The Spack Package Manager: Bringing Order to HPC Software Chaos

Todd Gamblin
tgamblin@llnl.gov

Matthew LeGendre
legendre1@llnl.gov

Michael R. Collette
mcollette@llnl.gov

Gregory L. Lee
lee218@llnl.gov

Adam Moody
moody20@llnl.gov

Bronis R. de Supinski
bronis@llnl.gov

Scott Futral
futral@llnl.gov

Lawrence Livermore National Laboratory

ABSTRACT

Large HPC centers spend considerable time supporting software for thousands of users, but the complexity of HPC software is quickly outpacing the capabilities of existing software management tools. Scientific applications require specific versions of compilers, MPI, and other dependency libraries, so using a single, standard software stack is infeasible. However, managing many configurations is difficult because the configuration space is combinatorial in size.

We introduce Spack, a tool used at Lawrence Livermore National Laboratory to manage this complexity. Spack provides a novel, recursive specification syntax to invoke parametric builds of packages and dependencies. It allows any number of builds to coexist on the same system, and it ensures that installed packages can find their dependencies, *regardless of the environment*. We show through real-world use cases that Spack supports diverse and demanding applications, bringing order to HPC software chaos.

1. INTRODUCTION

The Livermore Computing (LC) facility at Lawrence Livermore National Laboratory (LLNL) supports around 2,500 users on 25 different clusters, ranging in size from a 1.6 teraflop, 256-core cluster to the 20 petaflop, 1.6 million-core Sequoia machine, currently ranked second on the Graph500 [4] and third on the Top500 [30]. The simulation software that runs on these machines is very complex; some codes depend on specific versions of over 70 dependency libraries. They require specific compilers, build options and Message Passing Interface (MPI) implementations to achieve the best performance, and users may run several different codes in the same environment as part of larger scientific workflows.

To support the diverse needs of applications, system administrators and developers frequently build, install, and support many different configurations of math and physics libraries, as well as other software. Frequently, applications must be rebuilt to fix bugs and to support new versions of the operating system (OS), MPI implementation, compiler, and other dependencies. Unfortunately, building scientific software is notoriously complex, with immature build systems that are difficult to adapt to new machines [13, 23, 42]. Worse, the space of required builds grows combinatorially with each new configuration parameter. As a result, LLNL staff spend countless hours dealing with build and deployment issues.

Existing package management tools automate parts of the build process [2, 10, 11, 12, 23, 24, 38, 39, 41]. For the most part, they focus on keeping a single, stable set of packages up to date, and they do not handle installation of multiple versions or configurations. Those that *do* handle multiple configurations typically require that package files be created for each combination of options [10, 11, 12, 23], leading to a profusion of files and maintenance issues. Some allow limited forms of composition [11, 12, 23], but their dependency management is overly rigid, and they burden users with combinatorial naming and versioning problems.

This paper describes our experiences with the *Spack* package manager, which we have developed at LLNL to manage increasing software complexity. It makes the following contributions:

1. A novel, recursive syntax for concisely specifying constraints within the large parameter space of HPC packages;
2. A build methodology that ensures packages find their dependencies regardless of users' environments;
3. Spack: An implementation of these concepts; and
4. Four use cases detailing LLNL's use of Spack in production.

Our use cases highlight Spack's ability to manage complex software. Spack supports rapid composition of package configurations, management of Python installations, and site-specific build policies. It automates 36 different build configurations of an LLNL production code with 46 dependencies. Despite this complexity, Spack's *concretization* algorithm for managing constraints runs in seconds, even for large packages. Spack's install environment incurs only around 10% build-time overhead compared to a native install.

Spack solves software problems that are pervasive at large, multi-user HPC centers, and our experiences are relevant to the full range of HPC facilities. Spack improves operational efficiency by simplifying the build and deployment of bleeding-edge scientific software.

2. COMMON PRACTICE

Meta-build Systems.

Meta-build systems such as Contractor, WAF, and MixDown [3, 15, 16, 32] are related to package managers, but they focus on ensuring that a single package builds with its dependencies. MixDown notably provides excellent features for ensuring consistent compiler flags in a build. However, these systems do not provide facilities to manage large package repositories or combinatorial versioning.

SC '15, November 15-20, 2015, Austin, TX, USA. LLNL-CONF-675189.
© 2015 ACM. ISBN 978-1-4503-3723-6/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807623

Traditional Package Managers.

Package managers automate the installation of complex sets of software packages. *Binary package managers* such as RPM, yum, and APT [18, 33, 36] are integrated with most OS distributions, and they are used to ensure that dependencies are installed before packages that require them. Anaconda [7, 8], uses the same approach but is designed to run on top of a host OS. These tools largely solve the problem of managing a *single* software stack, which works well for the baseline OS and drivers, which are common to all applications on a system. These tools assume that each package has only a single version, and most install packages in a single, inflexible location. To install multiple configurations, users must create custom, combinatorial naming schemes to avoid conflicts. They typically require root privileges and do not optimize for specific hardware.

Port systems such as Gentoo, BSD Ports, MacPorts, and Homebrew [22, 24, 38, 39, 41] build packages from source instead of installing from a pre-built binary. Most port systems suffer from the same versioning and naming issues as traditional package managers. Some allow multiple versions to be installed in the same prefix [22], but again the burden is on package creators to manage conflicts. This burden effectively restricts installations to a few configurations.

Virtual Machines and Containers.

Packaging problems arise in HPC because a supercomputer's hardware, OS, and file system are shared by many users with different requirements. The classic solution to this problem is to use virtual machines (VMs) [5, 35, 37] or lightweight virtualization techniques like Linux containers [17, 29]. This model allows each user to have a personalized environment with its own package manager, and it has been very successful for servers at cloud data centers. VMs typically have near-native compute performance but low-level HPC network drivers still exhibit major performance issues. VMs are not well supported on many non-Linux operating systems, an issue for the lightweight kernels of bleeding-edge Blue Gene/Q and Cray machines. Finally, each VM still uses a traditional package manager, so running many configurations still requires a large number of VMs. For facilities, this profusion of VMs is a security concern because it complicates mandatory patching. Also, managing a large number of VM environments is tedious for users.

Manual and Semi-automated Installation.

To cope with software diversity, many HPC sites use a combination of existing package managers and either manual or semi-automated installation. For the baseline OS, many sites maintain traditional binary packages using the vendor's package manager. LLNL maintains a Linux distribution, CHAOS [26] for this purpose, which is managed using RPM. The popular ROCKS [40] cluster distribution uses RPM and Anaconda in a similar fashion. For custom builds, many sites adhere to detailed naming conventions that encode information in file system paths. Table 1 shows several sites' conventions. LLNL uses the APT package manager for installs in the `/usr/local/tools` file system and `/usr/global/tools` for manual installs. Oak Ridge National Laboratory (ORNL) uses hand installs but adheres to strict scripting conventions to reproduce each build [25]. The Texas Advanced Computing Center (TACC) relies heavily on locally maintained RPMs.

From the conventions in Table 1, we see that most sites use some combination of architecture, compiler version, package name, package version, and a custom (up to the author, sometimes encoded) build identifier. TACC and many other sites also explicitly include the MPI version in the path. MPI is explicitly called out because it is one of the most common software packages for HPC. However, it is only one of many dependencies that go into a build. None of these naming conventions covers the entire configuration space, and none has a way to represent, e.g., two builds that are identical save for the version of a particular dependency library. In our experience at LLNL, naming conventions like these have not succeeded because users want more configurations than we can represent with a practical directory hierarchy. Staff frequently install nonconforming packages in nonstandard locations with ambiguous names.

Environment Modules and RPATHs.

Diverse software versions not only present problems for build and installation; they also complicate the runtime environment. When launched, an executable must determine the location of its dependency libraries, or it will not run. Even worse, it may find the wrong dependencies and subtly produce incorrect results. Statically linked binaries do not have this issue, but modern operating systems make extensive use of dynamic linking. By default, the dynamic loader on most systems is configured to search only system library paths such as `/lib`, `/usr/lib`, and `/usr/local/lib`. If binaries are installed in other locations, the *user* who runs the program must typically add dependency library paths to `LD_LIBRARY_PATH` (or a similar environment variable) so that the loader can find them. Often, the user is not the same person who installed the library, and even advanced users may have difficulty determining which paths to add.

