Spack Based Production Programming Environments on Cray Shasta

Paul Ferrell
Los Alamos National Laboratory
PRE team, HPC-ENV
Los Alamos, NM, United States
pferrell@lanl.gov

Timothy Goetsch
Los Alamos National Laboratory
PRE team, HPC-ENV
Los Alamos, NM, United States
tgoetsch@lanl.gov

Francine Lapid
Los Alamos National Laboratory
PRE team, HPC-ENV
Los Alamos, NM, United States
lapid@lanl.gov

***Abstract*—The Cray Programming Environment (CPE) provided for Cray Shasta OS based clusters provides a small but solid set of tools for developers and cluster users. The CPE includes Cray MPICH, Cray Libsci, the Cray debugging tools, and support for a range of compilers – and not much else. Users expect a wide range of additional software on these clusters, and frequently request new software outside of what's provided by the CPE or what can be provided via the system packages. Forgoing our old manual installation process, the LANL HPC Programming and Runtime Environments Team has instead opted to utilize Spack as the installation mechanism for most additional software on all of our new HPE/Cray Shasta clusters. This brings with it several distinct advantages - Spack's vast library of package recipes, well-defined software inventories, automatically-generated module files, and binary packages produced through our CI infrastructure. It also brings with it substantial issues – remarkably higher manpower requirements, longer turnaround times for software requests, a more challenging build debug process, and questionable long-term maintainability. Our paper will detail our approach and the benefits and pitfalls of using Spack to install and maintain production software environments.**

***Index Terms*—programming environments, spack, cray**

I. Introduction

Los Alamos National Laboratory is one of the premier scientific computing centers in the United States. We are typically the host to at least a dozen production clusters (currently 6, not including test-beds) with thousands of nodes combined (currently 10k nodes). The clusters tend to be of a variety of architectures and operation systems, though they are currently primarily HPE/Cray Operating System 2 (COS2/Shasta[1]) based clusters.

The LANL High Performance Computing (HPC) Programming and Runtime Environments Team (PRE Team) is responsible for supporting both the vendor provided programming environment (PE) and internally installed software on our clusters. The COS2 clusters come with releases of the Cray Programming Environment (CPE), which provides Cray compilers, Cray MPICH, Cray debugging tools, HDF5 and NetCDF libraries, support for additional compilers (Intel or AMD vendor compilers, GNU), and a libfabric that interfaces with the proprietary Cray Slingshot based high speed network. While it's a solid foundation, the CPE is only a fraction of the software we need to provide our users. Visulaization tools such as Paraview, basic compilation tools such as cmake, and even utilities like LaTeXare expected as well.

Some of needed packages can be provided through system RPM installs. Yet users often expect multiple versions or versions newer than those provided through existing RPMs. In such cases the software is installed on shared file systems in user-space and made available through a module system — Lua based LMod modules being our current preference. Additionally, any libraries we provide must be maintained for the life of the cluster. It's expected that code compiled on the first day of system life still be usable by the end-of-life of the cluster.[2] All of this software must be built to support each supported compiler (Intel OneAPI, classic Intel, GNU, CCE), as well as the most recent MPI versions on each cluster[3].

Our legacy system for accomplishing this was very manual and dependent upon subject matter experts on our team. Each supported piece of software had an 'owner' on the team, who developed their own set of scripts to capture the knowledge of how that software needed to be installed and automate the process. These scripts varied greatly from person to person in complexity and programming language. This was installed in a sensible hierarchy based on cluster OS and host, with 'common' areas for software with general compatibility[4]. Module files were hand-written and expected to dynamically handle all versions and installations of a given piece of software. This central library of module files was not our modulefile hierarchy - that was made up of a separate structure of symlinks into the library. These symlinks were managed by some simple scripts. The sum of this made fixing module files issues fairly easy, and adding/removing module files trivial.

The legacy PE had several major drawbacks. The spe-

Identify applicable funding agency here. If none, delete this.

