Modern Scientific Software Management Using EasyBuild and Lmod

Markus Geimer
Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich GmbH
52425 Jülich, Germany
m.geimer@fz-juelich.de

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

Robert McLay
Texas Advanced Computing Center (TACC)
University of Texas
10100 Burnet Rd
Austin (TX) 78758, USA
mclay@tacc.utexas.edu

***Abstract*—HPC user support teams invest a lot of time and effort in installing scientific software for their users. A well-established practice is providing environment modules to make it easy for users to set up their working environment. Several problems remain, however: user support teams lack appropriate tools to manage a scientific software stack easily and consistently, and users still struggle to set up their working environment correctly. In this paper, we present a modern approach to installing (scientific) software that provides a solution to these common issues. We show how EasyBuild, a software build and installation framework, can be used to automatically install software and generate environment modules. By using a hierarchical module naming scheme to offer environment modules to users in a more structured way, and providing Lmod, a modern tool for working with environment modules, we help typical users avoid common mistakes while giving power users the flexibility they demand.**

I. Introduction

In a typical desktop environment, it is usually sufficient to have a single version of a software package installed to fulfill a particular purpose. HPC systems, on the other hand, are normally used by a large user community with widely varying demands. In particular, there is often the need to make multiple versions of software packages available. These sometimes include competing packages that provide either identical or significantly overlapping functionality. Examples include different implementations of the MPI standard (e.g., Open MPI vs. MVAPICH2) and linear algebra packages (e.g., OpenBLAS vs. Intel MKL).

A simple yet powerful solution to this issue are environment modules [1]–[4], which allow users to easily load, unload, and switch between software packages by modifying the user's environment. This is done by adjusting environment variables like `$PATH` and/or setting additional package-specific variables, for example, to specify a license server. However, while environment modules are used by many HPC sites around the world this approach is not without its challenges, a common one being managing large numbers of modules in a way that allows individual users to easily control their working environment.

Providing users an easy way to access the (scientific) software available on an HPC system is certainly a challenge, but installing large collections of scientific software packages is a non-trivial task in its own right. These packages are often written by domain scientists who are comfortable only with their own hardware and software environment; the developers are often not interested in providing a robust build and installation procedure based on portable build tools [5]. As a result, system administrators of different HPC sites may find they have to reinvent the wheel to get a particular software package installed on their local system. Moreover, this kind of hard-won experience typically is not shared between HPC sites. In addition, the required modifications and the exact installation steps are often poorly—if at all—documented, which significantly impedes maintainability and reproducibility of software installations.

In this paper, we address these issues by introducing an automated approach to installing scientific software and organizing the corresponding modules in a hierarchical way. This is achieved by leveraging the functionality provided by the two community-driven tools *EasyBuild* [6] and *Lmod* [7]. EasyBuild provides a framework for automating software installations with a particular focus on scientific software packages—with the intention to collect and share the knowledge that is currently distributed in the HPC community. Lmod is a significantly enhanced but (largely) backward-compatible implementation of environment modules including specific features targeting a hierarchical module organization.

The remainder of this paper is structured as follows. In Section II, we describe the traditional approach to installing scientific software stacks on HPC systems, highlighting the common problems experienced by user support staff and end users. Section III proposes hierarchical module naming schemes as a promising alternative, and outlines the associated implications and issues. In Sections IV and V, we argue that EasyBuild and Lmod are well suited for dealing with hierarchical module naming schemes, and describe how these tools enhance both the user experience and the efficiency of user support teams. The community-oriented aspect of both tools, and the synergy between them, is the subject of Section VI. We discuss future and related work in Sections VII and VIII respectively, and conclude in Section IX.

II. Traditional approach

In this section, we describe common approaches that are traditionally used at HPC sites to install and manage scientific software over the lifetime of an HPC system. We start by describing traditional module tools used by many sites, then introduce the concept of module files and various module naming schemes in common use today. Next, we review typical workflows for installing scientific software. Finally, we highlight the lack of collaboration among HPC sites with regard to these topics.

A. *Providing access to software using environment modules*

For software not installed in standard system locations, a user's `$PATH` environment variable is often modified to include the install location of the binaries it provides, in order to make them easily accessible. Other environment variables may also need to be changed: `$LD_LIBRARY_PATH` for libraries required at runtime, `$CPATH` for paths to include directories, etc. One approach is to provide a shell script for each software package that users can source to modify their working environment, for each of the supported shells.

The environment modules system extends this technique by managing this sourcing step "under the hood." This approach has several major advantages. The first is that all users configure their environment in the same way; there is no need for a separate approach for each type of shell. There are several popular implementations all sharing a similar user interface based on a command named `module`:

```
% module [options] <subcmd>
```

Typical subcommands (among many) include `load`, `unload`, `list` (to list all loaded modules), and `avail` (to print the modules that are currently available for loading).

To access a particular software package named '*foo*', the user simply 'loads' the corresponding module file:

```
% module load foo
```

The second major advantage is that users can *unload* a previously loaded module, to undo changes in the environment and restore their environment to the one they had before they loaded a particular module. This means that users maintain control of their working environment as they switch between different versions of the applications, libraries, compilers, or MPI stacks. These features explain why the environment module system has become ubiquitous on HPC systems since the late 1990s.

In all implementations, the `module` command is implemented as a simple shell function (for Bourne-compatible shells) or alias (for csh-compatible shells) which evaluates the commands printed to standard output by a helper tool (e.g., `modulecmd`). This helper tool does the heavy lifting: identifying the specified subcommand, locating and parsing the corresponding module files, and generating the commands necessary to modify the user's environment.

Over time, multiple implementations of the helper tool have been developed. The original implementation [1] was a collection of shell scripts. Today, the most commonly used version is written in C, using the Tcl scripting language to parse and evaluate module files [8]. A second implementation, never packaged as a release and still marked as experimental by the authors, is implemented using only Tcl [8]. For the most part, these two implementations offer identical functionality, but subtle differences sometimes lead to surprises when switching between them. A third product, a fork of the Tcl-only implementation, has been heavily adjusted to meet the requirements of the DEISA (Distributed European Infrastructure for Supercomputing Applications) project [9]. In 1997, a C-only implementation named *cmod* [10] made its debut, but this product has not been updated since 1998.

While these implementations provide the desired basic functionality, support has been uneven at best. In all cases, development progresses slowly. For example, there has been no activity in (the publicly accessible version of) the Tcl-only implementation for about two years. It is therefore reasonable to assume that new features (e.g., improved support for hierarchical schemes, see below) are unlikely to happen any time soon. In the case of the Tcl/C implementation there are active discussions on the modules-interest mailing list, but changes or improvements are infrequent as well. At the time of writing, the latest available version (3.2.10) was released Dec. 2012, which was over one year after the previous release.

In 2009, a radically new implementation of the module system called Lmod [7] emerged. Implemented from the ground up in Lua, it is largely compatible with the Tcl-based implementations. Lmod is actively developed and maintained, and has an active and thriving community. We discuss Lmod in detail in Section V.

