Spack:
The road to 1.0

InPEx 2025

Hayama, Japan
April 15, 2025

THE LINUX FOUNDATION

The most recent version of these slides can be found at:
https://spack-tutorial.readthedocs.io

Spack v0.23.0 was released in November

Highlights:

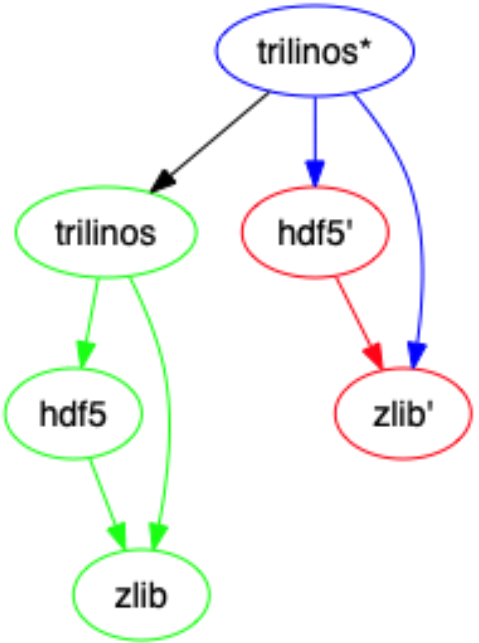
- Spec splicing
- Broader variant propagation
- Query specs by namespace
- Spack commands respect `concretizer:unify` config outside environments
- Improved formatting for `spack spec` and `spack find` commands
- `spack env track` converts environments from independent to managed
- New software stacks in CI and public binary cache
  - ML stack for linux/aarch64
  - Developer tools stack for macos/aarch64
- 329 new packages since v0.22
- Thank you to the 373 contributors to this release
  - 60 contributors to Spack core, 357 contributors to Spack packages

Spec splicing makes binary swapping possible

Reuse binary packages built against one dependency while using a new dependency.

Packages retain pointers to their original configuration for provenance

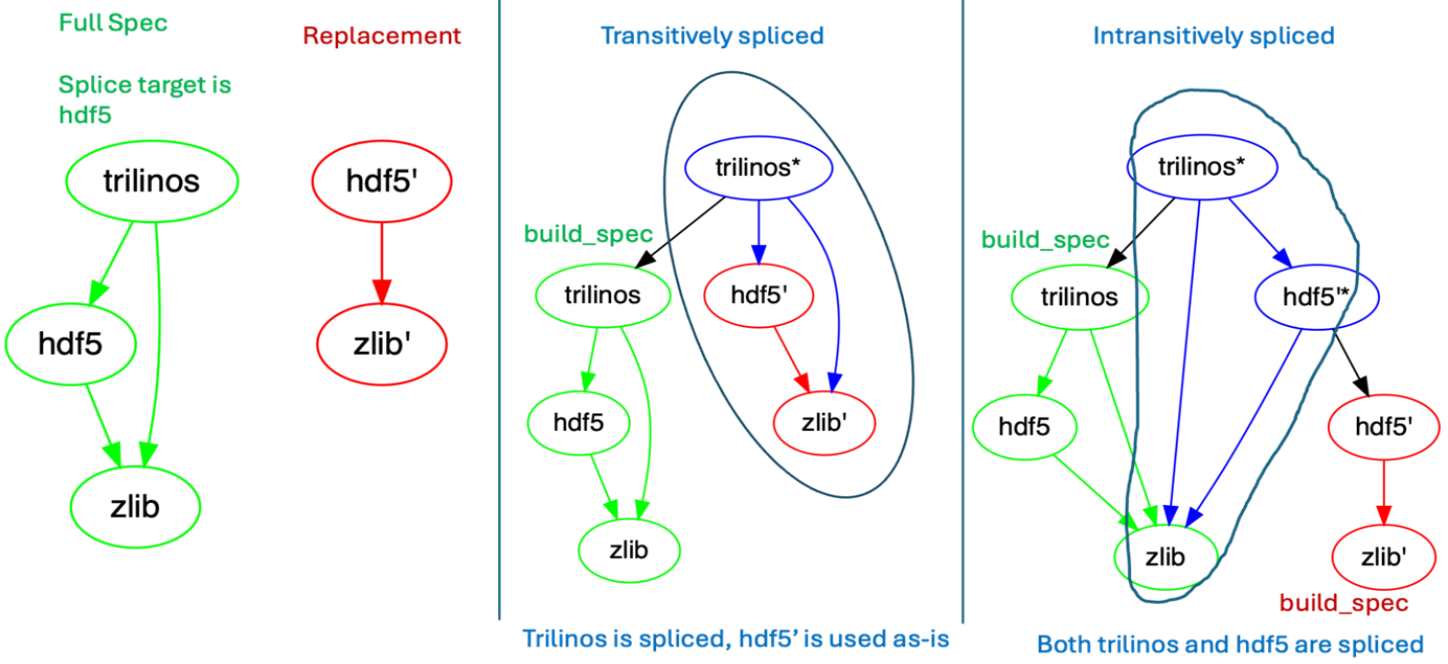
Relocation logic is repurposed for “rewiring” spec to its new configuration



