The Spack Package Manager: Bringing order to HPC software chaos

Todd Gamblin
tgamblin@llnl.gov

Matthew Legendre
legendre1@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 exponential in size.

We introduce Spack, a package manager used at Lawrence Livermore National Laboratory (LLNL) 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 a diverse and demanding user base, 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 first and third on the Graph500 [3] and Top500 [28] lists, respectively. The simulation software that runs on these machines is very complex; some codes depend on specific versions of over 50 dependency libraries. They require specific compilers, build options and 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 application 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), Message Passing Interface (MPI) implementation, compilers, and other dependencies. Unfortunately, building scientific software is notoriously complex, with immature build systems that are difficult to adapt to new machines [11, 21, 37]. Worse, the space of required builds grows combinatorically 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 [1, 8, 9, 10, 21, 22, 34, 35, 36]. For the most part, these systems 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 [1, 8, 9, 10, 21], leading to a profusion of files and maintenance issues. To our knowledge, no existing tool allows new configurations to be composed and assembled on demand. Some allow limited forms of composition [9, 10, 21], but their dependency management is overly rigid, and they burden users with the combinatoric problems of naming and versioning.

In this paper, we describe our experiences with the *Spack* package manager, which we have developed at LLNL to manage increasing software complexity. Specifically, this paper offers the following contributions:

1. A novel, recursive syntax for concisely specifying and querying 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. Three use cases that detail LLNL's use of Spack to compose software configurations rapidly, to manage Python installations, and to implement site-specific build policies.

Spack is in active development at LLNL, and our use cases outline some of its first applications in our environment. 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 [2, 13, 14, 29] are related to package managers, but they focus on ensuring that a single package builds correctly 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 combinatoric versioning.

Traditional package managers.

Package managers automate the installation of complex sets of software packages. *Binary package managers* such as RPM, yum, and APT [16, 30, 32] are integrated with most OS distributions, and they are used to ensure that dependencies are installed before packages that require them. 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 only has a single version and most of the tools install packages in a single, inflexible location. To install multiple configurations, the user must create a custom, combinatoric naming scheme 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 [20, 22, 34, 35, 36] build packages from source instead of installing from a pre-built binary. Most existing 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 [20], but again the burden is on package creators to manage conflicts. This burden effectively restricts installations to few configurations.

Virtual Machines and Containers.

Packaging problems arise in HPC because a supercomputer's hardware, OS, and filesystem are shared by many users with different requirements. The classic solution to this problem is to use virtual machines (VMs) [4, 31, 33] or lightweight virtualization techniques like Linux containers [15, 27]. This model allows each user to have a personalized environment with its own package manager, and it has been extremely 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. Security is another major concern: a profusion of VMs makes mandatory patching difficult when user's environments differ significantly from those of the facility. Perhaps most importantly, significant differences in versioning leaves the majority of the installation burden on end 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 [24] for this purpose, which is managed by RPM.

For more specific builds, many sites adhere to detailed naming conventions that encode information in file system paths. Table 1 shows conventions at several sites. At LLNL, we use the APT package manager for installs in the `/usr/local/tools` file system; we use `/usr/global/tools` for manual installs. ORNL uses hand installs but adheres to a strict scripting structure to reproduce each build [23]. TACC relies heavily on locally maintained RPMs for its installs.

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 the full set of 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, or 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 (often not the same person who installed the binary) must typically add dependency library paths to `LD_LIBRARY_PATH` (or a similar environment variable) so that the loader can find them.

Many HPC sites address this problem using *environment modules*, which allow users to "load" and "unload" such settings dynamically using simple commands. Environment emerged in 1991, and there are many implementations available [5, 17, 18, 25, 26]. The most advanced implementations, such as Lmod [25, 26], provide software hierarchies similar to the naming conventions in Table 1. They 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`. `RPATHs` and environment modules are not mutually exclusive. Modules can still be used to set `MANPATH`, `PATH` and other variables, but adding `RPATHs` ensures that binaries run correctly regardless of whether the right module is loaded. LC installs software with both `RPATHs` and `dotkit` [5] modules.

Modern Package Managers.