B. Module files

In essence, module files are shell-independent text descriptions of the changes to a user's environment required to support a particular software package. Such changes may include adjusting common environment variables such as `$PATH`, `$CPATH` and `$LIBRARY_PATH`, respectively pointing to the locations for binaries, header files and libraries, or setting additional package-specific variables. In addition, module files typically include a brief one-line description of the package displayed by `module whatis`, as well as a longer help text printed by `module help` to describe basic usage, where to find the package documentation, and whom to contact in case of usage problems.

Module files are searched in directories specified by the environment variable `$MODULEPATH`. The name of a module is defined as the path to the corresponding module file in one of the directories that are part of `$MODULEPATH`. For example, the module file located in `<prefix>/GCC/4.8.2` provides a module for version 4.8.2 of the GNU Compiler Collection (GCC) with the name `GCC/4.8.2`, with `<prefix>` being one of the `$MODULEPATH` entries.

C. Module naming schemes

When the environment module system was invented, HPC sites typically provided a single version of a single compiler on each of their systems. Today, HPC systems provide multiple compilers (GCC, Intel, Clang, PGI, ...), each available in multiple versions.

In most cases (there are some exceptions for programs written in pure C), programs compiled with one version of a compiler cannot safely link to libraries compiled with another. This means that multiple builds of libraries (e.g., Boost) need to be provided, one for each compiler and version. Moreover, since packages such as MPI implementations are inherently tied to a particular compiler and most often to a particular version, disambiguating module names can be a daunting task. For example, modules corresponding to version 1.7.3 of the Open MPI library built with different compilers might be named as follows:

```
% module avail OpenMPI
OpenMPI/1.7.3-GCC-4.8.2
OpenMPI/1.7.3-Intel-14.0
```

The situation becomes even more complicated for scientific software packages like WRF [11] compiled with a particular compiler and linked against a particular MPI library, e.g.:

```
% module avail WRF
WRF/3.5-GCC-4.8.2-OpenMPI-1.7.3
WRF/3.5-Intel-14.0-MVAPICH2-1.9
```

Note that such packages in many cases also depend on a set of mathematical libraries, such as Intel MKL vs. ACML vs. OpenBLAS+(Sca)LAPACK+FFTW, which complicates matters even further.

A possible solution to this issue is to define so-called *toolchain* modules, packaging together a compiler, an MPI library, and one or more packages providing linear algebra and FFT functionality. For example, a `goolf` toolchain module may combine (i.e., implicitly load modules for) a compatible combination of specific versions of GCC, Open MPI, OpenBLAS, (Sca)LAPACK and FFTW. In this approach, the first WRF module shown above could instead be installed on top of such a toolchain, and might take the, slightly shorter, name `WRF/3.5-goolf-1.6.10`. One downside of using toolchains with terse naming conventions, however, is clear: toolchain names can be cryptic, and the toolchain version generally has no direct relationship to the versions of the encapsulated packages, so users lack top-level insight into this information.

One common attempt to manage the potentially overwhelming set of available module files is to group them in different subdirectories. This makes it possible to list the subdirectories separately in the `$MODULEPATH`, resulting in clearer, more useful module names that are nicely separated in the output of `module avail`, for example:

```
% module avail
----- <prefix>/compiler -----
GCC/4.8.2   Intel/14.0  Clang/3.4
-------- <prefix>/mpi --------
OpenMPI/1.7.3-GCC-4.8.2
OpenMPI/1.7.3-Intel-14.0
```

However, this involves picking a single category for each software package to install, which may become cumbersome with scientific software that crosses multiple research domains. The toolchain concept also offers the possibility to categorize modules by toolchain, but if a software package is available for multiple toolchains, it will show up in multiple sections of the `module avail` output, which is not desirable.

Although all these approaches aim to improve the overall organization of the available modules, a typical module listing on an HPC system can still be overwhelming, as the total number of modules can easily be in the order of several hundreds. Moreover, none of these approaches prevent (especially novice) users shooting themselves in the foot by loading modules which are incompatible to each other. While there are ways to prevent this, they require identifying and explicitly listing all conflicting modules. For example, the Open MPI module may specify that it is incompatible with modules that load other MPI libraries like Intel MPI or MVAPICH2. However, this means that these conflict specifications have to be adjusted every time an additional MPI library is installed on the system, which is clearly a maintenance nightmare for system administrators.

The 'flat' module naming scheme discussed here is common, but it places a significant burden on users. If a user requires only a single application like WRF, picking a module is fairly straightforward. But when multiple software packages are required at the same time, e.g., when an application developer is using multiple libraries to build a parallel application, he or she must pick modules for a compiler, MPI stack and other required libraries that are compatible with each other. After loading a mismatched set of modules the user might get lucky: the application might fail to run or die immediately. However, the application may also fail in subtle and mysterious ways, and can thus consume a great deal of time from both users and system staff in trying to resolve the problem. Section III outlines another approach to remove this burden from users and staff.

D. Building and installing scientific software

HPC sites around the world use a wide variety of methods and tools to install scientific software. In this section, we provide a brief overview of these techniques, highlighting the associated issues and problems. These observations are supported by the results of a recent poll concerning these topics [12].

1) Manual installation: Commonly, sites rely heavily on the manpower of (a part of) the user support team, and simply manually install software packages following the install guides developed internally over time or provided by the respective software development teams (if the latter are available, current, and sufficiently detailed).

2) Scripting: Frequently, sites cobble together a collection of scripts to automate the often repetitive and error-prone tasks of configuring, building and installing the software packages, using any of a variety of scripting languages. Typically, this quickly results in a pile of loosely coupled, hard-to-maintain scripts, often understood by just a small fraction or even a single member of the user support team [13]. On top of this, these scripts tend to encode site-specific software installation policies, making reuse by other sites difficult (assuming the scripts are made available to others).

3) Package managers: Yet another approach is to rely on the package managing tools used by the operating system, e.g., RPMs and `yum` for RedHat-based systems, `apt-get` and Debian packages for Debian-like systems, Portage for Gentoo, etc. Package managers provide adequate support for some aspects of installing large software stacks, including dependency tracking/resolution, software updates, uninstalling software, etc. However, they are ill-suited for dealing with certain peculiarities that come into play when installing scientific software on HPC systems. These include supporting multiple builds/versions of the same software package to be installed at the same time, and heavily customized install procedures involving non-standard techniques beyond the common `configure – make – make install` paradigm. In addition, the package specification formats (e.g., `.spec` files for RPMs) tend to provide little support for factoring out common patterns in install procedures, leading to copy-pasted, hard-to-maintain package specifications.

Several of the larger HPC sites in the world take this approach, since they are able to dedicate large amounts of manpower to the task of installing scientific software, but this is typically infeasible for smaller HPC sites. Besides these concerns, the effort spent on shoe-horning the install procedures of scientific software into package specifications are unlikely to benefit other HPC sites: site-specific installation policies require labor-intensive modifications, and redistribution of packaged builds is sometimes prohibited for licensed software.

Examples include LosF [14] developed by the Texas Advanced Computing Center (TACC) to leverage RPMs developed in-house (encapsulating both the software itself and the accompanying module file), the XSEDE Compatible Basic Cluster (XCBC) [15] which provides a collection of RPMs that allow for making an HPC system 'XSEDE-compatible', and CernVM-FS [16], a read-only distributed file system based on HTTP which is optimized to distribute readily built software applications.