Transitive and Intransitive Splices

“Transitive” splices take shared dependencies from the new dependency

“Intransitive” splices take shared dependencies from the original spec



Explicit Splicing

```
concretizer:
  splice:
    explicit:
    - target: mpi
      replacement: mvapich2/abcdef
      transitive: false
```

Any spec that concretizes to depend on mpi will be spliced to use the local mvapich2 with hash abcdef.

Explicit splicing requires the user to ensure ABI compatibility

Automatic Splicing

```
concretizer:
  splice:
    automatic: true
```

Packages have a new directive `can_splice`

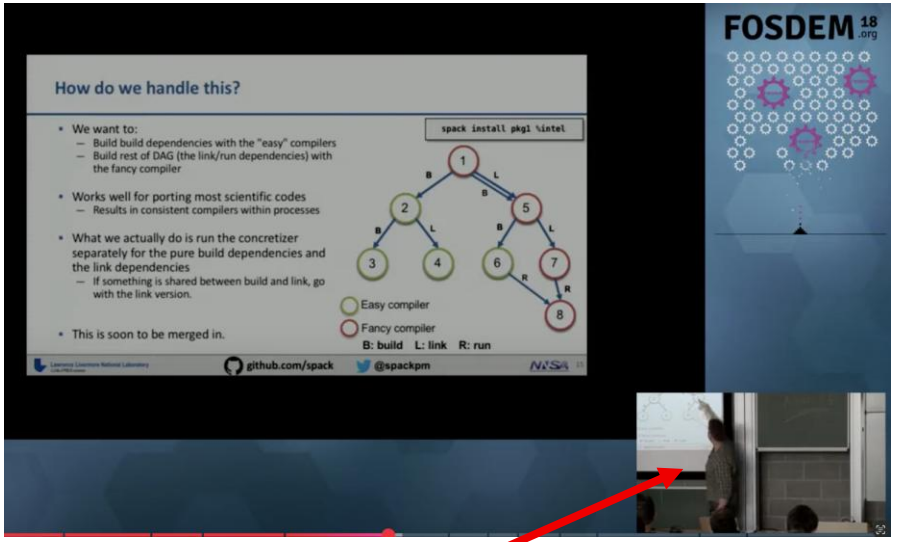
```
can_splice("foo@1.1+a", when="@1.1", match_variants=["bar"])
```

“This package at version 1.1 can be spliced in for any package that satisfies `"foo@1.1+a"` as long as the “bar” variant values are equal

If splicing is enabled, the concretizer will apply these constraints and optimize for package reuse.

The road to v1.0 has been long

- We wanted:
  - ☑ New ASP-based concretizer
  - ☑ Reuse of existing installations
  - ☑ Stable production continuous integration
  - ☑ Stable binary cache
  - 🔵 Compiler dependencies (nearly done!)
  - 🟡 Stable package API
  - 🟡 Separate builtin repo from Spack tool
- v1.0 will:
  - Change the spec model for compilers
  - Enable users to use entirely custom packages
  - Improve reproducibility
  - Improve stability 🤞
- This is the largest change to Spack since the new concretizer