[1]We often work with vendor technologies very early in their life cycle, and adopt the pre-release names internally. 'Shasta' is an example of this. Even internally the name used varies per-person. It's confusing!

[2]This isn't quite the case anymore - we expect major OS upgrades to break this compatibility. It's still a good habit though, so we pretend it's mostly true.

[3]We don't have to build against every compiler/MPI combination - just the 'leading edge' of the latest versions.

[4]Clusters for the most part shared a common architecture, but differences in high speed networks resulted in MPI library incompatibilities

cialized knowledge that went into each package and their installation scripts meant that typically only the author of those scripts could install new versions in a timely manner. The installation areas for each package were purely the domain of the owner, and were littered with partial and broken installations. No inventory governed what should be installed, or made it apparent what was missing on each system. The unified, per-package module files grew extremely complex and fragile as cluster and version differences accumulated. As a result of these problems, when users requested new software, the answer was some combination of 'no' and a funding request - after all someone would have to devote significant work-hours to figuring out dependencies, writing scripts and module files. The result was a software environment that was restrictive, messy, fragile and did not meet the changing needs of our users. It was clear that something needed to change.

A. Our Requirements, and How Spack [1] Fulfilled Them

Any new software deployment workflow we put into production needed to address the systemic issues with our legacy deployments.

1) Software Installs - Should be able to build and deploy most of our needed software, including missing system dependencies.
2) Inventory - Allow for the definition of a concise inventory of packages that should be installed.
3) Reproducible - We should be able install reproducibly and idempotently.
4) Combinatorial Installs - Support the installation of multiple, conflicting versions of the same package built with a variety of compilers.
5) Module Files - Automatically generate module files compatible with system and user module trees.
6) Efficiency - Reduce total install size and provide reusable *binary packages*[5] that we can reuse and share with users.
7) Reduce Manpower - Reduce the work-hours required to maintain the software environment.

B. Spack

Spack is a Lawrance Livermore National Laboratory (LLNL) developed system designed to resolve the complex dependency and installation issues that often arise with HPC software. In the years since it debuted, it has grown to be fairly robust, well-supported, and widely accepted by the scientific computing community.

Spack functions by providing a large library (*Spack repo*) of package build instructions in various configurations (*variants*), a language (*Spack specs*) for describing how you want those packages installed, and a system (the Spack *concretizer*) for solving the dependency tree. Once the dependency tree resolved, Spack knows exactly which packages to install, how to install them. and even how to create module files for them.

[5]This is Spack's term for it's RPM like format of pre-compiled packages.

Spack came with another benefit - promoting TriLab[6] collaboration. We were assured that Spack could meet most of our requirements by LLNL Spack devs, and were promised development support in areas where Spack couldn't. This led to the TriLab Computing Environment 2 (TCE2) project — a TriLab project to produce, if not a unified programming environment, common mechanisms (IE Spack) that we could share in producing PEs. Much of that effort focused on the development of LANL HPC's Spack-based PE, with the Spack developers on hand to expedite fixes for Spack issues that prevented progress. The arrangement worked well. After a few years (and a pandemic), we installed our first releases of TCE2 at LANL in the Spring of 2022.

II. TCE2

This first release of TCE2 used Spack to reproduce everything we had installed in our production software environments on both our TOSS 3[7] and our Cray CLE7 based clusters[8]. Our process was well structured: A git repo contained most of the components - one simply had to check it out and run the contained `process.sh`[9] script to install Spack, build/install all of the programming environment packages, and generate a modulefile tree in any desired location.

The core of this repo is the various Spack environment definitions. These are separated by architecture compatibility parameters - `<os_base>-<os>-<arch>-<net>-<gpu>` - since the contents and compatibility can vary widely based on these parameters[10]. System dependencies have different versions and paths. Different architectures may require different package configurations. Different high-speed network drivers yield incompatible MPI builds. Lastly, the presence and type of GPU's often requires an entire ecosystem of packages that aren't present on other systems. This allows us to build a single, common environment for clusters that are largely the same, while taking advantage of Spack's ability to reuse identical builds when there is overlap between signicantly different clusters.