4) Custom tools: Other solutions include custom-made tools for installing scientific software on HPC systems. We briefly discuss a number of these in Section VIII. Typically, these tools begin as a collection of scripts and mature as the complexity of the local operation increases; they often reach a point where they may prove useful for other sites. Unfortunately, these projects typically die a silent death as quickly as they surfaced: the sole original developer is no longer available, the documentation is inadequate, or the infrastructure is not sufficiently flexible or feature-rich to support the needs of multiple sites. In Section IV we discuss in detail one notable exception, EasyBuild [6].

5) Creating module files: It is tempting to deem module files "simple enough" to create manually, and in fact this is common practice. However, this significantly impedes the goal of maintaining a consistent set of module files. Moreover, the burden of doing this work often falls on a small number of intrepid staff members (if not a single person). This is clearly a major concern on production systems, where continuity of support is a high priority.

E. Lack of collaboration

Even though these problems are well recognized, there is an abundant lack of available tools and practices for addressing them. Moreover, there has been very little collaboration between HPC sites on these issues, despite the significant burden they present for support staff. This is especially true at smaller sites. Even though HPC sites around the world all face these problems, the duplication of effort and associated labor cost is alarming. There is a tremendous opportunity here: we all stand to benefit from collaborative initiatives that leverage the immensely valuable expertise available across HPC sites worldwide.

In the remainder of this paper we seek to help resolve these problems by describing a modern, robust, and flexible alternative to these traditional approaches.

III. Hierarchical module naming scheme

Using a *hierarchical module naming scheme* is an excellent way to help users avoid the pitfalls described earlier. This approach makes it possible to organize environment modules in a more structured way. The key idea is to make modules available in a step-by-step fashion as their dependencies become part of the user environment. Initially only a small number of so-called *core* modules are available to the user. Core modules are those that do not depend on software choices available to the user; they are either completely self-contained or depend only on basic system software. Examples include modules for compilers, and statically linked software like debuggers.

These core modules extend the module search path (`$MODULEPATH`) when loaded to make additional modules visible, for example the ones which are built with—and therefore depend on—the compiler being loaded. Separate sub-directories for each version of each compiler store the software and module files that depend on that compiler. For the Open MPI example presented in Section II-C, this means that the user only sees the module for Open MPI 1.7.3 that depends on the current compiler of choice:

```
% module avail
------------ <prefix>/Core ------------
GCC/4.8.2   Intel/14.0  Clang/3.4
% module load GCC/4.8.2
% module avail
------------ <prefix>/Core ------------
GCC/4.8.2   Intel/14.0  Clang/3.4
----- <prefix>/Compiler/GCC/4.8.2 -----
OpenMPI/1.7.3
```

Such a module hierarchy is not limited to a single level. The module files for each MPI implementation, for example, can further extend `$MODULEPATH` to make visible the modules that depend on the currently loaded compiler and MPI stack.

A. Advantages over traditional module naming schemes

Using a hierarchical module naming scheme has a number of important advantages. First, at any point in time, users see only the modules which are meaningful in the current context. That is, the list of available modules is much shorter and grouped by level of the hierarchy. This is easier for users to process when hundreds of modules are provided (which is generally the case on large HPC systems).

Second, encoding the dependency chain in the module name is no longer necessary. This leads to significantly shorter and more intuitive module names, usually consisting of the software name and version, e.g., `WRF/3.5` instead of the long and cryptic module names shown in Section II-C.

Third, loading incompatible modules is much more difficult, which eliminates a broad class of subtle errors that are difficult to debug. This not only dramatically improves the user experience; it also significantly reduces the time a user support team needs to spend on related problems.

Finally, this structured organization of module files provides a number of new opportunities to enhance the user experience. One significant example is module *swapping*: when the user chooses a new compiler or MPI stack by executing a command like `module swap GCC Intel`, the module system could (and should) automatically replace higher-level modules with new versions compatible with the user's new lower-level selections. We strongly believe that this capability should be part of any robust module system; see Section III-B.

B. Using a hierarchical module naming scheme

In theory, the commonly used Tcl/C and Tcl-only environment modules tools both support the concept of hierarchical modules. This is because these tools function in a way that is essentially independent of any particular choice in organizing the module files. In practice however, some difficulties when using these tools with a module hierarchy come forward.

1) Visibility of modules: In the hierarchical scheme we describe above, a module is not available (visible) to the user until its lower-level modules are loaded. This is by design, and has great advantages. But a module system should not leave to the user the burden of locating modules of interest in a complicated directory structure. Instead, the module system should provide a built-in, natural mechanism for exposing hierarchies and dependencies, and make it possible for users to load required modules without resorting to raw searches of the system's directory structure. In particular, the module system must provide a means for displaying relevant information about modules that are outside of the current module search path.

2) Awareness of `$MODULEPATH` extensions: The modules tool should be aware of the changes to the module search path that occur when loading modules in a hierarchical scheme. This is what makes automatic module swapping possible. When the user executes `module swap` to replace one module with another, the tool needs to i) detect that a particular module depends on the module being swapped out (this requires an awareness of the hierarchical structure); ii) unload the dependent modules and iii) afterwards automatically (try to) replace them with equivalent compatible modules, taking into account the correct order in which to (re)load those.

3) Module availability on different paths in the hierarchy: The modules tool also needs to take into account that it may not always be possible to reload all dependent modules after swapping modules. When swapping one compiler for another for example, it is possible that no compatible version of a higher-level module is available. Unless the modules tool deals with this issue appropriately, the user could end up stuck with a broken environment somewhere between the original state (before the swap) and the intended end state (in which all dependent modules are reloaded). Ideally, the modules tool should notify the user when dependent modules cannot be reloaded. Moreover, if the modules tool can keep track of which modules failed to reload it can also support the ability to revert the swap, and restore the original set of loaded modules.

The existing Tcl/C and Tcl-only modules tools currently do not provide any of this functionality. At one point, the Tcl-only tool did briefly support reloading dependent modules after a `module swap`, but this capability later disappeared because of incompatibility with the Tcl/C version.

C. Maintaining a module hierarchy

Next to the important issues related to using a module hierarchy discussed in the previous section, an additional consideration is the potential impact on the user support staff maintaining the modules.

In particular, a great deal of care must be exercised in constructing module files for a hierarchical software stack; failing to do so can produce unexpected results and inconsistent environments. One must think carefully about the necessary module search path extensions required at different levels of the hierarchy, take dependencies into account, and consider the order in which users should load modules. Additionally, the module files must be designed such that they use the short name visible to users (e.g., `WRF/3.5`) rather than the 'full' name relative to the top of the hierarchy (e.g., `MPI/GCC/4.8.2/OpenMPI/1.7.3/WRF/3.5`).

Creating module files manually is already a tedious task; doing so by hand in a hierarchical context only further complicates this. It should be clear that a modern HPC environment requires powerful, flexible, and reliable tools and techniques that truly *automate* the task of installing scientific software.