Me, presenting how simple all this would be at FOSDEM in 2018

Introducing Language Dependencies

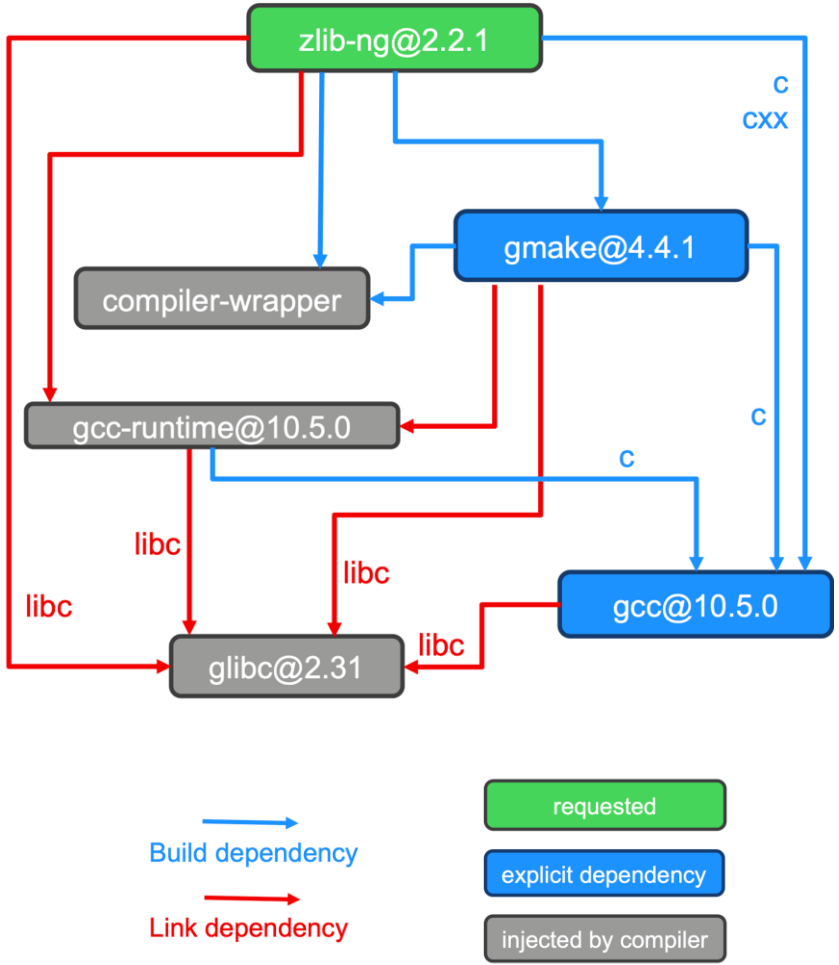
```
depends_on("c", type="build")
depends_on("cxx", type="build")
depends_on("fortran", type="build")
```

- You now need to specify these to use c, cxx, or fortran
  - No-op in the release as we prepare for compilers as dependencies
  - Backported to v0.22 release to assist teams working across Spack releases

- Spack has historically made these compilers available to every package
  - A compiler was actually “something that supports c + cxx + fortran + f77”
  - Made for a lot of special cases
  - Also makes for duplication of purely interpreted packages (e.g. python)

Compiler Dependencies

- Compilers are now proper build dependencies
- Runtime libraries modeled as packages
- gcc-runtime node is injected as a link dependency by gcc
- packages depend on c, cxx,
  fortran virtuals,
  which are satisfied by gcc node
- glibc is an automatically detected external
  - Injected as a `libc` virtual dependency
  - Does not require user configuration
  - Will eventually be able to choose implementations (e.g., musl)



Configuring compilers in Spack v1.*

Spack v0.x

compilers.yaml

```
compilers:
  - compiler:
      spec: gcc@12.3.1
      paths:
         c: /usr/bin/gcc
         cxx: /usr/bin/g++
         fc: /usr/bin/gfortran
      modules: [...]
```

Spack v1.x

packages.yaml

```
packages:
  gcc:
    externals:
    - spec: gcc@12.3.1+binutils
      prefix: /usr
      extra_attributes:
        compilers:
          c: /usr/bin/gcc
          cxx: /usr/bin/g++
          fc: /usr/bin/gfortran
        modules: [...]
```

- We will provide a tool for migrating configuration
- We will still support *reading* the old configuration until at *least* v1.1
- All fields from `compilers.yaml` are supported in `extra_attributes`

Breaking changes ⚠️

1. It is no longer safe to assume every node has a compiler.
   a. The tokens `{compiler}`, `{compiler.version}`, and `{compiler.name}` in `Spec.format` expand to `none` if a Spec does not depend on C, C++, or Fortran.
   b. `spec.compiler` will default to the c compiler if present, else cxx, else fortran for backwards compatibility.
   c. The new default install tree projection is `{architecture.platform}/{architecture.target}/{name}-{version}-{hash}`

2. The syntax `spec["name"]` will only search link/run dependencies and *direct* build dependencies.
   - Previously, this would find deep, transitive deps, which was almost always the wrong behavior.
   - You can still hop around in the graph, e.g. spec["cmake"]["bzip2"] will find cmake's link dependency

3. The % sigil in specs means "direct dependency".
   - Can now say: `foo %cmake@3.26 ^bar %cmake@3.31`
   - ^ dependencies are unified, % dependencies are not