Recently, a number of package managers have been developed to manage multi-configuration builds. Nix [9, 10] is a package manager and an OS distribution that supports the installation of arbitrarily many software configurations.

Site	Naming Convention
LLNL	`/ usr / global / tools / $arch / $package / $version` `/ usr / local / tools / $package-$compiler-$build-$version`
Oak Ridge [23]	`/ $arch / $package / $version / $build`
TACC/Lmod [26]	`/ $compiler-$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.

As at most HPC sites, each package in Nix is installed in a unique prefix, but Nix does not have a human-readable naming convention. Instead, Nix determines the prefix by hashing the package file along with its dependencies.

HashDist [1] is a meta-build system and package manager for HPC. Like Nix, it uses cryptographic versioning, and it archives installations in a git repository so that they can be recalled on demand. Both Nix and HashDist use `RPATHs` in their packages to ensure that libraries are found correctly.

The EasyBuild [21] tool is in production use at the University of Ghent. It allows multiple versions to be installed at once. Rather than setting `RPATHs`, it auto-generates module files to manage each package's environment, and it is closely coupled with Lmod [19]. EasyBuild groups the compiler, MPI, FFT, and BLAS libraries together in a *toolchain* that can be combined with any package file. This provides some degree of composability and separates compiler flags and MPI concerns from client packages.

Smithy [8] is a similar tool in use at ORNL. 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.

Gaps in current common practice.

The cryptographic versioning of Nix and HashDist is very flexible. It versions the package *and* its dependency configuration, and can represent any configuration. However, users cannot navigate or query the installed software. The systems are in a sense "write-only": Nix does not even offer a way to ask for a package's dependencies. EasyBuild and Smithy generate environment modules, which allow querying through their own interface. Naming schemes used in existing module systems, however, cannot handle combinatoric versions. The Lmod authors call this the "matrix problem" [26].

The main limitation of existing tools is the lack of build *composability*. The full set of package versions is combinatoric, and realizing an arbitrary combination of compiler, MPI version, build options, and dependency versions requires tediously modifying many package files. Indeed, the number of package files required for existing systems scales with the number of version *combinations*, not the number of packages, which quickly becomes unmanageable. As an example, the very flexible EasyBuild system has over 3,300 files for several permutations of only 600 packages. Other systems have similar issues. HPC sites need a mechanism to *parameterize* packages so that new builds can be *composed* on the fly.

3. THE SPACK PACKAGE MANAGER

Based on our experiences at LLNL, we have developed *Spack*, the Supercomputing PACKage manager. Spack is written in Python. We chose Python for its flexibility and its wide use in the HPC community. 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 language to specify and manage the combinatorial space of HPC software configurations. Spack has the following unique capabilities:

1. Packages are **parameterized**, allowing new builds to be composed on the fly for different versions, architecture, compiler, options, and dependencies.

2. To manage this large parameter space, spack provides a novel, recursive **spec syntax** for dependency graphs.

3. Specs support versioned, ABI-incompatible interfaces like MPI through **versioned virtual dependencies**.

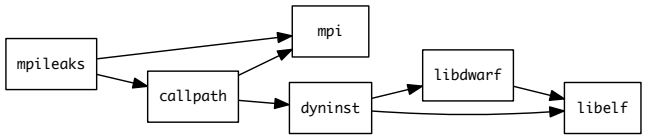
4. Spack builds with **compiler wrappers** that enforce build consistency and simplify package writing.

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

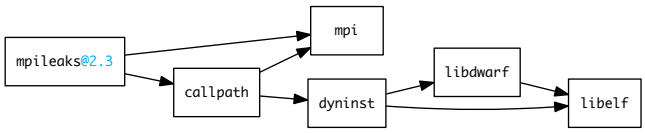
Figure 1: Spack package for the `mpileaks` tool.

3.1 Packages

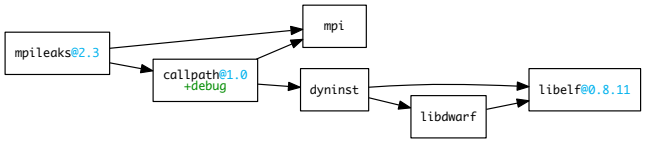
In Spack, packages are Python scripts that build software artifacts. Each package is a class that extends a generic `Package` base class. `Package` implements the bulk of the build process, but subclasses provide their own `install` method to handle the specifics of particular packages. The subclass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



(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.