Spack packages describe how to build and install each software package, and are collected into Spack repos - mainly Spack's internal `builtin` repository. Most of these can be used out-of-the box, but many require tweaks for local issues or simply have bugs in their Python code. TCE2 keeps an internal package repo[11] with such fixes. Any fixes we make are also submitted to Spack as merge requests, so that we can eventually revert back to Spack's builtin package definitions whenever we upgrade Spack.

The final major component is the facilities directory. This contains subdirectories named for each installation facility (IE LANL, Sandia, etc) meant to contain site specific files for

[6]IE - LANL, LLNL, Sandia

[7]TriLab Operating System Stack

[8]Such as Trinity, our now decommisioned 20k node cluster

[9]The name `process.sh` remains inexplicable to this day

[10]Most of our systems currently share an environment tagged `cray-sles15-x86_64_v3-slingshot-none`

[11]Originally named 'spacklemore'...

the deployment of TCE2. This includes license files for our licensed products, general Spack configs (to be installed in `etc/spack`), and a configuration script to set defaults for `process.sh`. Keeping site specific information separate not only makes it easier for other sites to install TCE2, it allows us to handle differences between networks cleanly.

All of these components come together whenever we build TCE2. A typical installation command looks like

```
./bin/process.sh
    -e cray-sles15-x86\_64\_v3-none
    -f lanl -i /tmp/my-tce-install
```

The completed install directory contains:

- A dedicated Spack install.
- A Spack environment with the configs for all of our installs.
- A log directory with logs of each install/update attempt.
- A module files directory.
- All of the Spack installs themselves (the install target is this directory).

All that remains is to add the module files to the `MODULEPATH` - we provide a 'lanlpe' modulefile that's loaded by default. The install process is idempotent, and updates should add to the installs without breaking existing links.

A. Binary Packages, CI, and Testing

Each TCE2 environment's `spack.yaml` file allows us to define exactly what should be installed. Spack's binary package generation feature let's us pre-build relocatable packages so that when we we install in production, we do so *reproducibly* without every having to build anything new. This reproducibility is important to us - if anything ever happened to break our production PE, we need to be able to recreate it perfectly to preserve linking in user applications. [12]

When we check in an updated version of TCE2, our GitLab runners go through a fairly typical series of stages (See 1). The Syntax checks ensure that our YAML and bash scripts are consistently formatted. Sanity Checks make sure that we haven't introduced anything into the Spack configs that shouldn't be there - like absolute paths (everything should be relative to the Spack directory). If any changes to scripts are made, we build and install a tiny PE in the operability tests to look for bugs in `process.sh`. The concretization checks look at what packages Spack intends to install, and verifies that it matches what we prescribed.

Finally, a CI job builds the entire environment, creates binary packages for all new installs and uploads them to our binary cache. The install step only takes around twenty minutes, as most of the packages are simply downloaded and installed from the binary cache. The building of binary packages always runs regardless of whether the install step completely succeeds - any new packages successfully built are uploaded. We rely on Spack's ability to differentiate slightly different builds to avoid installing incorrect or broken binary packages built this way, though we do occasionally have to manually remove a bad package. Once CI passes, we can install the entire environment purely from the Spack binary cache using `process.sh --cache-only`.

[12]This requirement even includes packages that are no longer exposed to the user through module files.

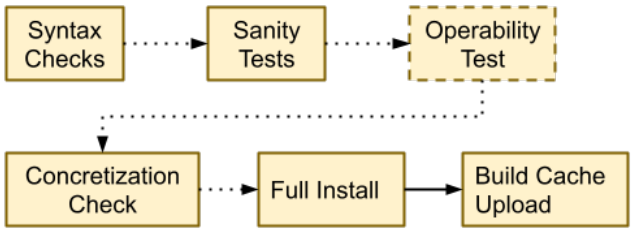

Fig. 1. TCE2 Gitlab CI steps

This methodology works fairly well. We've rolled out completely new environments, and have provided updates to existing environments.