In the sections that follow we describe how two community-driven tools work together to achieve this task: EasyBuild, a software build and installation framework, and Lmod, a modern alternative to the Tcl-based environment modules tools. Because our experience confirms the substantial benefits of a hierarchical module naming scheme, and because we strongly believe that the HPC community is ready to (and should) move in this direction, we focus primarily on the aspects of these tools that support this approach.

IV. EasyBuild: Automated software installation

EasyBuild [6] is a software build and installation framework written in Python. Its primary goal is to alleviate the ubiquitous burden of building and installing scientific software [5].

The EasyBuild project was started in 2009 by the HPC team of Ghent University (Belgium), out of frustration over the lack of appropriate tools for dealing with the installation of scientific software. The first public release of EasyBuild (version 0.5) occurred in April 2012 under an open source license (GPLv2), after 2.5 years of in-house development. In November 2012, EasyBuild v1.0 was released featuring a stable API. Under a 'release early, release often' strategy with a new major release every 4 to 6 weeks, the development team has a strong focus on continuous, high quality maintenance, support, and improvement. At the time of writing, the latest release is EasyBuild v1.15.2 (Oct'14). This version only supports x86-based Linux systems, however support for other platforms (Linux/POWER, Cray, ...) is being developed.

Under the motto "*building software with ease*", EasyBuild aims to provide an easy yet powerful way to automatically install (scientific) software stacks, in a robust, consistent and reproducible way. Its design is deliberately made very flexible and modular (see Sections IV-C2 and IV-C3) making it a well suited platform for collaboration across HPC sites, as is confirmed by the steadily growing EasyBuild community (see Section VI-A).

A. Concepts and design

EasyBuild consists of a collection of Python modules and packages that interact with each other, dynamically picking up additional Python modules as needed for building and installing a (stack of) software package(s) specified via simple specification files. Or, in EasyBuild terminology: the EasyBuild *framework* leverages *easyblocks* to automatically build and install software using a particular *compiler toolchain*, as specified by one or multiple *easyconfig files*.

1) EasyBuild framework: The EasyBuild framework embodies the core of the tool, providing functionality commonly needed when installing scientific software on HPC systems. For example, it deals with downloading, unpacking, and patching of sources, loading module files for dependencies, setting up the build environment, autonomously running (interactive) shell commands, creating module files that match the specification files, etc.

Included in the framework is an 'abstract' implementation of a software build and install procedure, which is split up into different *steps*: unpacking sources, configuration, build, installation, module generation, etc. Most of these steps, i.e., the ones that are generally more-or-less analogous across different software packages, have appropriate (default) implementations. The only exceptions are the configuration, build and installation steps that are purposely left unimplemented (since there is no common procedure for them). Each of the steps can be tweaked and steered via different parameters known to the framework, for which values are either obtained from the provided specification files (see Section IV-A4) or set to reasonable default values.

In EasyBuild v1.15.2 the framework source code consists of about 19 000 lines of code, organized across about 125 Python modules in roughly a dozen Python package directories, next to almost 7 000 lines of code for tests. This provides some notion of the size of the EasyBuild framework and the amount of supporting functionality it has to offer.

2) Easyblocks: The implementation of a particular software build and install procedure is done in a Python module, which is aptly referred to as an 'easyblock'. Each easyblock ties in with the framework API by defining (or extending/replacing) one or more of the step functions that are part of the abstract procedure used by the EasyBuild framework. Easyblocks typically heavily rely on the supporting functionality provided by the framework, for example for (autonomously) executing (interactive) shell commands and obtaining the command output and exit code.

A distinction is made between *software-specific* and *generic* easyblocks. Software-specific easyblocks implement a build and install procedure which is entirely custom to one particular software package (e.g., WRF), while generic easyblocks implement a procedure using standard tools (e.g., CMake). Since easyblocks are implemented in an object-oriented scheme, the step methods implemented by a particular easyblock can be reused in others via inheritance, enabling code reuse across build procedure implementations. This is a major advantage over a script-based approach which typically involves lots of copy-pasting of code.

For each software package being built, the EasyBuild framework will determine which easyblock should be used, based on the name of the software package or the value of the `easyblock` specification parameter. In case an easyblock specification is not provided and no (software-specific) easyblock matching the software name could be found, a fallback mechanism will resort to using the generic `ConfigureMake` easyblock, which implements the common `configure` – `make` – `make install` procedure.

At the time of writing, the most recent release of EasyBuild (v1.15.2) includes 139 software-specific easyblocks and 20 generic easyblocks, providing support for automatically installing a wide range of software packages. Examples range from fairly easy-to-build programs like gzip, over basic tools like compilers, various MPI stacks and commonly used libraries, to large scientific software packages that are notorious for their involved and tedious install procedures, such as CP2K, NWChem, OpenFOAM, QuantumESPRESSO, and WRF.

3) Compiler toolchains: EasyBuild also employs so-called 'compiler toolchains' (or simply 'toolchains' for short), which were already mentioned in Section II-C. A toolchain consists of a (set of) compiler(s), usually together with some libraries for specific functionality, e.g., for using an MPI stack for distributed computing, or which provide optimized routines for commonly used math operations, e.g., the well-known BLAS/LAPACK APIs for linear algebra routines. For each software package being built, the toolchain to be used must be specified (see Section IV-A4).

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 `$F90`), compiler and linker options (e.g., via `$CFLAGS` and `$LDFLAGS`), the list of library names to supply to the linker (via `$LIBS`), etc. This enables making easyblocks largely toolchain-agnostic since they can simply rely on these environment variables; that is, unless they need to be aware of, for example, the particular compiler being used to determine the build configuration options.

Recent releases of EasyBuild include out-of-the-box toolchain support for: various compilers, including GCC, Intel, Clang, and CUDA; common MPI libraries such as Intel MPI, MPICH2, MVAPICH2, and Open MPI; various numerical libraries including ATLAS, Intel MKL, OpenBLAS, and ScaLAPACK; and libraries providing FFT routines like FFTW.

4) Easyconfig files: The specification files that are supplied to EasyBuild are referred to as 'easyconfig files' (or simply 'easyconfigs'), which are basically plain text files containing (mostly) only key-value assignments for build parameters supported by the framework, also referred to as 'easyconfig parameters'. Some parameters are mandatory, like `name` and `version` to specify which software (version) should be installed, `toolchain` to indicate which compiler toolchain should be used, etc. Others are optional, and have appropriate defaults set for them in the EasyBuild framework; examples include `buildopts` to specify options for the build command, and `dependencies` to list the software dependencies that should be taken into account. Note that easyconfig files only provide the bits of information required to determine the corresponding module name; the module name itself is computed by EasyBuild framework by querying the module naming scheme being used, see also Section IV-D. The complete list of supported easyconfig parameters can be easily obtained via the EasyBuild command line.

As such, each easyconfig file provides a complete specification of which particular software package should be installed, and which settings should be used for building it. After completing an installation, EasyBuild copies the used easyconfig file to the install directory, and also supports maintaining an easyconfig archive which is updated on every successful installation. Therefore, reproducing installations becomes trivial.

EasyBuild v1.15.2 includes over 2 800 easyconfig files, for 511 different software packages.