Many HPC sites address this problem using *environment modules*, which allow users to "load" and "unload" such settings dynamically using simple commands. Environment modules emerged in 1991, and there are many implementations [6, 19, 20, 27, 28]. The most advanced of these, Lmod [27, 28], provides software hierarchies that are similar to the naming conventions in Table 1 and allow users to load a software stack quickly if they know which one is required.

The alternative to per-user environment settings is to embed library search paths in installed binaries at compile time. When set this way, the search path is called an `RPATH`. `RPATH`s and environment modules are not mutually exclusive. Modules can still be used to set variables that are unrelated to linking, such as `MANPATH` and `PATH`. Adding `RPATH`s still ensures that binaries run correctly, regardless of whether the right module is loaded. LC installs software with both `RPATH`s and `dotkit` [6] modules.

Modern Package Managers.

Recently, a number of HPC package managers have emerged that manage multi-configuration builds. ORNL uses the Smithy [10] installation tool. It can generate module files, but it does not provide any automated dependency management; it only checks whether a package's prerequisites have already been installed by the user.

The Nix [11, 12] package manager and OS distribution supports installation of arbitrarily many software configurations. As at most HPC sites, it installs each package in a unique prefix but it does not have a human-readable naming convention. Instead, Nix determines the prefix by hashing the package file and its dependencies.

The EasyBuild [23] tool is in production use at the University of Ghent and the Jülich Supercomputing Center. It allows multiple versions to be installed at once. Rather than setting `RPATH`s, it generates module files to manage the environment, and it is closely coupled with Lmod [21]. EasyBuild groups the compiler, MPI, FFT, and BLAS libraries together in a *toolchain* that can be used by package files. The grouping provides some composability and separates compiler flags and MPI concerns from client packages.

HashDist [2] is a meta-build system and package manager for HPC. Of the existing solutions, it is the most similar to Spack. Like Nix, it uses cryptographic versioning and stores installations in unique directories. Both Nix and HashDist use `RPATH`s in their packages to ensure that libraries are found correctly at runtime.

Site	Naming Convention

LLNL	`/ usr / global / tools / $arch / $package / $version`
	`/ usr / local  / tools / $package-$compiler-$build-$version`

ORNL [25]	`/ $arch / $package / $version / $build`
TACC / Lmod [28]	`/ $compiler-$comp_version / $mpi / $mpi_version / $package / $version`
Spack default	`/ $arch / $compiler-$comp_version / $package-$version-$options-$hash`

Table 1: Software organization of various HPC sites.

Gaps in Current Practice.

The flexible cryptographic versioning of Nix and HashDist manages the package *and* its dependency configuration and can represent any configuration. However, users cannot easily navigate or query the installed software. EasyBuild and Smithy generate environment modules, which supports some querying. Naming schemes used in existing module systems, however, cannot handle combinatorial versions, which the Lmod authors call the “matrix problem” [28].

The main limitation of existing tools is the lack of build *composability*. The full set of package versions is combinatorial, and arbitrary combinations of compiler, MPI version, build options, and dependency versions require non-trivial modifications to many package files. Indeed, the number of package files required for most existing systems scales with the number of version *combinations*, not the number of packages, which quickly becomes unmanageable. As an example, the EasyBuild system has over 3,300 files for several permutations of around 600 packages. A slightly different dependency graph requires an entire new package file hierarchy. HashDist supports composition more robustly but does not have first-class parameters for versions, compilers, or versioned interfaces. HPC sites need better ways to *parameterize* packages so that new builds can be *composed* in response to user needs.

3. THE SPACK PACKAGE MANAGER

Based on our experiences at LLNL, we have developed *Spack*, the Supercomputing Package manager. Spack is written in Python, which we chose for its flexibility and its increasing use in HPC. Like prior systems, Spack supports an arbitrary number of software installations, and like Nix it can identify them with hashes. Unlike any prior system, Spack provides a concise language to specify and manage the combinatorial space of HPC software configurations. Spack provides the following unique features:

1. Composable packages, explicitly **parameterized** by version, platform, compiler, options, and dependencies.
2. A novel, recursive **spec syntax** for dependency graphs and constraints, which aids in managing the build parameter space.
3. **Versioned virtual dependencies** to handle versioned, ABI-incompatible interfaces like MPI.
4. A novel **concretization** process that translates an abstract build specification into a full, concrete build specification.
5. A build environment that uses **compiler wrappers** to enforce build consistency and simplify package writing.

3.1 Packages

In Spack, packages are Python scripts that build software artifacts. Each package is a class that extends a generic `Package` base class. `Package` implements the bulk of the build process, but subclasses provide their own `install` method to handle the specifics of particular packages. The subclass does *not* have to manage the install location. Rather, Spack passes the `install` method a `prefix` parameter. The package implementer must ensure that the `install` function installs the package *into* the `prefix`, but Spack ensures that `prefix` is computed so that it is unique for every configuration of a package. To simplify package implementation further, Spack implements an embedded domain-specific language (DSL). The DSL provides directives, such as `depends_on`, `version`, `patch`, and `provides`, that add metadata to the package class.

```
1  class Mpileaks(Package):
2      """Tool to detect and report leaked MPI objects."""
3
4      homepage = "https://github.com/hpc/mpileaks"
5      url = homepage + "/releases/download/v1.0/mpileaks-1.0.tar.gz"
6
7      version('1.0', '8838c574b39202a57d7c2d68692718aa')
8      version('1.1', '4282eddb08ad8d36df15b06d4be38bcb')
9
10     depends_on('mpi')
11     depends_on('callpath')
12
13     def install(self, spec, prefix):
14         configure("--prefix=" + prefix,
15                   "--with-callpath=" + spec['callpath'].prefix)
16          make()
17          make("install")
```

Figure 1: Spack package for the `mpileaks` tool.

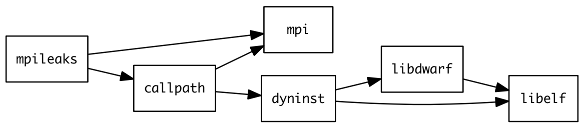
Figure 1 shows the package for `mpileaks`, a tool developed at LLNL to find un-released MPI handle objects in parallel programs. The `MpiLeaks` class provides simple metadata on lines 2-5: a text description, a homepage, and a download URL. Two `version` directives on lines 7-8 identify known versions and provide MD5 checksums to verify downloads. The two `depends_on` directives on lines 10-11 indicate prerequisite packages that must be installed before `mpileaks`. Last, the `install` method on line 13 contains the commands for building. Spack's DSL allows shell commands to be invoked as Python functions, and the `install` method invokes `configure`, `make`, and `make install` as a shell script would.

3.2 Spack Specs

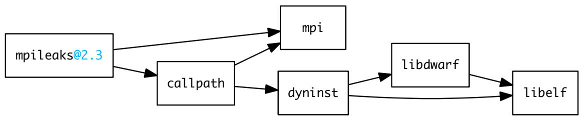
Using the simple script in Figure 1, Spack can build many different versions and configurations of the `mpileaks` package. In traditional port systems, package code is structured to build a single version of a package, but in Spack, each package file is a *template* that can be configured and built in many different ways, according to a set of *parameters*. Spack calls a single build configuration a *spec*. Spack communicates dependencies and parameters to package authors using the `spec` argument of the `install` method.

3.2.1 Structure

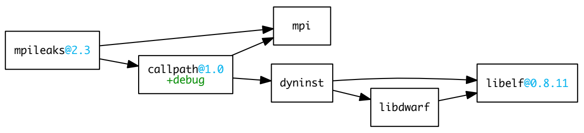
To understand specs, consider the `mpileaks` package structure. Metadata in the `Mpileaks` class (e.g., `version` or `depends_on`) describe its relationships with other packages. The tool has two direct dependencies: the `callpath` library and `mpi`. Spack recursively inspects the class definitions for each dependency and constructs a



(a) Spec for `mpileaks`



(b) `mpileaks@2.3`



(c) `mpileaks@2.3 ^callpath@1.0+debug ^libelf@0.8.11`

Figure 2: Constraints applied to `mpileaks` specs.

graph of their relationships. The result is a directed, acyclic graph (DAG).[1] To guarantee a consistent build and to avoid Application Binary Interface (ABI) incompatibility, we construct the DAG with only *one* version of each package. Thus, while Spack can install arbitrarily many configurations of any package, no two configurations of the same package will ever appear in the same build DAG.