Figure 1 shows the package for `mpileaks`, a tool developed at LLNL to find leaks of MPI objects in programs. The `MpiLeaks` class first provides simple metadata: a text description, a homepage, and a download URL. Next, two `version` directives identify known versions and provide MD5 checksums used to verify downloads. Below this, two `depends_on` directives indicate prerequisite packages that must be installed before `mpileaks`. Last, the `install()` method contains the commands used to build the tool. Spack's DSL allows shell commands to be invoked as Python functions, and the `install()` method invokes `configure`, `make`, and `make install` commands 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 built in many different ways. The build is *parameterized* so that it can be configured many different ways. Spack calls a single build configuration a *spec*, and the `spec` object passed to `install()` is how the Spack system encapsulates build parameters for package authors.

3.2.1 Structure

To understand how specs work, consider the `mpileaks` package structure. Metadata in the `Mpileaks` class (`version`, `depends_on`, etc.) describe its relationships with other packages. There are two direct dependencies: the `callpath` library and `mpi`. Spack recursively inspects the class definitions for each dependency and constructs a graph of their relationships. The result is a directed, acyclic graph (DAG) of dependencies[1]. To guarantee a consistent build, and to avoid ABI incompatibility, the DAG is constructed so that there is only *one* version of any particular package. Note the distinction: 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.

[1]Spack currently disallows circular dependencies.

$$
\begin{array}{lll}
\langle spec \rangle & ::= & \langle id \rangle \ [\ \text{constraints}\ ] \\
\langle constraints \rangle & ::= & \{\ \text{`@'}\ \langle version\text{-}list \rangle \mid \text{`+'}\ \langle variant \rangle \\
 & & \mid \text{`-'}\ \langle variant \rangle \quad \mid \text{`\~{}'}\ \langle variant \rangle \\
 & & \mid \text{`\%'}\ \langle compiler \rangle \quad \mid \text{`='}\ \langle architecture \rangle\ \} \\
 & & [\ \langle dep\text{-}list \rangle\ ] \\
\langle dep\text{-}list \rangle & ::= & \{\ \text{`\^{}'}\ \langle spec \rangle\ \} \\
\langle version\text{-}list \rangle & ::= & \langle version \rangle\ [\ \{\ \text{`,'}\ \langle version \rangle\ \}\ ] \\
\langle version \rangle & ::= & \langle id \rangle \mid \langle id \rangle\ \text{`:'} \mid \text{`:'}\ \langle id \rangle \mid \langle id \rangle\ \text{`:'}\ \langle id \rangle \\
\langle compiler \rangle & ::= & \langle id \rangle\ [\ \langle version\text{-}list \rangle\ ] \\
\langle variant \rangle & ::= & \langle id \rangle \\
\langle architecture \rangle & ::= & \langle id \rangle \\
\langle id \rangle & ::= & \text{[A-Za-z0-9\_][A-Za-z0-9\_.-]*}
\end{array}
$$

Figure 3: EBNF grammar for spec expressions.

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 to build with, 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 it is not reasonable to expect users to understand or 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` 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: there are simply too many parameters. However, the small set of important build constraints can be very specific, so we have two competing concerns. We need the ability to specify details of the configuration space, without the complexity of remembering all of them.

3.2.3 Spec Syntax

We have developed a syntax for specs that allows users to specify only those constraints they care about, but to be specific when necessary. Our syntax is expressive enough to represent software DAGs but concise enough to use on the command line. The spec syntax is recursively defined to allow users to specify parameters on dependencies as well as on the root of the DAG. The EBNF grammar we use to implement this in Spack is shown in Figure 3.