III. Problems Become Concrete

Our initial roll-out of TCE2 happened in Spring 2022 as a 'preview'. Our first official release of TCE2 as the default environment on a cluster didn't occur until a year and a half later on our new Cray clusters, and TCE2 for TOSS3 was abandoned. The intervening time contained some turmoil. The main TCE2 architect left LANL. It was decided at one point to abandon the whole concept as unmaintainable. It was also decided that we simply didn't have time to find an alternative before our new generation of clusters came online. While Spack met all of our requirements for software environment management in theory, it was found that the reality of implementing them through Spack was often excrutiating.

A. Software Installs

A huge advantage of Spack is it's very large library of ready to install package definitions in its `builtin` package repo. This provided packages for almost everything we already installed in our legacy environment, and given basic Python knowledge it's fairly easy to fill those gaps. Spack's general acceptance in the wider community gives regular updates to those packages, and package variants mean we install those packages configured exactly as we needed with minimal effort. We could throw out all our home-grown scripts and rely on shared, open source efforts.

This is what Spack is *phenomenal* at, in our opinion. Using Spack to install just about any package in its library just works, most of the time. Spack *specs* are well thought out, the package definitions are flexible and powerful, and how Spack wraps the build process for different build systems works exceptionally well.

It's in the failure modes where problems emerge. Spack really doesn't provide the tools or documentation needed for handling build (or most other) errors. The best way we've found to handle build failures is to go into the appropriate

`spack-stage` directory for the package being built[13], which contains everything Spack was using to perform the build. There you can recreate the Spack build environment using Spack's saved files, and directly run the configure and build commands. It works about 95% of the time - the problem is almost always a misconfiguration in the Spack package or one of it's dependencies. From there one can fix those packages directly, or create a fixed version in their own Spack package repo. For our environment that builds several hundred packages, we have about two dozen such fixes, and have merged many more with into Spack.

Spack provides a `build-env` command that seems designed to help (`spack build-env <spec> bash` should recreate alot of the steps above), but it shares a common problem with many Spack commands. While it's easy to specify a simple spec as the target of the command, specs for our production environments are a complicated combination of configurations for both itself and its dependencies. It's not just difficult to put that on the command line, we don't know of a way to extract those full specs at all[14]. This goes for commands as integral to Spack as `spack install` as well. Spack was designed for the command line use case, and the hand-written environment use case is not well supported.

B. Inventory and Reproducibility

Spack provides the *environment* system for creating a well defined set of packages and configurations to install together. The expected way of working with this system is via the `spack env` command, but we were encouraged to edit the `spack.yaml` file this command produces directly. It's a yaml file under which you can define a list of specs to install, and additionally add just about any sort of configuration Spack expects.

The `spack.yaml` for each of our Cray environment is about 1400 lines long. Defining which packages to install is fairly concise. Making sure Spack installs these packages consistently and reproducibly is not concise. The root of the problem is Spack's *concretization* of those packages.

1) Concretization Overview: Concretization is the process by which Spack solves the dependency tree problem. For Spack, this is more complicated than with a traditional package manager. RPM and DEB repos have a set version for each package that's tied to the system release. They're able to maintain ABI compatibility by only doing minor updates to each package over its lifetime. Options for packages are fairly set in stone, and they have a clear delineation based on OS and system architecture. Spack has none of these luxuries.

C++ and especially Fortran libraries are not necessarily compatible when built with different compilers. HPC software often depends on MPI, which in turn directly depends on the cluster fabric drivers. Packages depend on options in other packages that may or may not be enabled, and multiple packages can share a dependency yet require conflicting options for that dependency. Resolving the dependency tree and configuration settings is many degrees more complex, and can require significant computation time to solve.