B. Basic usage

As the name suggests, EasyBuild is intended to be particularly easy to use. A script aptly named `eb` is provided to interact with the EasyBuild framework. Launching `eb` with an easyconfig file as an argument triggers a series of events. First, the easyconfig file will be parsed by the EasyBuild framework to determine which software package needs to be installed, and which easyblock and build parameters should be used. Afterwards, the environment is prepared by loading the modules for the specified toolchain and dependencies—if those are available—and the toolchain support in the framework defines additional environment variables as discussed in Section IV-A3. In case the required modules are not available an appropriate error message is shown, unless the automatic dependency resolution mechanism is enabled via the command line option `--robot` (see Section IV-C1 for more details). Next, the appropriate build and install procedure is executed step by step, as defined by the framework and the selected easyblock. Finally, if the installation was successful, a module file is generated that matches the used specifications in terms of module name and contents.

For example, the following command line instructs EasyBuild to install WRF v3.5 and its missing dependencies with the `goolf` toolchain v1.6.10, as specified by the provided easyconfig file:

```
% eb WRF-3.5-goolf-1.6.10.eb --robot
```

Further details on using `eb` are beyond the scope of this paper; we refer to `eb --help` and the EasyBuild wiki pages [17] for extensive documentation.

C. Feature highlights

We now briefly highlight the key features of EasyBuild that are relevant to the topic of this paper, including automatically resolving dependencies and the flexibility of the EasyBuild framework. Other interesting features which are not discussed here include, but are not limited to, thorough logging of the build and install process, support for tweaking provided easyconfig files directly from the `eb` command line, and offloading individual installations to a cluster resource manager like PBS.

1) Automatic dependency resolution: EasyBuild supports installing an entire software stack, including the required toolchain if needed, with a single `eb` invocation. By enabling the `--robot` command line option, the dependency resolution mechanism will construct a full dependency graph for the software package(s) being installed, after which a list of dependencies is composed for which no module is available yet. Each of the retained dependencies will then be built and installed, in the required order as indicated by the dependency graph. This is particularly useful for software packages that have an extensive list of dependencies, or when reinstalling software using a different compiler toolchain.

2) Configurability: Allowing users to modify EasyBuild's default behavior to their needs is another important feature. EasyBuild can be configured in three different ways: via one or more configuration files, via environment variables prefixed with `EASYBUILD_`, and via command line options. For each command line option, a matching environment variable can be set or matching setting can be defined in a configuration file. Command-line options overrule environment variables, while they in turn take precedence over configuration files, resulting in a flexible way of controlling EasyBuild's behavior. Configuration parameters are available for straightforward aspects such as the installation prefix for software and module files, but also, for example, for the log level, the modules tool, and the module naming scheme that should be used.

3) Dynamic extensibility: Another key feature is that EasyBuild can be extended dynamically, i.e., the framework is designed such that additional functionality can be easily plugged in: additional easyblocks, support for more compilers and libraries to be part of a toolchain, custom module naming schemes, etc. Extending EasyBuild is done by implementing a Python module in the required namespace (e.g., `easybuild.easyblocks`), and modifying the `$PYTHONPATH` environment variable to make sure that it is available in the Python search path at runtime. Every time `eb` is used, the framework will 'scan' the Python search path to determine the available options for each of the dynamically extendable aspects. This provides EasyBuild users the freedom to experiment with functionality which is not (yet) available in the particular version they are using, or to extend the supported options for different aspects of EasyBuild with additional site-specific options, e.g., easyblocks for in-house software packages or a custom module naming scheme.

D. *Support for hierarchical module naming schemes*

Since EasyBuild v1.14.0, sufficiently fine-grained control of the active module naming scheme is available to support custom hierarchical module naming schemes. As discussed in Section IV-C, it suffices to implement the specific details of the hierarchical module naming scheme in a Python module, which must live in the `easybuild.tools.module_naming_scheme` namespace. In particular, the Python module must provide a Python class that derives from the 'abstract' `ModuleNamingScheme` class provided by the EasyBuild framework, and defines a number of methods that cover different aspects of the module naming scheme. For example, the `det_modpath_extensions` method must return a list of strings representing `$MODULEPATH` extensions, given a parsed easyconfig file as an argument. This is a simple yet powerful approach, providing full control over all particular aspects of the module naming scheme. With this in place, EasyBuild must be configured to use this particular module naming scheme, see also Section IV-C2.

Letting EasyBuild automatically generate module files under a specified (hierarchical) module naming scheme stands in stark contrast with the practice of manually creating module files while trying to consistently apply site policies. It avoids the various issues involved with manually creating module files with respect to consistency and correctness which were discussed in Section II, and as such mostly relieves user support teams from another tedious task related to installing scientific software.

V. LMOD: A MODERN MODULES TOOL

Lmod [7], [18], [19] is a Lua-based [20], modern alternative to the traditional environment modules implementations. It is a drop-in replacement for the Tcl/C and Tcl-only implementations, except for a few corner cases, and can consume module files written in both Tcl and Lua. Its primary design goals are to empower users to control their working environment and improve the user experience without hindering experts.

Lmod began as a prototype designed to test ideas on how to manage a hierarchical module system. Early on, it became clear that existing tools were inadequate; the developers began a complete rewrite based on the clean and powerful constructs of the Lua programming language. Lua also fills the same niche for which Tcl was originally designed: both languages embed and extend easily into other programs. Lmod's pre-release versions proved to be fast, flexible, and robust; the system quickly moved from a prototype to a worthy replacement for the other, established modules tools.

October 2008 marked the first public release of Lmod; it has been under active development ever since. TACC adopted Lmod for its production systems in October 2009. Since February 2011, there have been 153 tagged versions and 30+ official releases. At the time of writing, the latest available version is 5.7.5.

A. *Support for hierarchical module naming schemes*

The key to Lmod's support for a module hierarchy is the fact that it monitors changes to `$MODULEPATH`. Doing so enables it to support the advanced notion of swapping hierarchical modules from the ground up, resolving the issues described in Section III-B.

Swapping one (core) compiler module for another using Lmod triggers a chain of events. The loaded compiler module is unloaded, causing the matching paths to be removed from `$MODULEPATH`. Subsequently, any loaded modules for MPI and other packages that depend on it are unloaded as well, because the respective module files are no longer available in the active module search path. These modules are marked *inactive*. Next, the new compiler module is loaded, causing new entries to be added to the `$MODULEPATH`. This triggers a search for a compatible module in the new module search path for each of the inactive modules.

For example, swapping a loaded `GCC` compiler module for the `Clang` module results in reloading the modules that depend on it, e.g., `FFTW` (a parallel FFT library) and `MPICH` (an MPI library):

```
% module list
Currently loaded modules:
1) GCC/4.8.2  2) MPICH/3.1.1  3) FFTW/3.3.2
% module swap GCC Clang
The following have been reloaded:
1) FFTW/3.3.2  2) MPICH/3.1.1
% module list
Currently loaded modules:
1) Clang/3.4  2) MPICH/3.1.1  3) FFTW/3.3.2
```

Lmod lists reloaded modules alphabetically after a `module swap`, while the output of the `module list` command lists them in the order in which they were loaded.