DAGs for `mpileaks` are shown in Figure 2. Each node represents a package, and each package has five configuration parameters that control how it will be built: 1) the package version, 2) the compiler with which to build, 3) the compiler version, 4) named compile-time build options, or *variants*, and 5) the target architecture.

3.2.2 Configuration Complexity

A spec DAG has many degrees of freedom, and users cannot reasonably be expected to understand or to specify all of them. In our experience at LLNL, the typical user only cares about a small number of build constraints (if any), and does not know enough to specify the rest. For example, a user may know that a certain version of a library like `boost` [1] is required, but only cares that other build parameters are set so that the build will succeed. Configuration complexity makes the HPC software ecosystem difficult to manage: too many parameters exist to specify them all. However, the known and important ones often provide detailed build constraints. Thus, we have two competing concerns. We need the ability to specify details without having to remember all of them.

3.2.3 Spec Syntax

We have developed a syntax for specs that allows users to specify only constraints that matter to them. Our syntax can represent DAGs concisely enough for command line use. The spec syntax is recursively defined to allow users to specify parameters on dependencies as well as on the root of the DAG. Figure 3 shows the EBNF grammar through which Spack flexibly supports constraints.

We begin with a simple example. Consider the case where a user wants to install the `mpileaks` package, but knows nothing about its structure. To install the package, the user invokes `spack install`:

```
$ spack install mpileaks
```

Here, `mpileaks` is the simplest possible spec—a single identifier. Spack converts it into the DAG shown in Figure 2a. Note that even though the spec contains no dependency information, it is still

[1]Spack currently disallows circular dependencies.

```
⟨spec⟩          ::= ⟨id⟩ [ ⟨constraints⟩ ]
⟨constraints⟩   ::= { '@' ⟨version-list⟩ | '+' ⟨variant⟩
                    | '-' ⟨variant⟩     | '~' ⟨variant⟩
                    | '%' ⟨compiler⟩    | '=' ⟨architecture⟩ }
                    [ ⟨dep-list⟩ ]
⟨dep-list⟩      ::= { '^' ⟨spec⟩ }
⟨version-list⟩  ::= ⟨version⟩ [ { ',' ⟨version⟩ } ]
⟨version⟩       ::= ⟨id⟩ | ⟨id⟩ ':' | ':' ⟨id⟩ | ⟨id⟩ ':' ⟨id⟩
⟨compiler⟩      ::= ⟨id⟩ [ ⟨version-list⟩ ]
⟨variant⟩       ::= ⟨id⟩
⟨architecture⟩  ::= ⟨id⟩
⟨id⟩            ::= [A-Za-z0-9_][A-Za-z0-9_.-]*
```

Figure 3: EBNF grammar for spec expressions.

converted to a full DAG, based on directives supplied in package files. Since the spec does not constrain the nodes, Spack has leeway when building the package, and we say that it is *unconstrained*. However, a user who wants a specific `mpileaks` version can request it with a version constraint after the package name:

```
$ spack install mpileaks@2.3
```

Figure 2b shows that the specific version constraint is placed on the `mpileaks` node in the DAG, which otherwise remains unconstrained. A user who only requires a particular minimum version could use *version range* syntax and write `@2.3:`. Likewise, `@2.3:2.5.6` would specify a version between 2.3 and 2.5.6. In these cases, the user can save time if Spack already has a version installed that satisfies the spec – Spack will use the previously-built installation instead of building a new one.

Figure 2c shows the recursive nature of the spec syntax:

```
$ spack install mpileaks@2.3 ^callpath@1.0+debug ^libelf@0.8.11
```

The caret (`^`) denotes constraints for a particular dependency. The DAG now has version constraints on `callpath` *and* `libelf`, and the user has requested the `debug` variant of the `callpath` library.

Recall that Spack guarantees that only a single version of any package occurs in a spec. Within a spec, each dependency can be uniquely identified by its package name alone. Therefore, the user does not need to consider DAG connectivity to add constraints but instead must only know that `mpileaks` depends on `callpath`. Thus, dependency constraints can appear in an arbitrary order.

Table 2 shows further examples of specs, ranging from simple to complex. These examples show how Spack's constraint notation covers the rest of the HPC package parameter space.

Versions. Version constraints are denoted with `@`. Versions can be precise (`@2.5.1`) or denote a range (`@2.5:4.4`), which may be open-ended (`@2.5:`). The package in Figure 1 lists two "safe" versions with checksums, but in our experience users frequently want bleeding-edge versions, and package managers often lag behind the latest releases. Spack can extrapolate URLs from versions, using the package's `url` attribute as a model.[2] If the user requests a specific version on the command line that is unknown to Spack, Spack will attempt to fetch and to install it. Spack uses the same model to scrape webpages and to find new versions as they become available.

Compilers. With a compiler constraint (shown on line 3) the user simply adds `%` followed by its name, with an optional compiler version specifier. Spack compiler names like `gcc` refer to the full compiler *toolchain*, i.e., the C, C++, and Fortran 77 and 90 compilers. Spack can auto-detect compiler toolchains in the user's `PATH`, or they can be registered manually through a configuration file.

[2]This works for packages with consistently named URLs.

	Spec	Meaning
1	`mpileaks`	`mpileaks` package, no constraints.
2	`mpileaks@1.1.2`	`mpileaks` package, version 1.1.2.
3	`mpileaks@1.1.2 %gcc`	`mpileaks` package, version 1.1.2, built with `gcc` at the default version.
4	`mpileaks@1.1.2 %intel@14.1 +debug`	`mpileaks` package, version 1.1.2, built with Intel compiler version 14.1, with the "debug" build option.
5	`mpileaks@1.1.2 =bgq`	`mpileaks` package, version 1.1.2, built for the Blue Gene/Q platform (BG/Q).
6	`mpileaks@1.1.2 ^mvapich2@1.9`	`mpileaks` package version 1.1.2, using `mvapich2` version 1.9 for MPI.

7	`mpileaks @1.2:1.4 %gcc@4.7.5 -debug =bgq`
	`  ^callpath @1.1 %gcc@4.7.2`
	`  ^openmpi @1.4.7`
		`mpileaks` at any version between 1.2 and 1.4 (inclusive), built with `gcc` 4.7.5, without the debug option, for BG/Q, linked with `callpath` version 1.1 and building `callpath` with `gcc` version 4.7.2, linked with `openmpi` version 1.4.7.

Table 2: Spack build spec syntax examples and their meaning.

```python
def install(self, spec, prefix):  # default build uses cmake
    with working_dir('spack-build', create=True):
        cmake('..', *std_cmake_args)
        make()
        make("install")

@when('@:8.1')                    # <= 8.1 uses autotools
def install(self, spec, prefix):
    configure("--prefix=" + prefix)
    make()
    make("install")
```

Figure 4: Specialized `install` method in Dyninst.

Variants. To handle options like compiler flags or optional components, specs can have named flags, or *variants*. Variants are associated with the package, so the `mpileaks` package implementer must explicitly handle cases where debug is enabled (+debug) or disabled (-debug or ~debug). Names simplify versioning and prevent the configuration space from becoming too large. For example, including detailed compiler flags in spec syntax would violate our goal of conciseness, but known sets of flags can simply be named.

Cross-compilation. To support cross-compilation, specs can include the system architecture (line 5). Platforms begin with = and take names like `linux-ppc64` or `bgq`. They are specified per-package. This mechanism allows front-end tools to depend on their back-end measurement libraries with a *different* architecture on cross-compiled machines.

3.2.4 Constraints in Packages

So far, we have shown examples of specs used to request constraints from the command line, when `spack install` is invoked. However, the user is not the only source of constraints. Applications may require specific versions of dependencies. Often, these constraints should be specified in a package file. For example, the ROSE compiler [34] only builds with a certain version of the `boost` library. The `depends_on()` directives in Figure 1 contain simple package names. However, a package name is also a spec, and the same constraint syntax usable from the command line can be applied inside directives. So, we can simply write:

```python
depends_on('boost@1.54.0')
```

This constraint will be incorporated into the initial DAG node generated from the `ROSE` package. Dependencies can also be *conditional*. For example, the `ROSE` package builds with different versions of boost, depending on the compiler version. So, directives also accept spec syntax as a predicate in an optional `when` parameter:

```python
depends_on('boost@1.47.0', when='%gcc@:4')
depends_on('boost@1.54.0', when='%gcc@5:')
```

The same notation is used to build optionally with libraries like MPI:

```python
depends_on('mpi', when='+mpi')
```

and to ensure that specific patches are applied to Python source code when it is built on Blue Gene/Q, with different compilers:

```python
patch('python-bgq-xlc.patch',   when='=bgq%xl')
patch('python-bgq-clang.patch', when='=bgq%clang')
```

Constraints in the `when` clause are matched against the package spec. Outside of directives, constraints can also be tested directly on the `spec` object in the `install` method:

```python
def install(self, spec, prefix):
    if spec.satisfies('%gcc'):
        # Handle gcc
    elif spec.satisfies('%xl'):
        # Handle XL compilers
```

3.2.5 Build Specialization

In our experience maintaining packages at LLNL, we often must change entire package build scripts due to large changes in the way certain packages build. These changes are cumbersome, particularly since we often must maintain both the old and new version of a build script, as some users rely on the old version.

