs in exchange for an NVIDIA K20X GPU accelerator. Blue Waters serves a very broad range of high-end computational science as awarded by the National Science Foundation in the U.S, and primarily uses AMD Interlagos processors on both the compute and external service nodes, along with older AMD processors on the service nodes and test system login node. More information about Blue Waters is available at `https://bluewaters.ncsa.illinois.edu/`.

Titan – ORNL, US. Titan is a Cray XK7 system operated by the Oak Ridge Leadership Computing Facility for the United State Department of Energy. Titan has 18,688 compute nodes each containing an AMD Interlagos processor and an NVIDIA K20X GPU accelerator; see `https://www.olcf.ornl.gov/titan/` for more information.

3.4.2 Software applications

HPL. The HPL LINPACK benchmark version 2.1 served as the initial test case for the Cray support in EasyBuild, and was installed using two different versions of the currently supported Cray-specific toolchains, i.e. the `2015.06` and `2015.11` versions of the `CrayCCE`, `CrayGNU` and `CrayIntel` toolchains.

CP2K. The quantum chemistry and solid state physics software package CP2K was installed using the `CrayGNU` toolchain. The multi-threaded (`popt`) variant of CP2K version 2.6.0 was tested, as well as the distributed (`psmp`) variant of CP2K version 3.0. For the FFTW dependency the Cray-provided `fftw` module was used, while the other dependencies (`Libint`, `libxc`) were resolved via EasyBuild.

GROMACS. Installing the distributed variant of version 4.6.7 of the popular molecular dynamics software GROMACS was tested using the `CrayGNU` and `CrayIntel` toolchains.

WRF. Version 3.6.1 of the Weather Research and Forecasting (WRF) Model was installed using both the `CrayGNU` and `CrayIntel` toolchains, on top of the `cray-netcdf` and `cray-hdf5-parallel` modules provided by the Cray PE. Even though the installation procedure of WRF is highly custom, no changes were required to the WRF-specific easyblock that was already avaialble to install it on top of the Cray PE.

Python. Various recent versions of Python together with popular Python packages were installed using the different versions of the `CrayGNU` toolchain; we discuss the Python use case in detail in Section 4.

DCA++ dependencies. DCA++ is Dynamical Cluster Approximation software implemented in C++ to solve the 2D Hubbard model for high-temperature superconductivity. The developers used EasyBuild to install the dependencies required for DCA++ that are not provided by the Cray PE, on different Cray systems and using the `CrayGNU` toolchain. More specifically, easyconfig files were composed for the NFFT, spglib and MAGMA libraries; the generic `ConfigureMake` easyblock turned out to be sufficient for each of them. Where applicable, Cray-provided modules are leveraged, i.e., `cudatoolkit` for MAGMA, `fftw` for NFFT and `cray-hdf5` to resolve the HDF5 requirement of DCA++ itself.

It is worth noting that the easyconfig files were composed and tested on Piz Daint (CSCS), and then used to trivially reproduce the installations on both Titan (ORNL) and Sisu (CSC), clearly highlighting the potential value of employing EasyBuild on Cray systems.

4. USE CASE: PYTHON ON CRAY

In this section, we discuss the use case of providing an optimized installation of the standard Python distribution and additional Python packages on Cray systems, on top of the Cray-provided software stack.

4.1 Motivation

The Python programming language and platform is becoming increasingly more popular in traditional scientific domains. As a result of this, the expectation of today's scientific researchers is that a fully featured and heavily optimized Python installation is made available on modern HPC systems.

However, at the time of writing Cray does not yet provide such a high performance 'batteries included' Python distribution as a part of the provided software stack, despite the increasing demand for it. The only Python environment that is readily available on the recent versions of Cray Linux Environment (CLE) is the one provided through the SuSE Linux Enterprise Server (SLES) distribution. Although it is mostly functional, it does not leverage the heavily tuned software stack provided by Cray PE, nor was it tuned to the underlying Cray system architecture.

As a consequence, Cray sites are required to take care of an optimized Python installation themselves. Next to installing the standard Python distribution itself in a fully featured way, additional Python packages for scientific computing need to be included. This typically includes the heavily tuned SciPy stack consisting of popular libraries like `numpy`, `scipy`, `matplotlib`, `pandas`, and `sympy`, as well as Python packages leveraging these base libraries, like `scikit-learn` for machine learning and the Python Imaging Library (`PIL`). In addition, Python interfaces to established native libraries like MPI (`mpi4py`), HDF5 (`h5py`) and netCDF (`netcdf4-python`), domain-specific packages like BioPython and powerful interactive tools like IPython are frequently requested.