Another key issue with a hierarchical layout of modules is that not all existing modules are visible through `module avail` (see Section III-B1). Lmod includes a new command, `module spider`, to search for modules across the entire module tree and report all *existing* modules. The semantics of `module avail` is the same as it is with other module tools to ensure compatibility; this command reports all modules that can be *loaded* in the current context (determined by `$MODULEPATH`).

The ability to maintain a consistent set of hierarchical modules and navigate the module tree outside of the current `$MODULEPATH` are among the key strengths of Lmod. There are however a number of other interesting features, many of which have been implemented in response to requests by the members of the vibrant and demanding Lmod community.

B. *The `ml` command and unload/swap shortcut*

Lmod provides a additional command named `ml` for those who tend to misspell the `moduel`, `mdoule`, *err* `module` command, which focuses on commonly used subcommands. Without arguments `ml` is a synonym for `module list`, while using it with an argument that is not recognized as a subcommand (e.g., `ml foo`) corresponds to loading the specified module (e.g., `module load foo`). `ml` accepts other subcommands associated with `module`; for example `ml avail` and `module avail` are equivalent. In the unlikely case that a module name is the same as a subcommand supported by Lmod, some care must taken when using `ml`. For example, loading a module named `spider` should be written as `ml load spider`.

Another shortcut supported by the Lmod command line is unloading modules by prefixing module names with a minus sign ('-'). This also allows users to swap both the compiler and MPI stack with a single command, for example:

```
% ml -GCC -MPICH Clang OpenMPI
```

which is equivalent to:

```
% module swap GCC Clang
% module swap MPICH OpenMPI
```

C. Properties

Lmod also makes it possible to assign *properties* to modules, indicating that a package has some particular capability or characteristic. This is particularly useful on modern supercomputers that include one or more types of accelerators, e.g., GPUs or Intel Xeon Phi coprocessors. To designate that libraries or applications support execution on (one of) these accelerators, one can assign properties in the corresponding module files using the `add_property` function. Module properties are reported in the output of `module` subcommands, making them visible to users:

```
% module list
Currently Loaded Modules:
  1) Intel/14.0.2         3) MPICH/3.1.2
  2) imkl/11.1.3.174 (*)  4) Boost/1.55.0 (P)
Where:
 (*):  supports host, native and offload
 (P):  built for host and native Phi
```

Another use of this feature is marking modules as 'alpha' or 'beta' to characterize the maturity and stability of the associated software. Sites can easily add properties of their own.

D. Caching

Support for module properties does add a complication, however. Without properties, the `avail` subcommand only requires the names of available module files, not their contents. However, to support properties Lmod must parse the contents of all module files, whether or not these files include properties. On some file systems this can be slow. To mitigate this, Lmod supports *caching* of module files, as reading one single (possibly large) file is faster than walking a directory tree and reading lots of small files. Caching also makes other subcommands, e.g., `module spider`, significantly faster. An Lmod module file cache is also referred to as 'spider cache'; both system-level and user-level caches are supported.

Lmod contains hooks to check whether a cache is up-to-date and will update a user-level cache when deemed necessary, i.e., when executing a particular subcommand is taking too long. System cache files can be recomputed after (re)installing software and/or periodically via a dedicated cron job. Usually, the presence of a system-level cache eliminates the need for a user-level cache. Lmod trusts existing cache files in all cases except for the `load` subcommand, that is, modules can be loaded even if the cache is slightly out-of-date (e.g., immediately after installing a new software package).

E. Module families

Another Lmod enhancement is support for module *families* via calls to the `family` function in module files. Families are an alternative approach to managing module conflicts; only one module of a particular family can be loaded at any given time. While conflicts can only be defined based on module names, module family 'labels' can be chosen freely and are easy to maintain: a new family member does not require changes to existing module files.

For example, one might define a `compiler` family, by simply including `family("compiler")` in each compiler module file. This would result in the following meaningful error message when a user tries to load two compiler modules at the same time:

```
% module load Clang
% module load GCC
Lmod has detected the following error:
You can only have one compiler module loaded.
To correct this, please do the following:
    module swap Clang GCC
```

The constraint of only allowing to load one module per family can be disabled by using Lmod in 'expert mode', by defining the `$LMOD_EXPERT` environment variable.

F. Customizing Lmod behavior and hooks

Another key Lmod feature is support for site-specific customizations via a file named `SitePackage.lua`. Lmod provides several *hook* functions, allowing sites to plug in additional functionality. For example, using the available hook for the `load` subcommand, a site can add a new accompanying action. A typical example is a message logger: by adding this action to the `load` subcommand, a site can collect and analyze module usage across the entire system.

G. `pushenv`: stack-based setting of environment variables

Environment module systems support the `setenv` function for defining environment variables via module files. When a module file is unloaded, any environment variables that were defined by that module are cleared. In some cases, however, this can be problematic: the environment variable may have had a meaningful value before the module was loaded.

Lmod enables a more sophisticated approach by supporting a stack-based alternative via the `pushenv` function. For environment variables defined in this way, Lmod maintains a hidden stack in the environment so that previous values can be recovered. For example, consider a module for the GCC compiler that defines `$CC` to be `gcc`, while a module for the Open MPI library sets `$CC` to `mpicc`. The following table shows the value of `$CC` for a series of module commands, depending on whether `setenv` or `pushenv` is used:

command	`setenv`	`pushenv`
`export CC=icc`	`icc`	`icc`
`module load GCC`	`gcc`	`gcc`
`module load OpenMPI`	`mpicc`	`mpicc`
`module unload OpenMPI`	*(none)*	`gcc`
`module unload GCC`	*(none)*	`icc`

Note that modules using `pushenv` properly restore the previous value for `$CC` when they are unloaded.

H. Module collections

One final feature of Lmod worth mentioning is the support for user-defined *collections* of modules. Collections allow users to define and name sets of module files that can be loaded with a single command, or automatically on startup. Users only need to load the desired modules and then execute `module save` with an optional name specified as an argument; omitting the name defines the default collection. Users can list the available collections via `module savelist` or the shorthand `ml sl`. A collection can be restored by specifying its name to the `restore` subcommand. The `module restore` command without an argument (or starting a new login session) reloads the default module collection. This is a powerful, popular way to efficiently manage working environments.

VI. Other aspects to EasyBuild & Lmod

A. *Communities*

Active development and vibrant communities characterize both EasyBuild and Lmod. Requests and suggestions from their respective community lead to new features implemented by the core development teams. The users themselves also significantly contribute by reporting bugs, sharing patches, and even implementing new features that make their way into the public baselines. This effectively makes both tools platforms for collaboration, letting all users benefit from enhancements.

Estimating the size of these communities is difficult. Roughly a dozen different HPC sites actively contribute to EasyBuild, with several others using it. The EasyBuild mailing list has about 70 subscribers to date, so an estimate of over one hundred users is plausible. The EasyBuild IRC channel, one of the other main community hubs, has about a dozen of daily active users. HPC sites all over the world are using EasyBuild, including various sites in Europe (e.g., Jülich Supercomputing Centre in Germany), Idaho National Laboratory (US), University of Auckland (New Zealand), etc., as well as some commercial companies (e.g., Bayer).