Spack provides functionality that allows Python functions to have multiple definitions, each specialized for particular configurations of the package. This capability allows us to have two separate implementations of `install` or *any* method in a package class, as Figure 4 shows for the Dyninst package. The `@when` directive is a Python decorator: a higher order function that takes a function definition as a parameter and returns a new function to replace it. We replace the function with a callable multi-function dispatch object, and we integrate the predicate check into the function dispatch mechanism. The `@when` condition is true when Dyninst is at version 8.1 or lower, in which case the package will use the `configure`-based build. By default if no predicate matches, `install` will use the default CMake-based implementation. The simple `@when` annotation allows us to maintain our old build code alongside the new version without accumulating complex logic in a single `install` function.

3.3 Versioned Virtual Dependencies

Many libraries share a common interface and can be interchanged within a build. The archetypal examples in HPC are MPI libraries, which has several open source (e.g., MPICH, OpenMPI, and MVAPICH) and vendor-specific implementations. An application that can be built with one MPI implementation can generally be built with another. Another example is the Basic Linear Algebra Subroutines (BLAS), which has many fungible implementations (e.g., ATLAS, LAPACK-BLAS, and MKL).

Neither MPI nor BLAS has a standard ABI, so applications cannot simply be re-linked with a new version. They must be recompiled

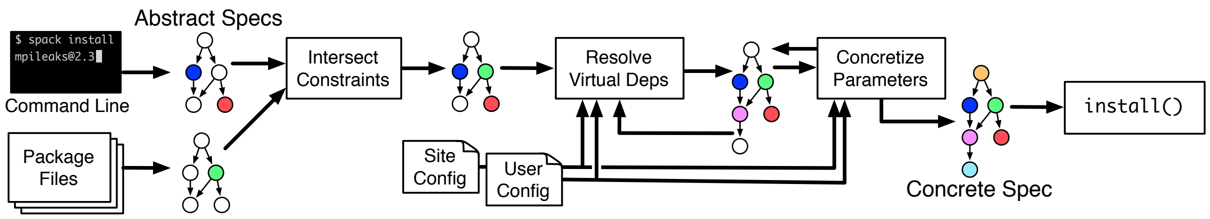


Figure 6: Spack's concretization process.

```
# providers of mpi
class Mvapich2(Package):
    provides('mpi@:2.2', when='@1.9')
    provides('mpi@:3.0', when='@2.0')
    ...
class Mpich(Package):
    provides('mpi@:3', when='@3:')
    provides('mpi@:1', when='@1:')
    ...
```

```
# mpi dependents
class Mpileaks(Package):
    depends_on('mpi')
    ...
class Gerris(Package):
    depends_on('mpi@2:')
    ...
```

Figure 5: Virtual dependencies.

and reinstalled. At LLNL, we must frequently build tools with many versions of MPI to support the many different applications that run at our center. Complicating matters, the MPI interface is versioned, and some packages need later versions of MPI to run correctly. Often, MPI *implementation* versions do not correspond obviously to MPI interface versions, and determining the right version of MPI to pair with an application can be tedious and tricky.

To allow rapid composition of libraries using an interface, Spack supports *virtual dependencies*. A virtual dependency is an abstract name that represents a library interface (or capability) instead of a library implementation. Packages that need this interface do need not depend on a specific implementation; they can depend on the virtual name, for which the user or Spack can select an implementation at build time. Other package managers support the notion of virtual dependencies, but Spack adds *versioning* to its interfaces, which directly supports concepts like MPI versions and BLAS levels. Spack handles the details of managing and checking complex versioning constraints.

Figure 5 shows how packages provide virtual interfaces in Spack. The spec syntax concisely associates ranges of `mpi` versions for the `mvapich2` and `mpich` packages. The `mpileaks` package requires `mpi`, but it does not constrain the version. Any version of `mvapich2` or `mpich` could be used to to satisfy the `mpi` constraint. The Gerris computational fluid dynamics library, however, needs MPI version 2 or higher. So any version except `mpich` 1.x could be used to satisfy the constrained dependency.

3.4 Concretization

We have discussed Spack's internal software DAG model, and we have shown how the spec syntax can be used to specify a partially constrained software DAG. This DAG is *abstract*, as it could potentially describe more than one software configuration. Before Spack builds a spec, it must ensure the following conditions:

1. No package in the spec DAG is missing dependencies;
2. No package in the spec DAG is virtual;
3. All parameters are set for all packages in the DAG.

If a spec meets these criteria then it is *concrete*. *Concretization* is the central component of Spack's build process that allows it to reduce an unconstrained abstract description to a concrete build.

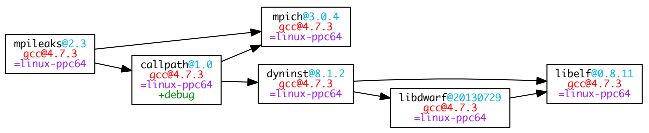


Figure 7: Concretized spec from Figure 2a.

Figure 6 shows Spack's concretization algorithm. The process starts when a user invokes `spack install` and requests that a spec be built. Spack converts the spec to an abstract DAG. It then builds a *separate* spec DAG for any constraints encoded by directives in package files. Spack intersects the constraints of the two DAGs package by package, and it checks each parameter for inconsistencies. Inconsistencies can arise if, for example, the user inadvertently requests two versions of the same package, or if a package file depends on a different version than the user requested. Likewise, if the package and the user specified different compilers, variants, or platforms for particular packages, Spack will stop and notify the user of the conflict. If the user or package specifies version ranges, they are intersected, and if the ranges do not overlap, Spack raises an error. When the intersection succeeds, Spack has a single DAG with the merged constraints of the user and the package files.

The next part of the process is iterative. If any node is a virtual dependency, Spack replaces it with a suitable interface provider by building a reverse index from virtual packages to providers using the `provides when` directives (Section 3.3). If multiple providers satisfy the virtual spec's constraints, Spack consults site and user policies to select the "best" possible provider. The selected provider may *itself* have virtual dependencies, so this process is repeated until the DAG has no more virtual packages.

With the now non-virtual DAG, Spack again consults site and user preferences for variants, compilers, and versions to resolve any remaining abstract nodes. Adding a variant like `+mpi` may cause a package to depend on more libraries (as in Section 3.2.4). The concretization algorithm evaluates conditions from when clauses on the DAG; if they result in new libraries or other changes, the cycle begins again. Spack currently avoids an exhaustive search by using a greedy algorithm. It will not backtrack to try other options if its first policy choice leads to an inconsistency. Rather, it will raise an error and the user must resolve the issue by being more explicit. The user might toggle a variant or force the build to use a *particular* MPI implementation by supplying `^openmpi` or `^mpich`. We leave automatic constraint space exploration for future work.

On completion, the concretization outputs a fully concrete spec DAG. Figure 7 shows a concrete DAG with architectures, compilers, versions, variants, and all dependencies resolved. This fulfills Spack's guarantees. At install time, Spack constructs a package object for each node in the spec DAG and traverses the DAG in a bottom-up fashion. At each node, it invokes the package's `install` method. For the `spec` parameter to `install`, it passes a sub-DAG rooted at current node (also a concrete spec). Package authors must query the spec in `install` to handle different configurations.

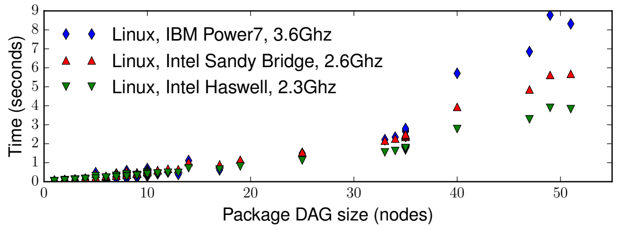


Figure 8: Concretization running time for 245 packages.