4.2 Python support in EasyBuild

EasyBuild includes extensive support for installing Python and Python packages out-of-the-box.

The (non-trivial) installation procedure of the standard Python distribution is encoded in a custom software-specific easyblock. It takes care of correctly configuring the build process, mostly related to picking up on optional dependencies like libreadline & ncurses, Tcl & Tk, OpenSSL, etc., and carefully checks whether the obtained Python installation is fully featured as intended.

For installing Python packages, a generic `PythonPackage` easyblock is available that implements the standard `setup.py` based installation procedure, and performs a trivial sanity check by testing whether the installed Python package can be imported. This generic easyblock can be leveraged to both install Python packages as a part of a Python installation, yielding a so called 'batteries included' installation, and to install stand-alone modules for (bundles of) Python packages; see also Section 4.4.

In addition, a number of software-specific easyblocks for individual Python packages are also available. Particular examples are the easyblocks for NumPy and SciPy, which implement the tedious install procedure for these respective Python packages, on top of the heavily tuned BLAS and LAPACK libraries provided by the compiler toolchain. They also include various checks to verify that the installation was indeed performed correctly and optimally, like running the provided test suites and performing a simple speed test for the `numpy.dot` routine by computing the dot product of two random 1000x1000 matrices, which is usually sufficient to detect misconfigurations since the performance difference between a basic and an optimized NumPy installation differs by over an order of magnitude.

Several Python-related easyconfig files are available: for 'bare' installations containing only the standard Python distribution, installations featuring a list of Python packages that should be installed as extensions, and for individual Python packages or bundles thereof.

4.3 Installing Python on top of the Cray PE

To install Python in an optimized way on top of the Cray PE, the existing EasyBuild support for both Python itself and Python packages could be easily combined with the Cray-specific compiler toolchains presented in Section 3.3.4. The relevant easyblocks were already sufficiently generic, such that no changes needed to be made to make the installations work.

It was sufficient to create the necessary easyconfig files that specify to EasyBuild the exact versions of Python itself, its dependencies and those of the Python packages to be included, and which Cray-specific toolchain should be used to install them. Except for specific Python packages like `h5py` where the HDF5 installation provided by the `cray-hdf5-parallel` module can be leveraged, all dependencies other than the toolchain components are resolved through EasyBuild.

Figure 4 shows the dependency graph for h5py 2.5.0 and Python 2.7.11 on top of version `2015.11` of the `CrayGNU` toolchain (versions of the dependencies not relevant to the discussion were omitted). Installations performed using EasyBuild are depicted using ovals, dependencies resolved via the 'external' modules provided by the Cray PE are shown as rectangles and marked with '`EXT`'; additional Python packages that are included as extensions in the Python installation are indicated using dashed lines.

The entire software stack for h5py and Python shown in Figure 4, including required dependencies and specified extension Python packages, can be installed on top of the Cray PE with a single EasyBuild command, assuming that EasyBuild was installed and configured properly and that the necessary easyconfig files are available:

```
$ eb h5py-2.5.0-CrayGNU-2015.11-Python-2.7.11.eb -r
```

It suffices to specify the name of the easyconfig file to the `eb` command, and enable the dependency resolution mechanism using the `-r` command line option. After verifying that

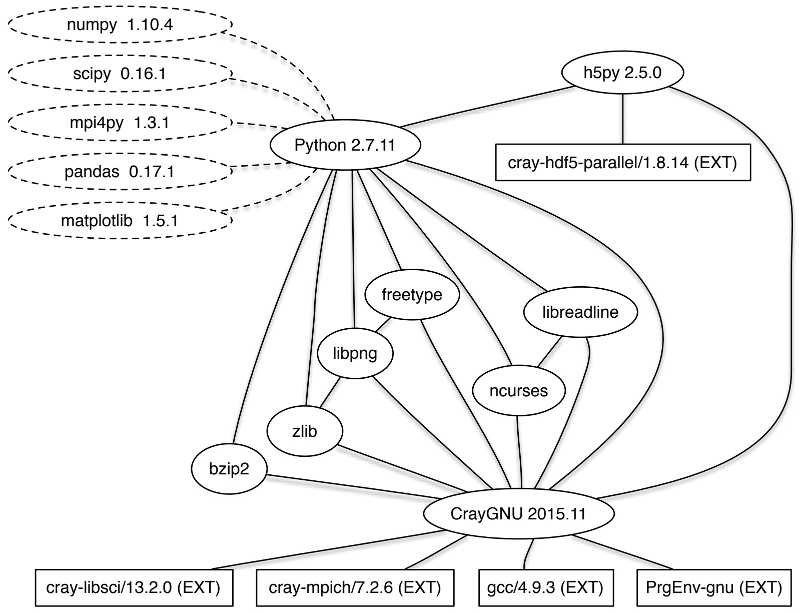

Figure 4: Complete dependency graph for Python with included extensions, and **h5py** on top of it.