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 the `spack install` command:

```
$ spack install mpileaks
```

Here, `mpileaks` is the simplest possible spec—a single identifier. Spack parses it and converts it into the DAG shown in Figure 2a. Note that even though the spec contains no dependency information, it is still converted to a full DAG, based on the directives supplied in package files. Since there are no constraints on the nodes, Spack has considerable leeway for how to build the package, and we say that the package is *unconstrained*. Now, suppose the user wants a specific

	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.

version of `mpileaks`. This can be requested with a version constraint after the package name:

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

We see from Figure 2b that the specific version constraint is placed on the `mpileaks` node in the DAG, but the rest of the DAG remains unconstrained. If the user does not need a specific version of `mpileaks`, but does require particular minimum version, then the user could use *version range* syntax and write `@2.3:`. Likewise, for a version between 2.3 and 2.5.6, she would use `@2.3:2.5.6` to designate the range. In these cases, the user can save build time if Spack already has a version installed that satisfies the constraint – Spack will just 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. In the DAG, we now see that there are version constraints on `callpath` and `libelf`, and that the user has asked for the debug variant of the `callpath` library.

Recall that Spack guarantees there will be only a single version of any package in the spec DAG. Therefore, within the same DAG, each dependency can be uniquely identified by only its package name. The user does not have to think about DAG connectivity to add constraints. She need only know that a package depends, somehow, on `callpath`. For the same reason, the constraint order does not matter; dependency constraints can appear in arbitrary order.

Table 2 shows further examples of specs, ranging from very simple to more complex. From these examples, we can see that Spack offers constraint notation to cover the rest of the HPC package parameter space.

Versions. Version constraints, denoted with `@`, were already covered above. 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. Package managers frequently lag behind the latest releases. Spack has a capability to extrapolate URLs from versions, using the package's `url` attribute as a model[2]. The user can request a specific version on the command line, even if it is unknown to Spack, and Spack will attempt to install it. Spack also uses the same model to scrape webpages and find new versions as they become available.

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

Variants. To handle build options like compiler flags or optional components, specs can have named flags, or *variants*. Variants are associated with the package, so the `mpileaks` package implementor must check the spec and handle the cases where debug is enabled (`+debug`) and disabled (with `-debug` or `~debug`). The names simplify the versioning and prevent Spack's configuration space from becoming too fine-grained. It would violate our goal of conciseness, for example, to include detailed compiler flags in spec syntax, but known sets of flags can simply be named.

Cross-compilation. To support cross-compilation, Spack includes the platform in the package spec (line 5). Platforms begin with `=` and take names like `linux-ppc64` or `bgq`. They are specified per-package; this 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 being 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, and it is often desirable to write constraints like this into a package file. For example, the ROSE compiler only builds with a certain version of the `boost` library. At first glance, the `depends_on()` directives in Figure 1 look like they take 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, the ROSE compiler can simply write:

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

This constraint will be incorporated into the initial DAG node generated from the ROSE package.

Spack's `patch` directive also accepts spec syntax as a predicate in an optional `when` parameter. For example, these concise directives in the Python package ensure that specific patches are applied to the Python source code when it is being built on Blue Gene/Q, with the appropriate compiler:

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

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

[2]This works for packages with consistently named URLs

```
    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.

```
# 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.

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

3.2.5 *Build specialization*

In our experience maintaining packages at LLNL, we have sometimes had to change entire package build scripts due to large changes in the way certain packages build. This can be very cumbersome, and it is difficult to maintain both the old and new version of a build script, but we must if we want to keep installing older versions for users who rely on them.

For cases like this, Spack provides functionality that allows Python functions to have multiple definitions, with some specialized for particular configurations of the package. This allows us to have two separate implementations of `install` or *any* method in a package class. Figure 4 shows how this is used in 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 put in its place. We replace the function with a callable multi-function dispatch object, and we integrate the predicate check into the function dispatch mechanism. Here, the `@when` condition is true when Dyninst is at version 8.1 or lower, and in those cases 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 source interface and can be interchanged for one another in a build. The archetypal example in HPC is the MPI interface, which has a number of open source implementations (MPICH, OpenMPI, MVAPICH, etc.) and vendor-specific implementations. An application that can be built with one MPI can generally be built with another. BLAS implementations such as ATLAS, LAPACK-BLAS, MKL, etc. are another common example.