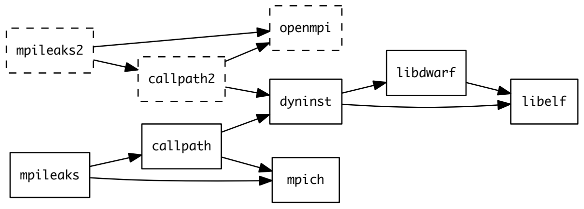


Figure 9: `mpileaks` **built with** `mpich`**, then** `openmpi`**.**

3.4.1 Concretization Runtime

Figure 8 shows the running time of concretization vs. package DAG size in nodes. We generated this plot by running `concretize` on all of Spack's 245 packages. We tested on three LLNL cluster front-end nodes: a 2.3GHz Intel Haswell node and a 2.6 GHz Intel Sandy Bridge node on our Linux clusters, and the 3.6 GHz IBM Power7 front-end node of a Blue Gene/Q system. Each point is the average of 10 trials. For all but the 10 largest packages, concretization takes less than 2 seconds on all machines. For larger DAGs, we begin to see a quadratic trend, but even for 50 nodes or more, concretization takes less than 4 seconds on the Haswell machine and 9 seconds on the Power7. We expect this performance from our greedy, fixed-point method, and it is sufficient for the packages we have built so far. Its running time is insignificant compared to the build time of most packages, and it only executes once per build. More generally, concretization is an instance of the constraint satisfaction problem, which is NP-complete. While concretization could become more costly, we do not expect to see packages with thousands of dependencies in the near future. We do not expect it to become a bottleneck, even if we use a full constraint solver.

3.4.2 Shared sub-DAGs

We mentioned in Section 3.1 that each unique configuration is guaranteed a unique install prefix. Spack uses the concrete *spec* to generate a unique path, shown in Table 1. To prevent the directory name from growing too long, Spack uses a SHA hash of dependencies' specs as the last directory component. However, Spack does *not* rebuild every library for each new configuration. If two configurations share a sub-DAG, then Spack reuses the sub-DAG's configuration. Figure 9 shows how the `dyninst` sub-DAG is used for both the `mpich` and `openmpi` builds of `mpileaks`.

3.4.3 Reproducibility

For reproducibility, and to preserve provenance, Spack stores a number of files in the installation directory that document how the installed package was built. These include the `package.py` file used to build, a build log that contains output and error messages, and a file that contains the complete concrete spec for the package *and* its dependencies. The spec file can be used later to reproduce the build, even if concretization preferences have changed.

3.4.4 Site Policies and Build Complexity

Our experience with manual installs helped us understand that much of the complexity of building HPC software comes from the size of the build parameter space. As previously mentioned, typical users only care about a few parameters. The rest add unneeded complexity. When building manually, LLNL staff tend to make arbitrary choices about the secondary build parameters, or they add logic to build scripts to make these choices. Manually built software is generally not installed in a consistent manner.

Concretization provides two benefits. First, it allows users and staff to request builds with a minimal spec expression, while still providing a mechanism for the site and the user to make consistent, repeatable choices for other build parameters. For example, the site or the user can set default versions to use for any library that is not specified explicitly. Second, concretization reduces the burden of packaging software, because package authors do not have to make these decisions. Packages do not need to contain complicated checks, or to be overly specific about versions. Other multi-configuration systems like Nix, EasyBuild, and HashDist require the package author to write a mostly concrete build spec in advance. This requirement puts undue burden on the package author, and it makes the task of changing site policies within a software stack difficult. Spack separates these concerns.

3.5 Installation Environment

Spack is designed to build a consistent HPC stack for our environment, and reproducible builds are one of our design goals. Experience at LLNL has shown that reproducing a build manually is vexingly difficult. Many packages have a profusion of build options. Specifying them correctly often requires tedious experimentation due to lack of build standards, as well as the diversity of HPC environments. For example, in some packages that depend on the `Silo` library, the `--with-silo` parameter takes a path to `Silo`'s installation prefix. In others, it takes the `include` and `lib` subdirectories, separated by a comma. The `install` method in Spack's packages encodes precise build incantations for later reuse.

3.5.1 Environment Isolation

Inconsistencies arise not only from building manually but also due to differences between the package installation environment and the user environment. For example, LLNL performance tools use two versions of the `libelf` library. One is distributed with RedHat Linux, while the other is publicly available. They have the same API but an incompatible ABI. Specifying the wrong version at build time has caused many unexpected crashes at runtime.

Spack manages the build environment by running each call to `install` in its own process. It also sets `PATH`, `PKG_CONFIG_PATH`, `CMAKE_PREFIX_PATH`, and `LD_LIBRARY_PATH` to include the dependencies of the current build. Build systems commonly use these variables to locate dependencies, and setting them helps to ensure that the correct libraries are detected. Using a separate process also gives package authors free reign to set build-specific environment variables without interfering with other packages.

3.5.2 Compiler Wrappers and RPATHs

Finding dependencies at build time is not the only obstacle to reproducible behavior. As mentioned in Section 2, binaries must be able to find dependency libraries at *runtime*. One of the most common user errors at LLNL is improper library configuration. Users frequently do not know the libraries with which a package was built, and constructing a suitable `LD_LIBRARY_PATH` for a package that was built by someone else is difficult. To reduce the number of support calls that we receive, we typically add `RPATHs` to public software installations, so that paths to dependencies are embedded

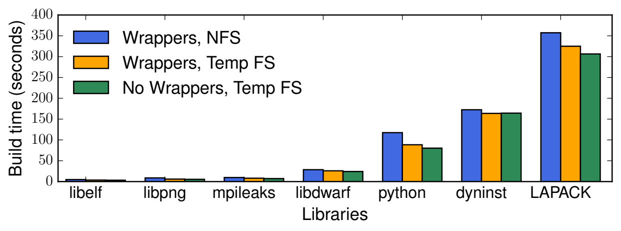


Figure 10: Build time on NFS and temp, with and without wrappers.

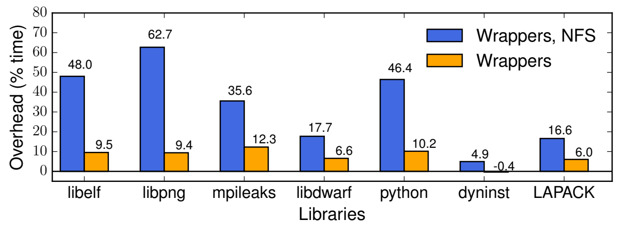


Figure 11: Build overhead of using NFS and compiler wrappers. ‘

in binaries. Thus, users do not need to know details of dependencies to run installed software correctly.

Spack manages `RPATHs` and other build policies with *compiler wrappers*. In each isolated `install` environment, Spack sets the environment variables `CC`, `CXX`, `F77`, and `FC` to point to its compiler wrapper scripts. These variables are used by most build systems to select C, C++, and Fortran compilers, so they are typically picked up automatically.[3] When run, the wrappers insert include (`-I`), library (`-L`), and `RPATH` (`-Wl,-rpath` or similar) flags into the argument list. These point to the `include` and `lib` directories of dependency installations, where needed headers and libraries are located. The wrappers invoke the real compiler with the modified arguments.

Spack's compiler wrappers have several benefits. First, they allow Spack to switch compilers transparently in most builds, which is how we implement compiler options like `%gcc`. Second, they enforce the use of `RPATHs` in installed binaries, which allows applications that Spack builds to run correctly *regardless of the environment*. Third, because compiler wrappers add header and library search paths for dependencies, header and library detection tests that most build systems use succeed without using special arguments for nonstandard locations. `configure` commands in Spack's `install` function can have fewer arguments, and can be written as for system installs, which reduces complexity for package maintainers and helps to ensure consistent, reproducible build policies across packages. Finally, because Spack controls the wrappers, package authors can programmatically filter the compiler flags used by build systems, which facilitates porting to bleeding-edge platforms or new compilers.

3.5.3 *Build Overhead*

Argument parsing and indirection cause Spack's compiler wrappers to incur a small but noticeable performance overhead. Figure 10 shows build times with and without compiler wrappers for seven different packages, and Figure 11 shows the corresponding overhead as a percentage of the wrapper-less runtime. `Dyninst` and `LAPACK` use CMake to build, and the rest use autotools. We ran each build three times on an Intel Sandy Bridge cluster node, and we report the average. In the case of `dyninst`, the overhead is negligible (-0.4%), but it is as high as 12.3% for shorter builds, like `mpileaks`.