the easyconfig files required to construct the entire dependency graph are available, EasyBuild will install any missing dependencies and generate modules for them, and subsequently install Python and the specified extensions; afterwards, h5py itself will be installed on top of the obtained Python installation.

This is done fully autonomously and leveraging the relevant parts of the Cray PE, resulting in the intended 'batteries included' Python installation that is optimized for the architecture of the target Cray system.

4.4 Managing Python packages

A number of different options are supported by EasyBuild to manage Python packages.

As already mentioned, the Python installation can be done with 'batteries included', meaning that additional Python packages can be installed in the same installation prefix as Python. If updates for particular Python packages become available later, additional stand-alone modules on top of the Python module can be installed that overrule their outdated equivalent that is part of the Python installation.

These stand-alone module can be either for individual Python packages, or bundles of Python packages that are somehow related to each other.

Additionally, users can manage their own Python packages on top of the centrally provided Python installation, if they so desire. For this, the standard Python installation tools like `pip` can be employed.

5. EASYBUILD AT CSCS

Founded in 1991, the Swiss National Supercomputing Centre (CSCS) develops and provides large scale supercomputing infrastructure to the international research community. The CSCS infrastructure consists of both standard Linux HPC clusters and different generations of Cray systems, including the recent Cray XC30 system *Piz Daint*, which is currently ranked 7th on the Top500 list (Nov'15).

Until recently, installing scientific software on the diverse pool of HPC systems hosted at CSCS was done with little or no automation. Instead, the support team mostly resorted to manual installations, and the expertise on installing complex applications was scattered across various individuals, personal scripts, support tickets, and non-centralized notes. By consequence, updating software installations was tedious and error-prone and a huge burden on the support team, and the software stack provided to end users was wildly inconsistent across the different systems and even within the same system.

Motivated by the need to improve the consistency of the user experience when switching between various systems and reducing the efforts involved in providing scientific applications, EasyBuild was evaluated as a potential platform for fulfilling these aspirations.

CSCS began evaluating EasyBuild with release of version 2.1 (Apr'15), when the first experimental support for Cray PE integration was included, by providing an optimized fully featured installation of Python on various systems hosted at CSCS. The experience was very positive, as Python and a set of additional Python packages were provided to end users in a matter of days, across both Cray and non-Cray systems in a consistent way and with limited effort. This, and the welcoming and supportive EasyBuild community, convinced the CSCS support team to integrate it into their daily workflow, see below.

5.1 CSCS systems using EasyBuild

Today, EasyBuild is deployed on a center-wide central shared file system (GPFS) on most of the production and test systems at CSCS, including:

- Piz Daint: Cray XC30 with $5,272$ compute nodes (Intel Sandy Bridge + NVIDIA Tesla K20X)
- Piz Dora: Cray XC40 with $1,256$ compute nodes (Intel Haswell); extension to Piz Daint

- Santis & Brisi: Test and Development Systems (TDS) for Piz Daint and Piz Dora
- Pilatus: Linux HPC cluster with 44 compute nodes (Intel Sandy Bridge-EP)
- Mönch: Linux HPC cluster with 440 compute nodes (Intel Ivy Bridge)
- Kesch & Escha: two Cray CS-Storm cabinets operated for MeteoSwiss (Intel Haswell + NVIDIA Tesla K80); see also Section 5.3

For more details, we refer to `http://www.cscs.ch/computers`.

5.2 Consistency for users and support team

EasyBuild has helped significantly in providing more consistency for both users and the CSCS support team, across various HPC systems hosted at CSCS.

It serves as a uniform interface for maintaining the software stack on both Cray and non-Cray systems, and allows maintaining a central repository where installation recipes (easyconfig files and customized easyblocks) are hosted. This approach could also be used to enable sharing of expertise w.r.t. installing scientific software across Cray sites.