At LLNL, we frequently need to build tools with many versions of MPI to support the many different applications that run at our center. However, MPI has no ABI, so applications cannot simply be re-linked; they must be recompiled and reinstalled. 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 by interface, Spack supports *virtual dependencies*. A virtual dependency is an abstract name representing a library interface or capability instead of a library implementation. Packages that need this capability need not depend on a specific implementation; they can depend on the virtual name, for which the user or Spack can select an an implementation at build time. Other package managers support the notion of virtual dependencies, but Spack adds *versioning* to its interfaces. This allows concepts like MPI interface versions and BLAS levels to be represented directly, and Spack can handle the details of managing complex constraints while the user focuses on composing software.

Figure 5 shows how packages provide virtual interfaces in Spack. The spec syntax is used to concisely associate 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 CFD 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 quickly specify a partially constrained software DAG. We say such a 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 all of these criteria, we say it is *concrete*. *Concretization* is the central component of the Spack build process that allows it to reduce a highly unconstrained abstract description to a concrete build.

Spack's concretization algorithm is shown in Figure 6. 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 that are encoded in directives in package files.

Spack intersects the constraints of the two DAGs package by package. 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 explicitly requests a different version from what the user requested. Likewise, if the package and the user specified different compilers, variants, or platforms for particular packages, this will cause the user to be notified of the

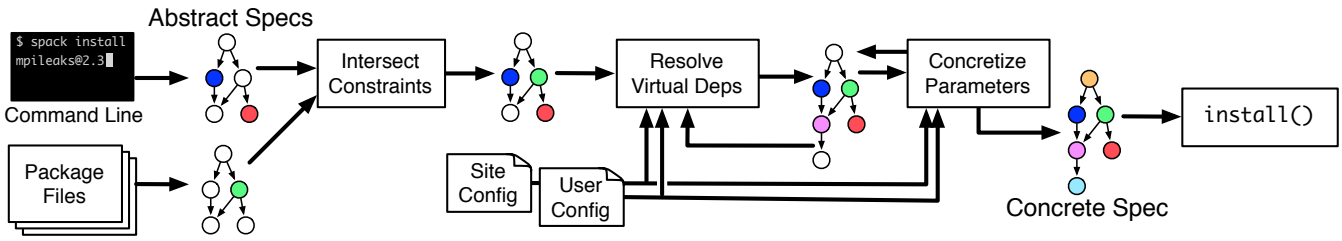


Figure 6: Spack's concretization process.

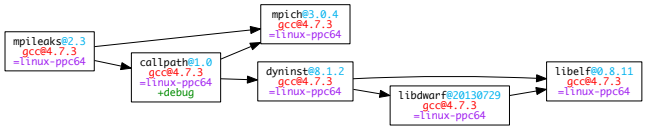


Figure 7: Concretized spec from Figure 2a.

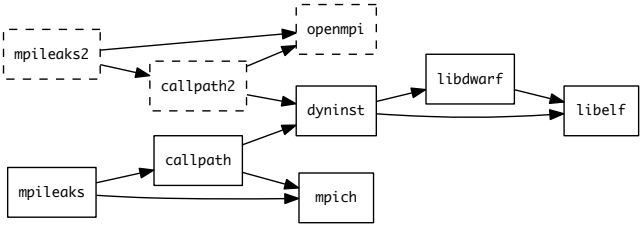


Figure 8: `mpileaks` built with `mpich`, then `openmpi`.

contradiction. If the user or package specified version ranges, they are intersected, and if the ranges do not overlap, an error will be raised. When the intersection succeeds, Spack has a single spec DAG with the merged constraints of the user and the package files.

The next part of the process is iterative. If any node in the DAG is a virtual dependency, spack replaces it with with a suitable interface provider. It does this by building a reverse index from virtual packages to providers using the `provides when` directives discussed in Section 3.3. If multiple providers could satisfy the constraints on the virtual spec, Spack will consult site and user policies to select the "best" possible provider. The selected provider may *itself* have virtual dependencies, so Spack repeats this process until there are no more virtual packages in the DAG.

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 *may* cause a package to depend on more libraries. For example, if the site default behavior is to have a particular package build with `+python`, and this causes it to depend on Python libraries, then 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 she might force the build to use a *particular* MPI implementation by supplying, e.g. `^openmpi` or `^mpich`. In future work we will investigate adding a SAT solver to Spack's concretization process to present users with better options. In our experience, however, complex policy interactions like these are very rare.