To increase build speed, by default Spack runs each `install` method in the default temporary directory, which is a fast, locally-

[3]If builds do not respect `CC`, `CXX`, `FC`, or `F77`, wrappers can be added as arguments or inserted into Makefiles by `install`.

```
patch('patch.gpeftools2.4_xlc', when='@2.4 %xlc')

def install(self, spec, prefix):
    if spec.architecture == 'bgq' and self.compiler.satisfies('xlc'):
        configure("--prefix=" + prefix,  "LDFLAGS=-qnostaticlink")
    elif spec.architecture == 'bgq':
        configure("--prefix=" + prefix,  "LDFLAGS=-dynamic")
    else:
        configure("--prefix=" + prefix)

    make()
    make("install")
```

Figure 12: Simplified install routine for gperftools.

mounted file system on most cluster nodes. In our experience, many users build manually in their home directories out of habit. At most sites, home directories are remotely mounted volumes (e.g., NFS). To compare Spack builds with *typical* manual builds, we timed the same seven builds using NFS. Figure 11 shows that building this way can be as much as 62.7% slower than using a temporary file system and 33% slower on average. We therefore expect many users to notice a speedup when Spack uses temporary space to build.

3.5.4 *Environment Module Integration*

In addition to managing the build-time environment, Spack can assist in managing the run-time environment. A package may need environment variables like `PATH`, `MANPATH`, or `PKG_CONFIG_PATH` set before it can be used. As discussed in Section 2, many sites rely on environment modules to setup the runtime environment. Spack can automatically create simple dotkit [6] and Module configuration files for its packages, allowing users to setup their runtime environment using familiar systems. While Spack packages do not need `LD_LIBRARY_PATH` to run, we set it in our generated module files as well, because it can be used by build systems to find libraries, as well as by dependent packages that do not use `RPATH`. Future versions of Spack may also allow the creation of Lmod [27] hierarchies. Spack's rich dependency information would allow automatic generation of such hierarchies.

4. USE CASES

We have recently begun using Spack in production at Livermore Computing (LC). We have already outlined a number of the experiences that drove us to develop Spack. In this section, we describe real-world use cases that Spack has addressed. In some cases, Spack had to be adapted to meet production needs, and we describe how its flexibility has allowed us to put together solutions quickly.

4.1 Combinatorial Naming

`Gperftools` is a suite of tools from Google that has gained popularity among developers for its high-performance thread-safe heap and its lightweight profilers. Unfortunately, two issues made it difficult to maintain `gperftools` installations at LC. First, `gperftools` is a C++ library. Since C++ does not define a standard ABI, `gperftools` must be rebuilt with each compiler and compiler version used by client applications. Second, building `gperftools` on bleeding-edge architectures (such as Blue Gene/Q) requires patches and complicated configure lines that change with each compiler. One application team tried to maintain their own builds of `gperftools`, but the maintenance burden soon became too great.

Spack presented a solution to both problems. Package administrators can use Spack to maintain a central install of `gperftools` across combinations of compilers and compiler versions easily. Spack's `gperftools` package also serves as a central institutional knowledge repository, as its package files encode the patches and configure lines required for each platform and compiler combination. Figure 12 illustrates per-compiler and platform build rules with a simplified version of the install routine for gperftools. The package applies a patch if gperftools 2.4 is built with the XL compiler, and it selects the correct configure line based on the spec's platform and compiler.

LC's mpileaks tool, which has been a running example in this paper, has an even larger configuration space than gperftools. It must be built for different compilers, compiler versions, MPI implementations, and MPI versions. As with gperftools, we have been able to install many different configurations of mpileaks using Spack. Moreover, Spack's virtual dependency system allows us to compose a new mpileaks build quickly when a new MPI library is deployed at LC, *without* modifying the mpileaks package itself.

4.2 Support for Interpreted Languages

Python is becoming increasingly popular for HPC applications, due to its flexibility as a language and its excellent support for calling into fast, compiled numerical libraries. Python is an interpreted language, but one can use it as a friendlier interface to compiled libraries like FFTW, ATLAS, and other linear algebra libraries. Many LLNL code teams use Python in this manner. LC supports Python installations for several application teams. Maintaining these repositories has grown increasingly complex over time. The problems are similar to those that drove us to create Spack: different teams want different Python libraries with different configurations.

Existing Python package managers either do not support building from source [7, 8], or they are language-specific [14]. None handles multi-configuration builds. More glaringly, Python extensions are usually installed into the Python interpreter's prefix, which makes it impossible to install multiple versions.[4] Installing each extension in its own prefix enables combinatorial versioning, but it requires users to add packages to the PYTHONPATH variable at runtime.

Per Spack's design philosophy, we wanted a way to manage many different versions easily, but *also* to provide a baseline set of extensions *without* requiring environment settings. To support this mode of operation, we added the concept of extension packages to Spack. Python modules use the extends('python') directive instead of depends_on('python'). Each module installs into its own prefix like any other package, and each module depends on a particular Python installation. However extensions can be activated or deactivated within the dependent Python installation. The activate operation symbolically links each file in the extension prefix into the Python installation prefix, as if it were installed directly. If any file conflict would arise from this operation, activate fails. Similarly, the deactivate operation removes the symbolic links and restores the Python installation to its pristine state.

Additional complications arose because many Python packages *install their own package manager* if they do not find one in the Python installation. Also, Python packages use a wide range of mechanisms to add themselves to the interpreter's default path, some of which conflict. We modified Spack so that extendable packages, like Python, can supply custom code in the package file that specializes activate and deactivate for the particular package. For Python, this feature merges conflicting files during activation. Overall, Python extensions can be installed automatically in their own prefixes, and they can be composed with a wide range of bleeding-edge libraries that other package managers do not handle.

Spack essentially implements a "meta package-manager" for each Python instance, which coexists with Spack's normal installation model. This mechanism efficiently supports application teams, for whom we can now rapidly construct custom Python installations. We have also reduced the time that LC staff spend installing Python modules. Because Spack packages can extend activation and deactivation mechanisms, this design could also be used with other languages with similar extension models, such as R, Ruby, or Lua.

[4] setuptools has support for multiple versions via pkg_resources, but this requires modifications to client code.

4.3 User and Site Policies

Spack makes it easy to create and to organize package installations, but it must also be easy for end-users to find and use packages on an HPC system. Different end-users have different expectations about how packages should be built and installed, and at LLNL those expectations are shaped by years of site policies, personal preferences, and lingering legacy decisions originally made on a DEC VAX. Spack provides mechanisms that allow users to customize it for the needs of particular sites and users.

4.3.1 Package Views

While Spack can easily create many installations of a package like mpileaks, end-users may find Spack's directory layout confusing, and they may not be able to find libraries intuitively. Spack installs packages into paths based on concretized specs, which is ideal for maintaining multiple package installations. However, end-users would find these paths difficult to navigate. For example, an mpileaks installation prefix might be:

```
spack/opt/linux-x86_64/gcc-4.9.2/mpileaks-1.0-db465029
```

As discussed in Section 3.5.4, environment modules are commonly used on HPC systems to solve this problem, and Spack allows the package author to create environment modules automatically for packages that Spack installs. Even when modules are available, many users still navigate the file system to access packages.

Spack allows the creation of *views*, which are symbolic-link based directory layouts of packages. Views provide a human-readable directory layout that can be adapted to match legacy directory layouts on which users rely. For example, the above mpileaks package may have a view that creates a link in /opt/mpileaks-1.0-openmpi to the Spack installation of mpileaks. The same package install may be referenced by multiple links and views, so the above package could also be linked from a more generic /opt/mpileaks-openmpi link. This reuse supports users who always want to use the latest version. Views can also be used to create symbolic links to specific executables or libraries in an install, so a Spack-built gcc@4.9.2 install may have a view that creates links from /bin/gcc49 and /bin/g++49 to the appropriate gcc and g++ executables.

Views are configured using configuration files, which can be set up at the site or user level. For each package or set of packages, the configuration file contains rules that describe the links that should point into that package. The link names can be parameterized. For example, the above mpileaks symbolic link might have been created by a rule like:

```
    /opt/${PACKAGE}-${VERSION}-${MPINAME}
```

On installation and removal, links are automatically created, deleted, or updated according to these rules.