On most Cray systems the Cray-specific toolchains discussed in Section 3.3.4 are used (one exception being the CS-Storm system, see Section 5.3); on the Linux HPC clusters, 'regular' compiler toolchains included in EasyBuild like `foss` and `intel` are employed.

Providing software installations through EasyBuild and reusing compiler toolchains where applicable (e.g. `CrayGNU` on Piz Daint and Piz Dora) results in a consistent software stack being provided across the different systems, which is more appealing to end users.

Although the CSCS support team only provides full support for a select set of applications, EasyBuild has enabled to provide additional limited support to users for installing other software as well, by providing easyconfig files and letting users maintain their own software stack on top of what is centrally provided through EasyBuild.

5.3 Use case: EasyBuild on Cray CS-Storm

MeteoSwiss, the Swiss national weather forecasting service, hosts their dedicated production systems at CSCS. For their most recent platform Cray CS-Storm systems were deployed, i.e. Kescha & Escha [11], with CSCS being responsible for getting the system production-ready and for provisioning of the software stack.

The software environment on the CS-Storm system is different to the XC and XE/XK line of systems, in the sense that for the particular deployment only `PrgEnv-cray` based part of the Cray PE release is supported. Therefore, CSCS opted to install their own software stack from scratch. However, when building the software for MeteoSwiss [25] on this system, it became clear that the provided GCC-based compiler stack was unable to assemble optimized (AVX2) instructions for system's Intel Haswell processors.

While this issue was being discussed with Cray support, it became clear that most of the software required by MeteoSwiss was already supported in EasyBuild. Moreover, it turned out that the problem with the Haswell support of the standard compiler stack was already solved in the recent compiler toolchains defined by the EasyBuild community, by including a more recent version of GNU binutils, the set of tools that includes the GNU assembler `as` employed by GCC.

As a result, the support ticket opened with Cray was closed, and the software stack required by MeteoSwiss was provided via EasyBuild within a matter of days. This not only resulted in a more efficient workflow and a solution to compiling the required software stack optimized for the target architecture, it also further aligned the workflow for software installations with other CSCS systems where EasyBuild was being used.

5.4 Testing via continuous integration

During the life cycle of an HPC system, updates to the software environment are inevitable. Cray PE updates appear as frequently as every month. The accumulated set of modules that were installed as a part of various Cray PE releases may quickly result in about 1000 modules.

Qualifying software and hardware updates on dedicated test and development systems is considered good practice and should happen as frequently as possible, and ideally as an continuous process. In order to accomplish this, all involved steps of the process should be designed to run fully automated. A recent comparison of existing continuous integration and deployment tools that were evaluated within the context of performance monitoring is available in [23].

CSCS decided on using the Jenkins project as the platform of choice for implementing continuous validation techniques within the scope of testing applications delivered to end users through EasyBuild.

Jenkins is oriented around projects that define a set of instructions that are automatically executed upon user defined triggers like commits to central source code repositories. The project defined by CSCS for continuously testing installations takes a list of easyconfigs and performs the installations on selected systems. The Jenkins dashboard allows every stakeholder to quickly understand the capacity of the entire software stack to be reinstalled on every system the center operates. In addition to being the central source of information about the current state of the deployed software stack, provenance and history on the information about previous failures or success for builds can easily be stored and accessed upon interest.

Another benefit of employing continuous and immediate verification of every piece of software supported by the team is the feedback loop introduced into the change processes, and the increased capability to pinpoint sources of problems more quickly and with higher confidence.

The Jenkins project was designed to mimic the user experience for installing applications on production systems, providing an additional bonus of being able to understand the experience perceived by the end user.

The approach outlined in this section enables a completely new approach to testing and deployment of requested applications: the user support team prepares the easyconfig files and provides them to Jenkins, which verifies the installations before they are deployed.

6. RELATED WORK

Over the recent years, various tools similar to EasyBuild have been made available. We briefly discuss the most prevalent ones here, highlighting the similarities and key differences with EasyBuild.

6.1 Spack

The most relevant tool in the context of this discussion is definitely *Spack* [16, 27], a package management tool created at Lawrence Livermore National Laboratory (LLNL). In terms of design and functionality, it is quite similar to EasyBuild: it is implemented in Python, has the notion of 'packages' which are basically equivalent to a combination of easyblocks and easyconfig files, and was created to deal with the daily struggle of dealing with software installations on HPC systems. Like EasyBuild, a community has emerged around it that is actively contributing to the project.