When the concretization process completes, it outputs a fully concrete spec DAG. A concrete DAG with architectures, compiler, versions, and variants, and all dependencies resolved is shown in Figure 7. 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 uses a sub-DAG rooted at current node, which is also a concrete spec. Package authors are responsible for querying the spec in `install` and handling different configurations.

3.4.1 *Directory Layout*

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. This does *not* mean that Spack rebuilds every library for each new configuration. If two configurations share a sub-DAG, then the sub-DAG's configuration will be reused. Figure 8 shows how the `dyninst` sub-DAG is used for both the `mpich` and `openmpi` builds of `mpileaks`.

3.4.2 *Reproducibility*

For reproducibility and to preserve provenance, Spack stores a file containing the concrete spec in each install prefix. The spec can be examined later to understand how the installed package was built. It can also be used to reproduce the build on another machine or at another site.

3.4.3 *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 mentioned, there are only a few parameters most users care about, and the rest only serve to add complexity to the build. When doing manual builds, LLNL staff tend to make arbitrary choices about many of the secondary build parameters, or they would include logic in build scripts to make many of these choices. In either scenario, publicly installed software was generally not installed in a consistent way.

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 about 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. There is no need for packages to contain complicated checks, or for packages 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. We believe that this 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 intended to build a consistent HPC stack for our multi-user environment, and reproducible builds are one of our design goals. Experience at LLNL has shown that it is vexingly difficult to reproduce a build manually. Many packages used at LLNL have a profusion of build options, and specifying them correctly often requires tedious experimentation. This is due to lack of build standards and to 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 package files allows us to record precise build incantations for later reuse.

Environment isolation.

In addition to command-line issues, we frequently encounter errors due to inconsistencies between the environment of the package installer and the package user. For example, there are two versions of the `libelf` library used by LLNL performance tools. One is distributed with RedHat Linux, while another publicly available version has the same API but an incompatible ABI. Failure to specify the right version at build time has caused many unexpected crashes. Spack manages the build environment by running each `install` invocation in a new process. It helps packages find dependencies correctly, by setting `PATH`, `PKG_CONFIG_PATH`, `CMAKE_PREFIX_PATH`, and `LD_LIBRARY_PATH` to include the dependencies of the current build. These variables are commonly used by build systems to locate dependencies, and setting them helps to ensure that correct libraries are detected. The isolated build environment also gives package authors free reign to set build-specific environment variables without interfering with other packages.

Compiler wrappers and RPATHs.

Finding compilers at build time is not the only obstacle to reproducible behavior. As mentioned in Section 2, it is also important for binaries to be able to find dependency libraries at *runtime*. One of the most frequent user errors at LC is improper library configuration. Users frequently do not know what libraries a package was built with, and it is difficult for them to construct a suitable `LD_LIBRARY_PATH` for a package that was built by someone else. Because of frequent support calls, we typically add `RPATHs` to public software installations, so that paths to dependencies are embedded in binaries and so that users do not have to know this information to run installed software correctly.

Spack manages `RPATH` settings and other build policies with *compiler wrappers*. In each isolated `install` environment, Spack sets the standard environment variables `CC`, `CXX`, `F77`, and `FC` to point to its own compiler wrapper scripts. These variables are used by most build systems to select C, C++, and Fortran compilers, so they are generally 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 library installations, where needed headers and libraries are located. The wrappers then invoke the real compiler with the modified arguments.

Spack's compiler wrappers have a number of useful effects. First, they allow Spack to transparently switch compilers in most builds. This is how compiler options like `%gcc` are implemented. Second, they enforce the use of `RPATHs` in installed binaries. This causes applications built by Spack 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 are run by most build systems succeed automatically, *without* the need to use special arguments for nonstandard locations. `configure` commands in Spack's `install` function can have fewer arguments, and can be written as they would be for system installs. This reduces complexity for package maintainers and enforces consistent, reproducible build policies across packages. Finally, because Spack has control over the wrappers, package authors can programmatically filter the compiler flags used by software build systems, a useful last resort when porting to bleeding-edge platforms or new, esoteric compilers.