Spack's views are a projection from points in a high-dimensional space (concretized specs, which fully specify all parameters) to points in a lower-dimensional space (link names, which may only contain a few parameters). Several installations may map to the same link. For example, the above mpileaks link could point to an mpileaks compiled with gcc or icc—the compiler parameter is not part of the link. To keep package installations consistent and reproducible, Spack has a well-defined mechanism for resolving conflicting links; it uses a combination of internal default policies and user- or site-defined policies to define an *order* of preference for different parameters. By default, Spack prefers newer versions of packages compiled with newer compilers to older packages built with older compilers. It has a well-defined, but not necessarily meaningful, order of preference for deciding between MPI implementations and different compilers. The default policies can be overridden in configuration files, by either users or by sites. For example, at one site users may typically use the Intel compiler, but some users also use the system's default `gcc@4.4.7`. These preferences could be stated by adding:

```
    compiler_order = icc,gcc@4.4.7
```

to the site's configuration file, which would cause the ambiguous `mpileaks` link to point to an installation compiled with `icc`. Any compiler not in the `compiler_order` setting is less preferred than those explicitly provided. In a similar manner, Spack can be configured to give specific package configurations priority over others, which can be useful with a new, unstable and untested version.

4.3.2 External Package Repositories

By default, Spack stores its package files in a mainline repository that is present when users first run Spack. At many sites, packages may build sensitive, proprietary software, or they may have patches that are not useful outside of a certain company or organization. Putting this type of code back into a public repository does not often make sense and, if it makes the mainline less stable, it can actually make sharing code between sites more difficult.

To support our own private packages, and to support those of LLNL code teams, Spack allows the creation of site-specific variants of packages. Users can specify additional search directories for finding additional `Package` classes via configuration files. The additional packages are like the `mpileaks` package shown in Figure 1. However, the extension packages can extend from not only `Package`, but also any of Spack's built-in packages. Custom packages can inherit from and replace Spack's default packages, so other sites can either tweak or completely replace Spack's build recipes. For example, a site can write a local Python class that inherits from Spack's base class. The local class may simply add configure flags to Spack's version, while leaving the dependencies and most of the build instructions from its parent class. For reproducibility, Spack also tracks the `Package` class that drove a specific build.

4.4 The ARES Multi-physics Code

For our final use case, we describe our experiences using Spack to build ARES. ARES [9, 31] is a 1, 2 and 3-dimensional radiation hydrodynamics code, developed for production use at LLNL. It can run both small, serial and large, massively parallel jobs. ARES is used primarily in munitions modeling and inertial confinement fusion simulations. At LLNL, it runs on commodity Linux clusters and on Blue Gene/Q systems. It also runs on the Cielo Cray XE6 system at Los Alamos National Laboratory (LANL), and it is being ported to LANL's forthcoming Trinity Cray XC30 machine on Trinitite, a smaller version of the full system. The Trinity machine will consist of two partitions; one using Intel Haswell processors and another using Intel Knights Landing processors. Currently, only the Haswell partition is deployed on Trinitite.

ARES comprises 47 packages, with complex dependency relationships. Figure 13 shows the DAG for the current production configuration of ARES. At the top is ARES itself. ARES depends on 11 LLNL physics packages, 4 LLNL math/meshing libraries, and 8 LLNL utility libraries. The utility libraries handle tasks including logging, I/O, and performance measurement. ARES also uses 23 external software packages, including MPI, BLAS, Python, and many other libraries. Together, these packages are written in a diverse set of languages including C, C++, Fortran, Python and tcl.

	Linux			*BG/Q*	*Cray XE6*
	MVAPICH	*MVAPICH2*	*OpenMPI*	*BG/Q MPI*	*Cray MPI*
GCC	C P L D			C P L D	
Intel 14	C P L D				
Intel 15	C P L D	D			
PGI		D	C P L D		C L D
Clang	C P L D			C L D	
XL				C P L D	

Table 3: Configurations of ARES built with Spack: (C)urrent and (P)revious production, (L)ite, and (D)evelopment).

We have configured Spack to build ARES with external MPI implementations, depending on the host system. This configuration exploits vendor- or site-supplied MPI installations that often use host-specific, optimized network drivers. MPI is thus shown as a virtual dependency in the figure. ARES builds its own Python version, so that it can run on machines where Python is not well supported, like Blue Gene/Q. Specifically, for Blue Gene/Q, it builds Python 2.7.9, which the native software stack does not support.

Prior to using Spack, ARES managed its software stack with MixDown. Thus, the ARES team already had some experience supporting automated builds of dependencies. We developed Spack packages for the packages in Figure 13. Many of the external packages were already available in Spack, but some, such as Python, required modifications to support the new platforms and compilers.

Table 3 shows configurations of ARES that the ARES team tests nightly. The rows and columns show architectures, compilers, and MPI versions. The ARES Spack package supports four different code configurations: the current (C) and previous (P) production versions, a "lite" version (L) that includes a smaller set of features and dependencies, and a development version (D). Each cell in the table indicates the ARES configurations built for an architecture, compiler, and MPI combination. Each configuration requires a slightly different set of dependencies and dependency versions, but one common ARES package supports all of them with conditional logic on versions and variants.

Altogether, the initial packaging effort required roughly two months for an experienced build engineer working 20 hours per week. This includes time spent learning to use Spack. As shown in the table, 36 different configurations have been run using Spack (up to 4 versions on each of 10 architecture-compiler-MPI combinations). Prior to using Spack, only Linux/Intel configurations were automated. The ARES team listed a number of key features that enabled the increased automation:

1. Spack's version tracking and optional dependencies were required to build the four configurations with correct libraries;
2. The spec syntax allowed build scripts to test compiler, compiler version, and dependency versions concisely—a necessity for handling the different architectures;
3. Patching packages for particular platforms was necessary to build many packages; and
4. Using a DSL embedded in Python was a significant benefit; certain packages required custom scripting to patch.

The ARES team also mentioned two potential long-term payoffs. First, Spack allows the team to test with Clang. This compiler is not currently used in production but probably will be on future LLNL machines. Testing with Clang revealed many incompatibilities, which were patched with Spack. The team is communicating these

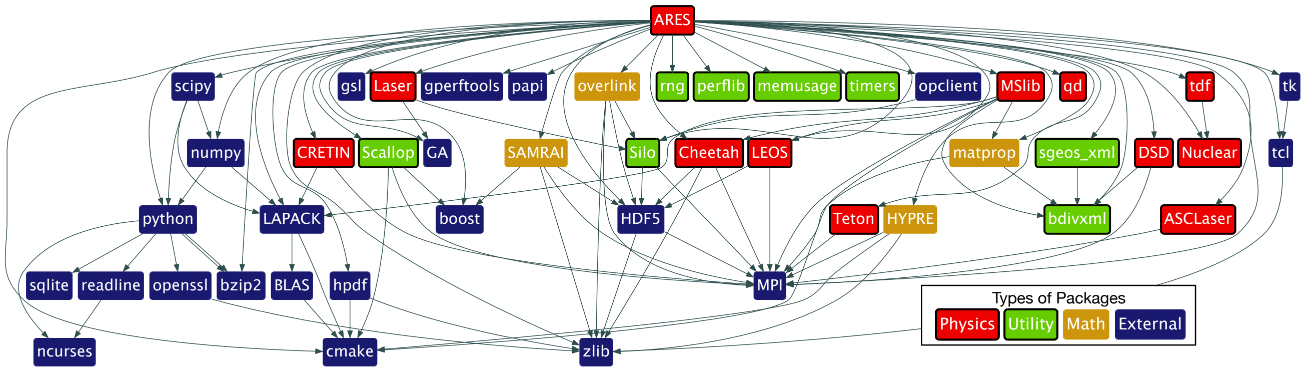


Figure 13: Dependencies of ARES, colored by type of package.

changes back to library developers, who will integrate them in future versions. In this case, build automation has allowed more testing, which helps both ARES and LLNL library developers build more robust software. Second, other LLNL code teams use many libraries that ARES uses. The LLNL code teams have begun creating an internal repository of Spack build recipes. Leveraging this repository will make packaging the next code significantly easier.