Under Spack's concretizer every package requested is *concretized* to a hash based on every detail of how Spack intends to build the package AND all of its dependencies. Spack's original concretizer worked purely on these hashes. The slightest change in a package spec would cascade up the tree, requiring rebuilds of every package that depended on it. Spack's new 'Clingo' based concretizer composes a massive declarative logic program[15] based on the requested software, dependency and variant information, and then using the Clingo solver to find a 'best' solution to it. This new logic can now differentiate between a change that won't effect a parent package and one that will, making reuse much more flexible.

C. Concretizing TCE2

TCE2's initial release was under the old concretizer. Error messages under the old concretizer were obscure, but we could typically track them down and debug in Spack's source. Clingo errors, in contrast, are there result of a failed solve of its massive machine generate logic program. These errors often far removed from their actual cause, and nigh impossible to track down directly. The trial and error involved to solve these issues is exacerbated by the fact that Clingo takes upwards of 30 minutes to solve the dependency tree for our PE.

The killer feature of Clingo is it's ability more more flexibly reuse built packages. Concretization under Clingo depends on not only what Spack intends to build, but also what packages are already available in the binary cache. The result is that we can verify that our PE builds, only for it to fail on install because Spack will choose different answers and generate a dependency tree with requires new builds. Clingo will come up with a different solution than it did originally based on what is in the binary cache. To get around this problem, we turn reuse off entirely.

Inconsistency is our core problem with concretization. For every single package built in the entire dependency tree, we set a version in our Spack environment definition. The defaults set by packages aren't enough to keep the versions of dependencies from wandering with the slightest change in configuration. To keep Spack from producing largely duplicate and unnecessary builds, we also often have to force dependencies into a set of variants state that satisfy the requirements for all parents.[16] The concretization CI step runs scripts the checks for these issues, ensuring that we don't end up with unnecessary builds, and verifying that we only build versions we explicitly enabled. We've developed additional tools to look at the concretization results and hunt down the reasons for these differences, tooling that Spack does not itself provide. Locking down versions and variants accounts for about 85% of our `spack.yaml` content.

[13] It's in `/tmp/<usr>/spack-stage`.

[14] After a package is built, the `spack find` command can help

[15] It's megabytes of plain text.

[16] Preventing this from happening is what the Spack concretizer is for, in theory.

D. Combinatorial Installs

When we started down the road of using Spack, Spack environments only supported one instance of each package per environment. We quickly made it clear to the Spack developers that a large part of the problem with production programming environments was that we often had to install multiple versions of each package built against multiple compilers and MPI libraries. The Spack developers fairly quickly added features to support this issue, and supported us as we worked through issues we found with those features as we developed TCE2.

The only remaining problem in this space is fairly new - `extends(python)`. In an effort to more sanely support Python libraries[17], Spack added a feature to setup packages as extensions of other packages, particularly Python. This again restricts some packages to only one instance per environment. Anytime we have to install multiple instances of a package that extends Python, we have to override the entire package just to comment out this feature. There's currently no way to disable it globally or through configuration.

E. Module Files

Spack can generate a modulefile tree for either Environment Modules (TMod) or LMod. This process is highly configurable. You can select which packages to expose, extend the module files with extra environment variables, or even change the base modulefile template. One can even generate multiple, special purpose module files directories (such as per-compiler/MPI trees). This is a significant improvement over our old manual process.

Configuration of this is cumbersome. We have to specify both which packages to expose and which packages to white-list in our configuration. The duplicated configuration is error prone and unnecessarily long.