3.5.1 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`, `LD_LIBRARY_PATH`, or `MANPATH` 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 [5] and Module configuration files for its packages, allowing users to setup their runtime environment using familiar systems. Future versions of Spack may also allow the creation of Lmod [25] hierarchies, as discussed in Section 2. Spack's rich dependency information would allow such hierarchies to be generated automatically.

4. USE CASES

We have recently begun using Spack in production at Livermore Computing. 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 Combinatoric 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` for their preferred compilers, but this task was

[3]If builds do not respect `CC`, `CXX`, etc., 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 9: Simplified install routine for gperftools.

complicated due to the second issue.

Spack presented a solution to both problems. Package administrators can use Spack to easily maintain a central install of `gperftools` across combinations of compilers and compiler versions. Spack's `gperftools` package also serves as a central institutional knowledge repository for builds, as its package files encode the patches and configure lines required for each platform and compiler combination. We are investigating whether we can move some of these settings into compiler and architecture descriptors to further simplify Spack's build template for cross-platform installations.

Figure 9 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 we have used as 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 implementation 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 quickly compose a new `mpileaks` build 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, and maintenance of 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 are either language-specific like `easy_install` [12], or they do not effectively support building from source [6, 7], let alone multi-configuration builds like Spack's. More glaringly, the default model for Python extensions is to install them *directly* into the Python interpreter's prefix, but this makes it impossible to install multiple versions. An alternative is to install Python extensions in their own prefix, but then users must manually add each package to their `PYTHONPATH`. We wanted a way to easily manage many different versions and *also* allow teams to have a baseline set of Python packages available by default.

To support this mode of operation, we added the concept of `extension` packages to Spack. Python modules now 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. Additionally, we added the capability to `activate` and `deactivate` an extension within its dependent Python installation. During the activate operation, each file in the module prefix is symbolically linked into the Python installation prefix, as though it were directly installed within the Python instance. If any file conflict would arise from this operation, the activate operation fails and notifies the user. The deactivate operation removes the symbolic links and restores the Python installation to its pristine state.

There were additional complications because many Python packages *install their own package manager* if they do not find `easy_install` in the Python installation. There are also no fewer than three ways that Python packages add themselves to the interpreter's default path, and some of these conflict. We modified Spack so that extendable packages, like Python, can supply custom code in their package file that is executed during activation and deactivation to handle any necessary language-specific tasks. Python uses this feature to merge conflicting files that are generated by the `easy_install` tool.

The end result is that Python extensions can now 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. To experiment with these extensions, users can load environment modules generated by Spack. If they want a particular version to be available without any special environment settings, they can activate it within the Python instance.

Spack now essentially implements a "meta package-manager" for each Python instance, and it can live together with Spack's normal installation model. This has allowed us to efficiently support our application teams, for whom we can now rapidly construct custom Python installations. We have also reduced the amount of time that LC staff spend installing Python modules. Because Spack packages can extend the activation and deactivation mechanisms, we believe the same mechanism could be used with other interpreted languages with similar extension models, such as R, Ruby, or Lua.

4.3 User and Site Policies

Spack makes it easy for package maintainers to create and organize package installations. But it also needs to be easy for end-users to find and use those packages. Different end-users may have different expectations about how packages should be built and installed, and those expectations are typically shaped by years of site policies, personal preferences, and lingering legacy decisions originallly made on a DEC VAX. Rather than try to dictate these expectations, Spack allows both end-users and package maintainers to create custom policies dictating how packages are built and installed at their site.

4.3.1 Package Views

While Spack can easily create many installations of a package like `mpileaks` for different compilers and MPI implementations, end users may find Spack's directory layout

confusing, and they may not be able to find the right libraries. As discussed in Section 3.5.1, environment modules are commonly used on supercomputer systems to solve this problem, and Spack allows the package author to automatically create module and dotkit files for packages that Spack installs.