4.5 Limitations and Future Work

Spack's current implementation has several limitations. As mentioned in Section 3.4, we use a greedy concretization algorithm. For example, if package `P` depends on both `hwloc@1.9` and `mpi`, and if the algorithm chooses an MPI implementation for `P` that depends strictly on `hwloc@1.8`, a conflict on `hwloc` arises. In this case, Spack raises an error and the user must resolve the issue. Our implementation does not backtrack to find an MPI version that does not conflict. These cases have been rare so far. However, we plan to add better constraint solving to Spack in future work.

While Spack supports different architectures as part of the configuration space, we cannot currently factor common preferences (like `configure` arguments and architecture-specific compiler flags) out of packages and into separate architecture descriptions, which leads to some clutter in the package files when too many per-platform conditions accumulate. We are adding features that will further simplify Spack's build templates for cross-platform installations.

Spack requires more disk space than a module-based system, as otherwise identical packages with different dependencies must be built separately. The exact space overhead depends on the structure of the installed software; some builds can share more dependency libraries than others (see Figure 9). In our view, the significant reduction in complexity for the end user justifies this cost.

The use of Python has been a barrier for some users, and a major attraction for others. For those who would otherwise create local workarounds, Python's flexibility has allowed them to extend local Spack packages to suit their needs. Such customizations frequently guide our priorities for new core features. Some users have found the learning curve to be too steep, but this only prevents them from packaging software. They still use Spack as a command-line tool.

In the near term, we are actively working to grow a community around Spack and to build a larger base of contributors. We plan to increase build robustness by deploying a continuous testing system at LLNL. Finally, to support the growing number of HPC languages, runtimes, and programming models, we will add capabilities to Spack that allow packages to depend on particular compiler features. More and more, our codes are relying on advanced compiler capabilities, like C++11 language features, OpenMP versions, and GPU compute capabilities. Ideally, Spack will find suitable compilers and ensure ABI consistency when many such features are in use.

5. CONCLUSION

The complexity of managing HPC software is rapidly increasing, and it will continue unabated without better tools. In this paper, we reviewed the state of software management tools across a number of HPC sites, with particular focus on Livermore Computing (LC). While existing tools can handle multi-configuration installs, none of them sufficiently addresses the combinatorial nature of the software configuration space. None of them allows a user to *compose* new builds with version, compiler, and dependency parameters rapidly.

We introduced Spack, a package manager in development at LLNL, that provides truly *parameterized* builds. Spack implements a novel, recursive *spec* syntax that simplifies the process of working with large software configuration spaces, and it builds software so that it will run correctly, regardless of the environment. We outlined a number of Spack's unique features, including versioned virtual dependencies, and its novel *concretization* process, which converts an abstract build DAG into a concrete, build-able spec.

We showed through four use cases that Spack is already increasing operational efficiency in production at LLNL. The software management techniques implemented in Spack are applicable to a broad range of HPC facilities. Spack is available online at http://github.com/scalability-llnl/spack.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory, Contract DE-AC52-07NA27344. LLNL-CONF-675189.

References

[1] Boost C++ Libraries. http://www.boost.org, 2012.

[2] Hashdist. http://github.com/hashdist/hashdist, 2012.

[3] J. F. Amundson. Contractor: A Meta-build System for Building Heterogeneous Collections of Software Packages. http://home.fnal.gov/ amundson/contractor-www/.

[4] D. Bader, P. Kogge, A. Lumsdaine, and R. Murphy. The Graph 500 List. http://www.graph500.org.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[6] L. Busby and A. Moody. The Dotkit System. http://computing.llnl.gov/?set=jobs&page=dotkit.

[7] Continuum Analytics. Anaconda: Completely Free Enterprise-Ready Python Distribution for Large-Scale Data Processing, Predictive Analytics, and Scientific Computing. https://store.continuum.io/cshop/anaconda/.

[8] Continuum Analytics. Conda: A Cross-Platform, Python-Agnostic Binary Package Manager. http://conda.pydata.org.

[9] R. Darlington, T. McAbee, and G. Rodrigue. A Study of ALE Simulations of Rayleigh-Taylor Instability. *Computer Physics Communications*, 135:58–73, 2001.

[10] A. DiGirolamo. The Smithy Software Installation Tool. http://github.com/AnthonyDiGirolamo/smithy, 2012.

[11] E. Dolstra, M. de Jonge, and E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th Large Installation System Administration Conference (LISA XVIII)*, LISA '04, pages 79–92, Berkeley, CA, USA, 2004. USENIX Association.

[12] E. Dolstra and A. Löh. NixOS: A Purely Functional Linux Distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 367–378, New York, NY, USA, 2008. ACM.

[13] P. F. Dubois, T. Epperly, and G. Kumfert. Why Johnny Can't Build. *Computing in Science and Engineering*, 5(5):83–88, Sept. 2003.

[14] P. J. Eby. Setuptools. http://pypi.python.org/pypi/setuptools.

[15] T. Epperly and C. White. MixDown: Meta-build tool for managing collections of third-party libraries. https://github.com/tepperly/MixDown.

[16] T. G. W. Epperly and L. Hochstein. Software Construction and Composition Tools for Petascale Computing SCW0837 Progress Report. Technical report, Lawrence Livermore National Laboratory, September 13 2011. LLNL-TR-499074.

[17] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. *Technology*, 28:32, 2014.

[18] E. Foster-Johnson. Red Hat RPM Guide. 2003.

[19] J. L. Furlani. Modules: Providing a Flexible User Environment. In *Proceedings of the Fifth Large Installation System Administration Conference (LISA V)*, pages 141–152, Dallas, Texas, January 21-25 1991.

[20] J. L. Furlani and P. W. Osel. Abstract Yourself With Modules. In *Proceedings of the Tenth Large Installation System Administration Conference (LISA X)*, LISA '96, pages 193–204, Berkeley, CA, USA, 1996. USENIX Association.

[21] M. Geimer, K. Hoste, and R. McLay. Modern Scientific Software Management Using EasyBuild and Lmod. In *Proceedings of the First International Workshop on HPC User Support Tools*, HUST '14, pages 41–51, Piscataway, NJ, USA, 2014. IEEE Press.

[22] F. Groffen. Gentoo Prefix. wiki.gentoo.org/wiki/Project:Prefix.

[23] K. Hoste, J. Timmerman, A. Georges, and S. De Weirdt. EasyBuild: Building Software with Ease. In *High Performance Computing, Networking, Storage and Analysis, Proceedings*, pages 572–582. IEEE, 2012.

[24] M. Howell. Homebrew, the Missing Package Manager for OS X. http://brew.sh.

[25] 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)*, Helsinki, Finland, May 2008.

[26] Lawrence Livermore National Laboratory. Linux at Livermore. https://computing.llnl.gov/linux/.

[27] R. McLay. Lmod: Environmental Modules System. https://www.tacc.utexas.edu/research-development/tacc-projects/lmod.

[28] R. McLay. Lmod Tutorial. Presented at University of Ghent. http://hpcugent.github.io/easybuild/files/sllides_-mclay_20140617_Lmod.pdf, 2014.

[29] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.

[30] H. Meuer, E. Strohmaier, J. Dongarra, and S. Horst. Top 500 Supercomputer Sites. http://www.top500.org.

[31] B. Morgan and J. Greenough. Large-Eddy and Unsteady RANS Simulations of a Shock-Accelerated Heavy Gas Cylinder. *Springer-Verlag Berlin Heidelberg*, 2015.

[32] T. Nagy. WAF. http://github.com/waf-project/waf.

[33] T. F. Project. Yellowdog Updater, Modified (YUM). http://yum.baseurl.org.

[34] D. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. In *Proceedings of Conference on Parallel Compilers (CPC2000)*, Aussois, France, January 2000.

[35] M. Rosenblum. VMware's Virtual Platform. In *Proceedings of Hot Chips*, pages 185–196, 1999.

[36] G. N. Silva. APT Howto. Technical report, Debian, 2001. http://www. debian. org/doc/manuals/apt-howto.

[37] J. E. Smith and R. Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, 2005.

[38] The FreeBSD Project. About FreeBSD Ports. http://www.freebsd.org/ports/.

[39] The MacPorts Project. The MacPorts Project Official Homepage. http://www.macports.org.

[40] The ROCKS Group. ROCKS:Open Source Toolkit for Real and Virtual Clusters. http://www.rocksclusters.org.

[41] G. K. Thiruvathukal. Gentoo Linux: The Next Generation of Linux. *Computing in Science and Engineering*, 6(5):66–74, 2004.

[42] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. Huff, I. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson. Best Practices for Scientific Computing. *CoRR*, abs/1210.0530, 2012.