More importantly, Spack module files don't play well when combining Spack environments. We provide our Spack produced module files and so do code teams, and they need to be able to work in tandem. When loading a module from our tree with dependent module files, Spack module files autoload any modules they depend on. This often *unloads* modules from the code team's tree, breaking their environment. This autoloading can be disabled in Spack, but now our loaded module won't have access to its needed dependencies. The problem is in the `extends(python)` feature. For all link level dependencies, Spack RPATH's to them. For Python, however, it relies the currently loaded Python to have them in its site-packages directory. Between the two environments we have multiple conflicting Pythons with different sets of dependencies. Without `extends(python)` this would work. [18]

IV. Spack Versions

Further discussion of Spack and TCE2's use of it requires addressing a very fundamental problem with Spack — the unstable nature of Spack versions and the packages included with it. Spack's base use case of performing a `spack install somepackage` works fine regardless of version. If we have an ecosystem of dozens of packages with their likely hundreds of dependencies, that's far more likely to run into substantial problems when one tries to upgrade Spack versions. The whole structure probably won't concretize. Changes to the builtin packages might change the meaning or format of package variants, other package changes will prevent many of them from building. We're currently in the process of upgrading from Spack 0.19 to 0.21. Three people have dedicated almost a month the upgrade so far. I've heard similar stories from code teams and other Spack users - everyone sticks with a single version of Spack for as long as possible because upgrades are painful.

This had particularly sobering consequences for TCE2. Upgrading often results in changes to most packages in Spack, requiring a rebuilding almost the entire software stack. Package reuse helps, but a rebuild of a dependency forced by package changes would still propagate up the dependency tree and require substantial rebuilds across the environment[19]. If we intended to grow our environment over time by adding new compilers, MPI libraries, and additional packages we would have to contend with ever increasing Spack upgrade complexity alongside longer and longer concretization times. It was deemed untenable, and largely why we almost abandoned Spack.

A. Compatibility

Our hope with Spack was that we could chain our installs through *upstreaming*[20] to reduce the installed size of the PE, reuse builds through binary packages, and share our library of binary packages with code teams to reduce their build times. Only one of these things has come to fruition - building and reusing our own packages has worked wonderfully. It works largely because we're only consuming what we produce and we can guarantee that we're using a version of Spack that will be happy with those packages.

Upstreaming turned out to be something we simply couldn't use. Mostly importantly, it breaks modulefile generation - you can't make module files for upstreamed packages. We could only use upstreamed packages for which we didn't create module files, but Spack doesn't give control over which packages you get from upstream and which you don't. We also ran into a lot of compatibility issues when trying to upstream as well, especially in regards to custom Spack package repos. In the end, we found upstreaming to be generally unusable for our purposes.

In contrast, Spack's binary packages are a huge leap forward for package managers. Unlike any other package manager we know of, they are relocatable by default. Unlike RPMs, we can build packages without regard for their final install location.

[17]The prior solution required separately loading python packages and creating a very long PYTHONPATH

[18]There are solutions to this, but it requires manually configuring additional, complex Spack views and adding those views to the dependent packages PYTHONPATH.

[19]This happens more frequently than one would expect, as package variants and dependency requirements change.

[20]Upstreaming is the process of pointing at an installed set of packages from another Spack instance.

There were initially issues installing packages into paths that were longer than the original install path, but our CI processes ensure that isn't an issue.[21] We've had to suddenly change where we install TCE2, and relocatable packages made that a non-issue.

Binary packages should also allow us to share those packages with users and code teams. Unfortunately, that's not something we can do with our environment as it stands. Our TCE2 deployments use a patched 0.19 version of Spack that can build against architectures that aren't supported in that release. We also have a large number of patched and overridden packages, which could lead to unexpected behavior for users. Additionally, users with older versions of Spack (which is common), will likely have compatibility problems with binary cache itself. [22]

B. Efficiency

When we started TCE2, we expected the gains in productivity to allow us time to become active Spack developers, as well as give us more time to spend on internal projects. That has not happened. Getting new team members full up-to-speed on Spack takes months. We've almost certainly spent multiple combined years debugging concretization, package misconfigurations, and package build problems. As mentioned, several team members have spent months just trying to upgrade our Spack version.