More on direct dependencies with %

- You could previously write:

```
pkg %gcc +foo      # +foo would associate with pkg, not gcc – will error in 1.0
```

- Now you'll need to write:

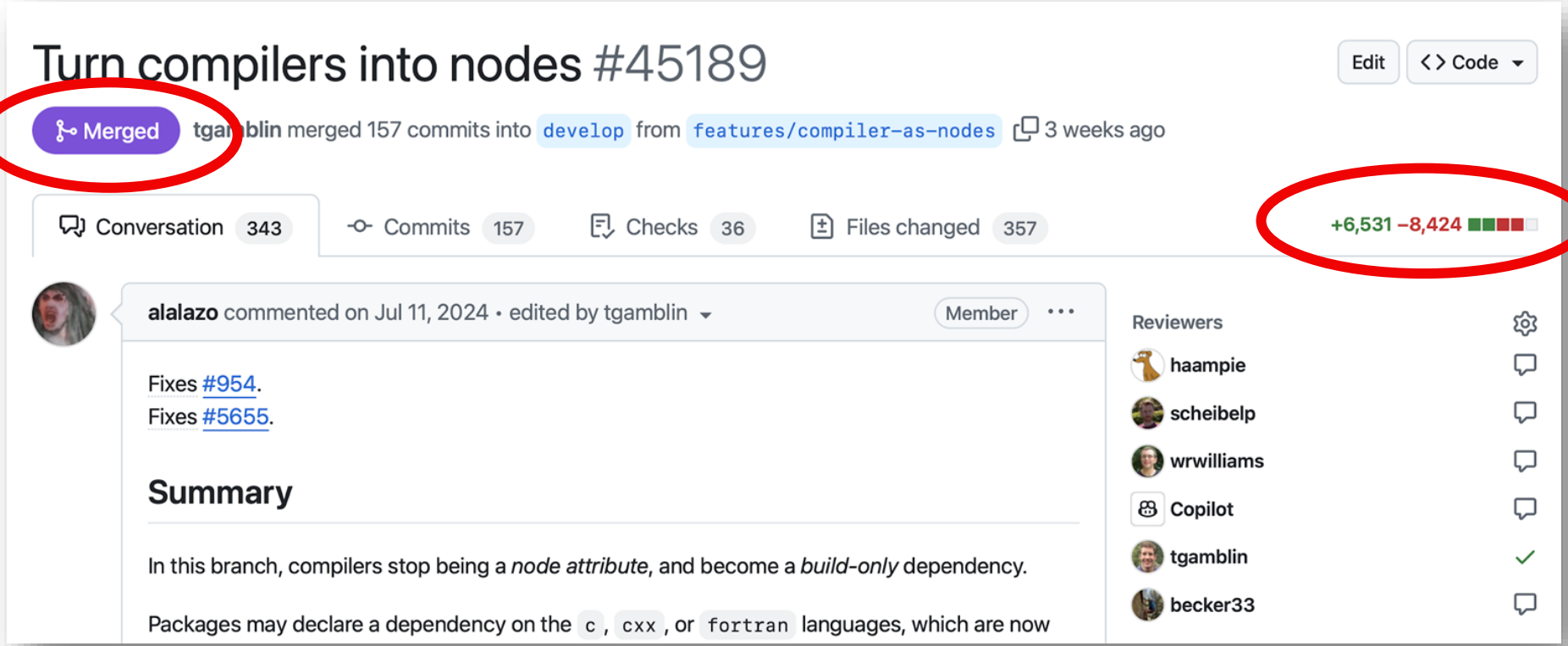
```
pkg +foo %gcc      # +foo associates with pkg
```

- We want these to be symmetric:

```
pkg +foo %dep +bar  # `pkg +foo` depends on `dep +bar` directly
pkg +foo ^dep +bar  # `pkg +foo` depends on `dep +bar` directly or transitively
```

- spack style --spec-strings --fix can remedy this automatically
  - Fixes YAML files, scripts, package.py files
  - Alternative was to have a very hard-to-explain syntax – we surveyed users and they decided it was better to break a bit than to explaining the subtleties of the first 10 years of Spack forever

It’s finally done!