However, even when modules and dotkits are available, many users will go directly through the filesystem to access package installs. Spack installs packages into paths based on concretized specs, which is ideal for maintaining multiple package installations, but may be difficult for an end-user to navigate. A version of `mpileaks`, for example, may be installed in a location like

```
spack/opt/chaos_5_x86_64_ib/gcc@4.9.2/mpileaks@1.0-db465029
```

Spack thus allows the creation of views, which are a symbolic-link based directory layout of packages. Views provide a human-readable directory layout, which can be taylored to fit alongside existing directory layouts or used to create new layouts. 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 `mpilleaks`. 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 (which is useful for users who don't want to hardcode a specific 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 through a `spackconfig` file, which can be set up at the site or user level. For each package or set of packages, the `spackconfig` file contains rules describing 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 or removal, links are automatically created, deleted, or updated according to these rules.

Spack's views are a projection from a point in a high-dimension space (the concretized spec, which fully specifies all parameters) to point in a lower-dimension space (the link name, which may only specify a few parameters). Thus several package 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. Spack defines a sort order for packages, which arranges packages from "most desired" to "least desired". When a symbolic link could resolve to multiple package installations, Spack points the link at the most desired package.

By default, Spack treats newer versions of packages compiled with newer compilers as more desirable than older packages built with older compilers. It has well-defined, but not necessarily meaningful, preferences for deciding between MPI implementations and different compilers. A user can override Spack's default sort order through their `spackconfig` file. 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 `spackconfig` file, which would cause the ambigious `mpileaks` link to point to an `icc` installation. Any compiler left out of the compiler_order setting is treated as less preferred than explicitly stated compilers. Spack can also be configured to treat specific package versions as more preferred than other versions, which can be useful if a new version is unstable and untested, and should not be advertised with symbolic links.

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. Via the `spackconfig` file, users can specify additional search directories for finding additional `Package` classes. 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 the Spack's default packages, allowing sites to either tweak or completely replace Spack's build recipes. To continue the previous example, a site can write a `LocalSpindle` Python class, which inherits from Spack's `Spindle` class. `LocalSpindle` may simply add additional configure flags to the `Spindle` class, 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.

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 tools exist that can handle multi-configuration installs, none of them addresses the combinatorial nature of the software configuration space directly. None of them allows a user to rapidly *compose* new parametric builds.

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 the *concretization* process, which converts an underspecified build DAG into a buildable spec.

We showed through three use cases that Spack is already increasing operational efficiency in production at LLNL. We believe that the software management techniques implemented in Spack are applicable to a broad range of HPC facilities. Spack is available online at: http://bit.ly/spack-git.

Acknowledgements

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

References

[1] hashdist, 2012. http://github.com/hashdist/hashdist.

[2] J. F. Amundson. Contractor meta-build system. http://bit.ly/mbcontractor.

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

[4] 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.

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

[6] 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/.

[7] Continuum Analytics. Conda: a cross-platform, python-agnostic binary package manager. http://conda.pydata.org.

[8] A. DiGirolamo. The Smithy installation tool, 2012. http://github.com/AnthonyDiGirolamo/smithy.

[9] 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.

[10] 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.

[11] P. F. Dubois, T. Epperly, and G. Kumfert. Why johnny can't build. *Computing in Science and Engineering*, 5(5):83–88, Sept. 2003.

[12] P. J. Eby. Setuptools and easy_install. http://pypi.python.org/pypi/setuptools.

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

[14] 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.

[15] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

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

[17] 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.

[18] 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.

[19] 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.

[20] F. Groffen. Gentoo Prefix. Online. https://wiki.gentoo.org/wiki/Project:Prefix.

[21] 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.

[22] M. Howell. Homebrew, the missing package manager for OS X. http://brew.sh.

[23] 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.

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

[25] R. McLay. Lmod: Environmental Modules System. http://bit.ly/tacc-lmod.

[26] R. McLay. Lmod Tutorial Slides. Presented at Ghent University, 2014. http://bit.ly/lmod-tutorial-slides.

[27] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[28] H. Meuer, E. Strohmaier, J. Dongarra, and S. Horst. Top 500 Supercomputer Sites.

[29] T. Nagy. Waf. http://github.com/waf-project/waf.

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

[31] M. Rosenblum. Vmware's virtual platform. In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.

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

[33] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

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

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

[36] G. K. Thiruvathukal. Gentoo linux: the next generation of linux. *Computing in Science and Engineering*, 6(5):66–74, 2004.

[37] 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.