Despite Spack version upgrades being difficult and time consuming, smaller upgrades to packages in our PE go quickly. Backporting a package version, adding it to our config, and letting CI handle the builds is rather efficient. The team is no longer limited to the package expert - we're all Spack experts instead. With some major changes in how we think about TCE2 we were able to make it into a system that we think will work long term, and is finally on the cusp of yielding the productivity gains we hoped for.

V. TCE2 on Cray Shasta/COS2

We mentioned in the prior section that we almost gave up on Spack. We wrote an internal paper outlining why TCE2 simply would not work long term - largely because it would grow in size and complexity until it was unmaintainable. After some thought, a separate proposal was written - how to change the base concepts of TCE2 into a supportable infrastructure. The key to this was our new clusters, and the release model of the HPE/Cray programming environment.

A. The Upgrade Problem

The time and effort it took to get our initial TOSS 3 based TCE release into production was substantial. Figuring out how to coax Spack into concretizing and building our environment ran into all of the issues mentioned in the prior section and quite a few more that were fixed along the way. In order to alleviate that strain, we intended on installing TCE2 as independent releases in a shared Spack install area. Most packages could be reused, we wouldn't be building one massive concretization problem, and the cycle would give us the time needed to get new versions working and tested. We were assured that this would work just fine, that newer Spack versions would happily build (or at least reinstall the binary packages for) the old environment.[23]

This is when we discovered the depth of the Spack upgrade problem. Separating into separate releases might make the concretization problem smaller, but every Spack upgrade would break both the new *and* old TCE2 releases. We were stuck. We could live with ever increasing concretization, debug, and install times as a monolithic TCE2 grew, or we could debug breaks in old releases at every Spack upgrade.

With Cray Shasta systems, there was a shortcut. Since Cray provides the base environment in releases anyway, we simply match our releases to theirs - supporting only that CPE release in an entirely independent release. Each release would be installed with its own Spack and install directories. Old releases now *never* need a Spack version upgrade. The entire release can always be installed reproducibly - simply because nothing changes outside of our direct, configured control. New packages could be added to these environments in-place, since these environments weren't encumbered with a complicated history. From the user perspective, the latest TCE2 release is autoloaded on login to match the default CPE packages.

B. CPE Problems

The Cray Programming Environment has itself proven problematic in regards to Spack. Spack doesn't quite correctly find compilers on the system, so we provide and rely on a static `compilers.yaml`. We use hierarchical LMod with the Cray CPE[24], which confuses Spack's ability to include external software by module. They must be explicitly defined by spec (IE `cray-mpich@8.1.14%intel`) and path instead. There were also issues integrating Spack's module files with the Cray PE. We currently use LMod `inherit` statements to dynamically patch in TCE2 module paths according to the loaded compiler and MPI version. These were all minor, solvable issues in the grand scheme of things.

Our most challenging problem with the Cray CPE is the release cycle. A problem reported to HPE/Cray, even if fixed immediately, may only be available in a CPE released months later. That same CPE and related system patches might not make it to clusters for further months due to the Cray System Management (CSM) upgrade schedule. The closed source nature of cray-mpich exacerbates this. Regardless of all our local MPI expertise, little can be done to hunt down the roots

[21] Also, we think it's fixed in newer versions of `patchelf`.

[22] It has been recommended that we build out a binary cache for users using the oldest version of Spack amongst all our users. This would guarantee that they could read from our binary package cache, and use any created packages. That leaves us with the problem of backporting any newer packages that the users want, and 'reversing' any changes that aren't supported by our older Spack release.

[23] These separate releases would be combined (with a new Spack feature that never quite came) into a signal modulefile tree.

[24] LMod configured such that loading a module expands `MODULEPATH` to include additional module files that depend on the now loaded module. This greatly simplifies module composition.

of these problems without the source. With any new cluster platform it's expected for there to be issues, but this cycle prevents those issues from being fixed in a timely manner.