There are a couple of key differences however. The most striking difference is the flexible support for specifying the software dependency graph. Spack consumes a so-called abstract spec that is a partial specification of what should be installed, which is then transformed in a concrete spec (basically through constraint solving) before the actual installation is performed. This allows for easily navigating the combinatorial space of dependency versions and build variants, and appeals to the main target audience of Spack: software developers of large scientific applications that need to juggle with lots of dependencies. As a result of this, Spack does not have a concept like compiler toolchains. In comparison, EasyBuild is more focused on making software installations easy to reproduce across HPC sites, and mainly targets HPC user support teams that need to provide a consistent software stack for scientists.

The Spack development team was actively working on integration with the Cray PE at the time of writing, in collaboration with the National Energy Research Scientific Computing Center (NERSC).

6.2 Other installation tools for HPC systems

SWTools [22, 26] and its successor Smithy [13] were developed by Oak Ridge National Laboratoy (ORNL). Smithy is implemented in Ruby, follows the Homebrew style of 'formulas' defining software installation procedures, and provides support to leverage the Cray PE. From a design point of view, Smithy is less of a framework compared to EasyBuild or Spack; there is little code reuse across different formulas, which also include a hardcoded template for module files. To the best of our knowledge, Smithy is only used by the HPC support team at ORNL.

HeLmod (Harvard Extensions for Lmod deployment) [28] is a software management system developed by FAS Research Computing (FASRC) department at Harvard University. It is built on top of Lmod, a modern alternative to the traditional environment modules tool, and consists of a collection of RPM `spec` files that implement software installation procedures and include the accompanying module file. As such, it is prone to lots of code duplication, as opposed to EasyBuild that takes a more centralized approach where code reuse is actively pursued. To the best of our knowledge, it does not provide any integration with the Cray PE.

Maali [8] (previously iVEC Build System or iBS) is a software build system developed by the Pawsey Supercomputing Centre. It is implemented in the Bash scripting language, and provides basic support for automatically installing software packages. Although it includes support for installing software on Cray systems, the functionality it provides is fairly basic – for example, there is not support (yet) for resolving dependencies, and it provides a limited flexibility – a lot of the configuration aspects are hardcoded in it.

6.3 More generic software installation tools

Next to software installation framework and tools targeted towards HPC systems, there are also several other projects worth mentioning in this context.

Two recent similar projects are Nix [5, 12] and GNU Guix [3,9], both so-called functional package managers. They take a radically different approach to building and installing software, and have reproducible builds as their main focus, to the point where they strive for making builds bit-wise reproducible. Some noteworthy features include performing builds in an isolated environment, transactional upgrades, per-user profiles and portability of packages across different operating systems. Recent work [10] promotes the use of these tools on HPC systems; however, there is a lack of integration with established practices like environment modules and stringent requirements to ensure true isolation (due to reliance on `chroot` and requiring a running daemon that performs the actual build), leading to limited adoption in the HPC community thus far.

The Hashdist environment management system [4] is a software build and installation tool to manage the installation of large software projects on Linux, OS X and Windows; it used by the FEniCS project, for example. It provides support for defining software profiles in YAML syntax.

Portage [7] is the official package management and distribution system for the Gentoo Linux distribution. pkgsrc [6] is a framework for building third-party software on NetBSD and other UNIX-like systems. Both of these tools distinguish themselves from other Linux distribution package managers by their focus on building from source.

Conda [1] is a package/environment management system for Linux, OS X and Windows. It was originally created for Python, but now supports other software as well. The Anaconda project leverages conda and provides an extensive Python distribution that includes over 150 scientific Python packages. Although it has some support for leveraging optimized libraries like Intel MKL, it's primary focus is easy of use rather than highly optimized software installations.

To the best of our knowledge, none of these projects provide integration with the Cray PE.

7. CONCLUSIONS

We motivated the need for automating the process of installing (scientific) software on HPC systems, and on Cray systems in particular, with the requirement of being able to easily reproduce software installations later or on different systems in mind.

EasyBuild was presented as a software installation framework that has been successfully deployed on HPC sites around the world for this purpose. After introducing EasyBuild, we discussed how it was successfully integrated with limited effort with the Cray Programming Environment, through the support for external modules and defining Cray-specific compiler toolchains.