The Lmod mailing list has about 50 subscribers, which is no more than 20% of the Lmod community. Since a couple of hundred HPC sites already deploy Lmod, the actual number of Lmod users is certainly many thousands. TACC, with 15 000–20 000 users on three major systems, provides only Lmod as a modules tool. Lmod downloads from the SourceForge code repository number several thousand, across all versions. Lmod is deployed at numerous HPC sites world-wide, including North America (TACC, Stanford, Harvard, etc.) and Europe (University of Warwick (UK), Arctic University of Norway, etc.), and also has commercial users, including Total.

B. *Synergy*

Recently, the EasyBuild and Lmod teams have developed a strong and mutually beneficial synergy. Both projects have a community-oriented vision and highly value flexibility, which appeals to users from both communities. Since EasyBuild makes installing scientific software significantly easier, user support teams can quickly generate many environment modules, which could become overwhelming for users. This strengthens the need for a modern modules tool that can efficiently deal with large collections of available modules, which led to the EasyBuild community reaching out to the Lmod community. Since then, support for using Lmod as a modules tool and hierarchical module naming schemes has been integrated into EasyBuild, and further enhancements are planned (see also Section VII).

Similarly, Lmod has benefited greatly from feedback by EasyBuild users. For example, support for `pushenv` (see Section V-G) grew out of discussions on the Lmod mailing list by EasyBuild stakeholders. Recent versions of Lmod have also included significant speed improvements triggered by issues uncovered through EasyBuild. A particular example is the fact that the `module --terse avail` command, which provides machine-readable output and is used by EasyBuild, no longer parses module files to obtain module properties, since this is not necessary to compose the terse output text.

The synergy between EasyBuild and Lmod results in improved tools on both sides, and is driving the state of the practice in directions that we hope will have major impact.

VII. Future work

Although the latest versions of both EasyBuild and Lmod already provide flexible support for hierarchical module naming schemes, further enhancements to extend the capabilities and improve the user experience are planned. Here we outline several promising possibilities.

The latest public version of EasyBuild is still unaware of Lmod-specific concepts that can be included in module files, like properties and families. Additionally, other concepts including non-strict version specifications, e.g., `load(atleast("GCC","4.8"))`, in load statements can only be included in Lua module files. Integrating these capabilities into EasyBuild will greatly enhance the value of both products, since it would allow HPC sites to maintain module families and properties with little effort.

Although the required support is available, a deeper module hierarchy that better matches the toolchain concept in EasyBuild is worth working out. Usually only the `Compiler` and `MPI` hierarchy levels are used, while EasyBuild toolchains typically also include math libraries. Extending the two-level hierarchy to capture this level of flexibility is not straightforward however; BLAS/LAPACK/FFT functionality can be provided by one single library (e.g., Intel MKL), or by multiple distinct libraries (e.g., OpenBLAS, LAPACK and FFTW). This complicates the design of a deeper hierarchy that supports swapping one math library for one or more alternatives.

Related to this is the more generic concept of a ‘multi-dimensional’ module hierarchy, also referred to as a module ‘matrix’, which is currently an area of active research. In such a multi-dimensional module hierarchy, the goal is that a module would only become available when *multiple* other modules are loaded together. This is useful for software packages that provide support for multiple libraries that incorporate some particular functionality, like the parallel solver library PETSc which supports different LAPACK/FFT libraries. In such a module hierarchy a module for PETSc, for example, would not be available until modules for both, e.g., OpenBLAS *and* FFTW have been loaded. On top of this, the system should also support swapping out the loaded modules for OpenBLAS and FFTW for one single module providing equivalent functionality (e.g., an Intel MKL module), and trigger a reload of the PETSc module. Supporting this would be particularly useful for application developers and benchmarking studies.

Another improvement to EasyBuild would be a dependency resolution mechanism that is aware of subtoolchains. When the EasyBuild framework checks the availability of modules that match the required dependencies, it only uses the toolchain selected for the software being installed (unless this is overridden specifically on a per-dependency basis). This may require that modules which are, for example, only dependent on a compiler are built and installed multiple times with different toolchains, even though they are providing the same software builds. This yields more modules than required and consumes additional disk space. Checking for modules that were installed with a compatible subtoolchain (providing for example only the compiler) would ensure that these tools only need to be installed once.

Mixing environment modules installed with different naming schemes leads to problems, especially when both flat and hierarchical module naming schemes are in place. As a result, migrating to a hierarchically organized set of modules can be painful. To enable a smooth transition, a separate module tree with a different layout would allow users to opt out of the old layout gradually. This requires that EasyBuild supports generating multiple module files for a single software installation under different naming schemes. It should also be possible to (re-)generate only a module file for an existing software installation, so that an alternative module stack can be populated without having to re-install the software itself.

VIII. Related work

We are aware of a number of projects similar to EasyBuild.

SWTools [21], [22] is an "infrastructure for software management" which was developed by National Center for Computational Sciences (NCCS) and Oak Ridge National Lab (ORNL). It defines a structure for organizing a set of (bash) scripts per supported software package, one per major installation step (configuration, build, test, installation, etc.), allowing for easily reproducing software installations. It also takes care of generating module files, and is able to update documentation listing the available software packages. Only one version is publicly available, SWTools v1.0, which was released Jan. 1st 2011.

Smithy [23] is a follow-up to SWTools, and is also being developed at NCCS/ORNL. It is compatible with the SWTools infrastructure and hence can be used as a drop-in replacement, but also supports an alternative approach using *formulas* following the well-established *Homebrew* [24] package management system for Mac OS X. As such, it provides supporting functionality readily available to be used in these formulas, and enables code reuse across formulas. Smithy is publicly available alongside detailed documentation. The development activity has slowed down significantly since Sept. 2013, with occasional changes mostly focusing on bug fixes. Smithy formulas are available for about 80 software packages.

Another related project is the *iVEC Build System (iBS)*, which is developed by iVEC in Australia. Similarly to EasyBuild it consists of a framework providing (some) commonly needed functionality, which picks up so-called "iBS files" that implement the install procedure for a particular software package. Both the main command (aptly named `ibs`) and the iBS files are bash scripts, the latter being sourced by the former as needed. At the time of writing iBS was not publicly available yet, but the developers are known to be working towards a public release.

Finally, *Spack (Supercomputing PACKage Manager)* [25] is another tool, written in Python, that is similar to EasyBuild with respect to functionality and goals. It provides a powerful and well-documented command line interface giving control over which dependencies, software versions, compiler, architecture and various options should be used for installing a particular software package. Spack also supports some particularly useful options like automatically determining whether an update for a particular software package is available (by scraping the project's website), automatically completing incomplete build specifications, optional/virtual dependencies, etc. Like EasyBuild, the Python codebase is object-oriented, enabling code reuse and efficient maintainability; the *packages* concept in it is quite similar to easyblocks in EasyBuild. At the time of writing, Spack included support for about 50 different software packages.