VI. Into the Future

While our TCE2 software deployment process is stable, upgradable, and maintainable in the long term, there are still several problems to solve. The lack of a dedicated development environment is at the top of this list. Our releases our tied to Cray CPE releases, yet we don't have a system where we can perform builds against CPE's other than our clusters themselves. Our CI workflows currently build on cluster head nodes. Moving our CI jobs into containers is the obvious solution, but a COS2 container isn't enough. We must also develop a process to assemble Cray PE releases to attach to that container. All the parts are available to us to accomplish this, and we expect to have it operational this year.

Testing is another sticking point. A set of software tests were developed for for the initial TCE2 release using Pavilion [2], LANL's HPC testing framework. These tests provided a quick verification that each module was loadable, and that the software it provided worked to some degree. These tests have not been maintained, however, and need to be made into a required step in our CI pipelines. We also need to add a check for the tests themselves, ensuring that a test exists for each modulefile that TCE2 adds.

Spack has produced some improvements to managing environments that we can take advantage of once we upgrade to a more recent version. The `spack.lock` file captures the concretized state of environments, but has always been ephemeral. New changes allow for the reuse of concretization solutions captured in that file. By adding a CI step that commits the `spack.lock` file back to the repo if all other tests pass, we hope to greatly shorten the concretization times for our environments. Most importantly, this should resolve our issues with Spack package reuse inconsistencies.

As of this writing we deploy changes to TCE on a two week cycle, deploying the latest collection of built packages by hand using `process.sh`. After testing the environment and saving the concretization state, there's no reason why we can't automatically deploy those updates in production using runners on the clusters themselves. User software requests would be resolvable in hours instead of weeks.

VII. Conclusion

Over the course of this paper, we've discussed a subset of the issues we've discovered in building a software environment deployment workflow with Spack. While these issues remain, it's also clear that we've been successful in using Spack to create a functional software deployment workflow with tangible benefits. The question that remains is: "Was it worth it?"

In some ways, it was clearly not. For the amount of work we put into getting Spack to deploy our existing environment, we could have easily built an alternative solution that more simply and directly solved our software deployment problems. Our existing deployments were built around having minimal dependencies. Spack solves a very hard problem of dependency management that we simply didn't have. While Spack has improved in ways that have made it possible to create our software deployments, the tooling around long term management of these deployments remains insufficient. Many of the benefits we expected Spack to yield did not come about. Our generated binary packages are not easily shared with code teams. The 'inventory' of packages is unwieldy and difficult to maintain. Working with Spack frequently consumes far to much employee time, and we expect Spack upgrades to continue to be a major pain point.

This project has also yielded great successes. TCE2 has put everyone on our team on equal footing - everyone is experienced at debugging a variety of Spack concretization and build issues. Any member of our team can now fulfill user requests for software that we would never have attempted before. Spack has enabled us to transition from a user programming environment that was static, rigid and stale to one that can flourish and grow with the needs of our users. This could not have come at a better time. The PE requirements for our new Cray clusters are requiring that we substantially expand the libraries of software we provide. Our modernized, self-testing, automated TCE2 build infrastructure is essential for providing software support for this new generation of clusters, and Spack is the heart of it.

Acknowledgments

A massive amount of credit goes to Nick Sly, the originally architect of TCE2. He spent years getting our initial TCE2 environment working for TOSS 3 and CLE7. Nick is the greatest.

The Spack developers at LLNL also deserve a gracious shout-out. While this paper is critical of Spack in this use case, that does not undermine the achievement of Spack as a whole. Greg Becker in particular helped us overcome many of the early hurdles that made this work possible.

Jennifer Green was our team leader through most of this work. While she's moved on to other opportunities, we could not have gotten to this point without her guidance and her uncanny ability to put together the great team we have today.

References

References

[1] LLNL (2024, April 29) Spack Documentation, `https://spack.readthedocs.io/en/latest/`

[2] LANL (2024, April 29) Pavilion Documentation, `https://pavilion2.readthedocs.io/en/latest/`