The use case of providing an optimized 'batteries included' Python installation on Cray systems using EasyBuild was discussed in detail, and we outlined how EasyBuild has been deployed at CSCS to efficiently deal with providing software installations in a consistent way across the large amount and variety of (both Cray and non-Cray) systems.

We hope this work is the start of a revolution in how scientific software is installed on top of the Cray-provided software stack, and that it forms the base for a collaboration across Cray sites to benefit from each others work and expertise in providing users with the scientific software stack they require.

8. ACKNOWLEDGEMENTS

We would like to thank Cray, the IT Center for Science (CSC) in Finland – in particular Olli-Pekka Lehto, and the the Swiss National Supercomputing Centre (CSCS) for providing us with exploratory access to their Cray systems, and for their valuable feedback on our work.

The development of EasyBuild is supported by Ghent University, the Flemish Supercomputer Centre (VSC), the Flemish Research Foundation (FWO) and the Flemish Government, department EWI.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

9. REFERENCES

[1] Conda. http://conda.pydata.org/docs/.
[2] Easybuild documentation. http://easybuild.readthedocs.org.
[3] Gnu guix package manager. https://www.gnu.org/software/guix/.
[4] Hashdist – build it once. https://hashdist.github.io/.
[5] Nix – the purely functional package manager. https://nixos.org/nix/.
[6] pkgsrc – portable package build system. https://www.pkgsrc.org/.
[7] Portage. https://wiki.gentoo.org/wiki/Portage.
[8] P. S. Centre. Maali – pawsey supercomputing centre build system. https://github.com/Pawseyops/maali.
[9] L. Courtès. Functional package management with guix. *CoRR*, abs/1305.4584, 2013.
[10] L. Courtès and R. Wurmus. Reproducible and user-controlled software environments in hpc with guix. In *Euro-Par 2015: Parallel Processing Workshops*, pages 579–591. Springer, 2015.
[11] CSCS. Piz kesch cs-storm system description. http://www.cscs.ch/computers/kesch_escha/index.html.
[12] A. Devresse, F. Delalondre, and F. Schürmann. Nix based fully automated workflows and ecosystem to guarantee scientific result reproducibility across software environments and systems. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pages 25–31. ACM, 2015.
[13] A. DiGirolamo. The smithy software installation tool, 2012. http://anthonydigirolamo.github.io/smithy/.
[14] J. L. Furlani. Providing a Flexible User Environment. In *Proceeding of the Fifth Large Installation System Administration (LISA V*, pages 141–152, 1991.
[15] J. L. Furlani and P. W. Osel. Abstract yourself with Modules. In *Proceeding of the Tenth Large Installation System Administration (LISA '96*, pages 193–204, 1996.
[16] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The spack package manager: Bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC*, volume 15, 2015.
[17] M. Geimer, K. Hoste, and R. McLay. Modern scientific software management using easybuild and lmod. In *Proceedings of the First International Workshop on HPC User Support Tools*, pages 41–51. IEEE Press, 2014.
[18] K. Hoste, J. Timmerman, A. Georges, and S. De Weirdt. Easybuild: Building software with ease. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 572–582. IEEE, 2012.
[19] HPC-UGent. Easybuild – building software with ease. http://hpcugent.github.io/easybuild/.
[20] C. Inc. Cray programming environment user's guide. http://docs.cray.com/books/S-2529-116//S-2529-116.pdf.
[21] Environment Modules Project. http://modules.sourceforge.net.
[22] N. Jones and M. R. Fahey. Design, Implementation, and Experiences of Third-Party Software Administration at the ORNL NCCS. In *Proceedings of the 50th Cray User Group (CUG08)*, 2008.
[23] V. G. V. Larrea, W. Joubert, and C. Fuson. Use of continuous integration tools for application performance monitoring. In *Proceedings of the 57th Cray User Group (CUG15)*, 2015.
[24] J. Layton. Environment Modules – A Great Tool for Clusters. Admin HPC. http://www.admin-magazine.com/HPC/Articles/Environment-Module.
[25] MeteoSwiss. The new weather forecasting model for the alpine region. http://www.meteoswiss.admin.ch/home/latest-news/news.subpage.html/en/data/news/2016/3/the-new-weather-forecasting-model-for-the-alpine-region.html.
[26] SWTools, 2011. https://www.olcf.ornl.gov/center-projects/swtools.
[27] Todd Gamblin. Spack. http://scalability-llnl.github.io/spack/.
[28] F. R. C. H. University. Harvard extensions for lmod deployment (helmod), 2013. https://github.com/fasrc/helmod.