EasyBuild differs from these tools in a number of ways. First and foremost, it is more configurable than any of the other tools. Although they provide some support for configuring their behavior, there is no control over certain aspects like, for example, the module naming scheme being used. Second, several useful more advanced features are missing in most of them, like automatic dependency resolution (Spack being the exception here). Third, exactly reproducing previous installations is more difficult since most of these tools do not employ separate specification files (again, except for Spack); through easyconfig files this is particularly easy with EasyBuild however. Fourth, none of these tools has been able to get a sizable community going, which has significant implications with respect to user contributions and the number of supported software packages. Other issues only apply to some of them, e.g., limited code maintainability and reuse of code (SWTools and iBS), public availability of recent versions (SWTools and iBS), and a lack of active development (SWTools and Smithy).

To the best of our knowledge, no recent module tools other than Lmod and the commonly used Tcl-based tools discussed in Section II-A are available. There are some alternative tools with comparable functionality, however. Dotkit [26] is a tool that supports loading and unloading package description files in a similar fashion to modules. Softenv [27] also provides similar functionality, but uses a monolithic database to provide the package description data. Both tools use a flat layout, however, making them incompatible with the concept of a hierarchical naming scheme. Neither tool is widely used at HPC sites as opposed to the environment modules system. On top of this, both projects are no longer actively developed, with latest versions being made available in Aug. 2008 and Mar. 2007, respectively.

Several HPC sites have been using a hierarchical module naming scheme for years, including TACC [14], the Arctic University of Norway, the University of Michigan, Calcul Québec, and the University of Florida. Most of these also provide Lmod to their users, but we are unaware of any sites using both EasyBuild and Lmod for efficiently handling software installations in a hierarchical context on production systems. However, some sites are actively looking into potentially applying such a methodology in the foreseeable future, including HPC-UGent, the Jülich Supercomputing Centre, and Stanford University.

IX. Conclusion

Despite the well-established environment modules system, users of HPC systems still run into problems managing their working environments. The reasons are numerous; key issues include an overwhelming number of modules on modern systems, avoiding incompatibilities between some modules, and module tools that cannot meet users' needs and expectations. Additionally, HPC user support teams struggle to maintain a consistent set of module files, not to mention the ubiquitous problem of installing (scientific) software correctly and repeatably. Although these issues are widely recognized, suitable practices and tools have been lacking.

In this paper, we have presented a modern approach to installing scientific software that deals with these problems. We have explained the advantages of using a module hierarchy, and highlighted the need for more advanced tools to efficiently support this in a user-friendly way. We discussed in detail two actively developed and community-driven open-source projects that provide the necessary features to address this need, EasyBuild and Lmod. EasyBuild not only automates the tedious and time-consuming process of installing scientific software and the accompanying module files, it also provides full control over important aspects such as the module naming scheme being used. EasyBuild acts as a platform for collaboration between HPC sites worldwide. Lmod on the other hand allows end-users to easily navigate a hierarchically organized module stack, and delivers a variety of other useful features missing in the commonly used Tcl-based module tools.

Together, EasyBuild and Lmod allow HPC user support teams to efficiently implement and tailor a hierarchical module naming scheme that truly meets their requirements, and offers end users the simple and robust interface they deserve for managing their working environments.

References

[1] J. L. Furlani, “Providing a Flexible User Environment,” in *Proceeding of the Fifth Large Installation System Administration (LISA V*, 1991, pp. 141–152.

[2] J. L. Furlani and P. W. Osel, “Abstract yourself with Modules,” in *Proceeding of the Tenth Large Installation System Administration (LISA ’96*, 1996, pp. 193–204.

[3] D. Eadline, “Keeping It Straight: Environment Modules,” Admin HPC, http://www.admin-magazine.com/HPC/Articles/Managing-the-Build-Environment-with-Environment-Modules.

[4] J. Layton, “Environment Modules – A Great Tool for Clusters,” Admin HPC, http://www.admin-magazine.com/HPC/Articles/Environment-Module.

[5] P. F. Dubois, T. Epperly, and G. Kumfert, “Why Johnny Can’t Build,” *Computing in Science and Engineering*, vol. 5, pp. 83–88, 2003.

[6] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt, “EasyBuild: Building Software With Ease,” in *Workshop on Python for High Performance and Scientific Computing (PyHPC)*, 2012.

[7] R. McLay, “Lmod: Environmental Modules System,” http://www.tacc.utexas.edu/tacc-projects/lmod.

[8] “Environment Modules Project,” http://modules.sourceforge.net.

[9] “Distributed European Infrastructure for Supercomputing Applications,” http://en.wikipedia.org/wiki/DEISA.

[10] P. Cederqvist, K. Engtröm, H. Rindlöw, and D. Kågedal, “Cmod - Modules Done Right,” 1997, http://www.lysator.liu.se/cmod/.

[11] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, M. G. Duda, X. Huang, W. Wang, and J. G. Powers, “A description of the advanced research WRF version 3,” National Center for Atmospheric Research (NCAR), Tech. Rep. NCAR/TN-475+STR, 2008.

[12] K. Hoste and J. Timmerman, “Poll results of the ISC’14 birds-of-a-feather sesstion Getting Scientific Software Installed: Tools & Best Practices,” http://hpcugent.github.io/easybuild/files/ISC14_BoF_show-of-hands-results.pdf.

[13] O. Widder, “Geek & Poke: How To Become Invaluable,” http://geekandpoke.typepad.com/geekandpoke/2010/05/how-to-become-invaluable.html.

[14] R. McLay, K. W. Schultz, W. L. Barth, and T. Minyard, “Best Practices for the Deployment and Management of Production HPC Clusters,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[15] J. Fischer, R. Knepper, M. Standish, C. A. Stewart, R. Alvord, D. Lifka, B. Hallock, and V. Hazlewood, “Methods For Creating XSEDE Compatible Clusters,” in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, 2014, pp. 74:1–74:5.

[16] C. Condurache and I. Collier, “CernVM-FS – beyond LHC computing,” *Journal of Physics: Conference Series*, vol. 513, no. 3, p. 032020, 2014.

[17] EasyBuild, “EasyBuild documentation wiki,” https://github.com/hpcugent/easybuild/wiki.

[18] J. Layton, “Lmod – Alternative Environment Modules,” Admin HPC, http://www.admin-magazine.com/HPC/Articles/Lmod-Alternative-Environment-Modules.

[19] A. Dubrow, “Lmod: The “Secret Sauce” Behind Module Management at TACC,” http://www.tacc.utexas.edu/news/feature-stories/2012/lmod.

[20] R. Ierusalimschy, *Programming in Lua*, 3rd ed. Self-Published, 2013.

[21] “SWTools,” 2011, https://www.olcf.ornl.gov/center-projects/swtools.

[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] Anthony Di Girolamo, “Smithy,” http://anthonydigirolamo.github.io/smithy/.

[24] “Homebrew – The missing package manager for OS X,” http://brew.sh.

[25] Todd Gamblin, “Spack,” http://scalability-llnl.github.io/spack/.

[26] L. Busby, “Dotkit,” https://computing.llnl.gov/?set=jobs&page=dotkit.

[27] R. Evard and A. Bailey, “SoftEnv,” http://www.lcrc.anl.gov/info/Software/Softenv.