Step 2: Splitting out the packages

- Spack is 2 things:
  - Core tool
  - 8,400+ package.py files

- Coupling between core and packages is tight in some places:
  1. Package base classes for using build systems are in core (cmake, autotools, etc.)
  2. Compiler wrappers used to inject flags and RPATHs are in core
  3. Package files are used after installation, e.g., at load/unload time
     - Leads to drift between old installations and package files
  4. Packages *live* in Spack's GitHub repository -- not easy to separate

We reducing coupling between packages and core

1. Build system classes are moving *into* the package repository
   - Need to provide a way to include “utility” code from package.py files
2. Compiler wrappers will *become* a package
   - Also improves build provenance and reproducibility
3. Generate shell code for environment changes *at install time*
   - Bakes load/unload logic into installations and binary packages
   - Removes need for package.py files to remember all past versions
4. Spack packages will live in a separate GitHub repository
   - Need Spack to bootstrap this new repository
   - Will need to download automatically on first install

Some complexities left to navigate

- We are adding a larger concept of a “toolchain” to Spack
  - Basically an alias for a set of compilers, runtime libs, flags, other options
  - Specified with % in its own configuration
  - Need to handle entirely as a preprocessing step – *not* in the concretizer
- Compiler wrappers have already become their own package
  - Now injected by compilers
  - Still some coupling with the build environment
    - Spack sets variables to control RPATH flags
- In some cases Spack still knows compiler and runtime library names
  - A few optimizations in the solver know about, e.g., gcc-runtime, intel runtime, etc.
  - Working to fully generify this without losing solver performance
- Some parts of our tests rely on builtin packages
  - May need to mock these, or ensure that tests auto-checkout builtin repo

v1.0 Release plan

- Done:
  - ✅ Merge compiler dependencies
  - ✅ Start fielding bug reports on develop
- April-May:
  - Split out the builtin package repository
  - Ensure bootstrapping / repo cloning is smooth
  - Toolchains
- June
  - Release v1.0 at the Spack BOF at ISC 2025

https://bit.ly/spack-changes for more

One month of Spack development

October 17, 2024 – November 17, 2024

Period: 1 month

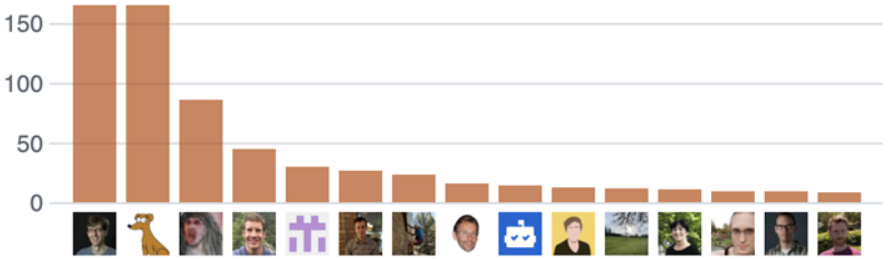
Overview

583 Active pull requests

99 Active issues

457	126	61	38
Merged pull requests	Open pull requests	Closed issues	New issues

Excluding merges, **142 authors** have pushed **461 commits** to develop and **793 commits** to all branches. On develop, **1,428 files** have changed and there have been **18,717 additions** and **9,238 deletions**.



Spack Governance

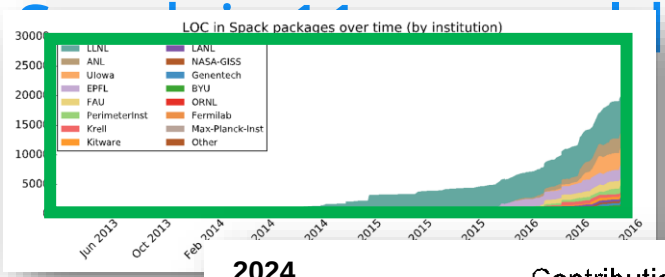
https://github.com/spack/governance

We aim to continue governing primarily through consensus

The Technical Steering Committee will vote to resolve technical discussions that cannot be resolved by consensus

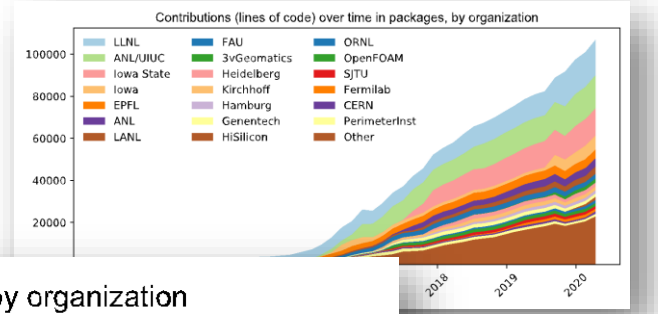
The TSC meets monthly to discuss

- Big-picture technical priorities for Spack development
- Release schedule and feature sets
- Technical disagreements requiring votes
- Pull request and issue backlog and trajectory



2016

Fall 2020



2024