Building a DSL for ABI compatibility

Scalable Tools Workshop 2022 | Spack BUILD SI

Greg Becker and Nathan Hanford

June 21, 2022

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

Lawrence Livermore National Laboratory

What would make people happy

- Can it be as easy to install scientific software as it is to install your favorite editor?
- Can it be as ***fast*** to install scientific software as it is to install your favorite editor?

The HPC software space is immense

- Not much standardization in HPC: every machine/app has a different software stack
- Sites share unique hardware among teams with *very* different requirements
  - Users want to experiment with many exotic architectures, compilers, MPI versions
  - All of this is necessary to get the best ***performance***
- Example environment for some LLNL codes:

48 third party packages

x

3 MPI versions
IntelMPI MVAPICH2
OpenMPI

x

Multiple Platforms
x86_64 aarch64 PPC64LE

x

Up to 7 compilers
Intel GCC XLC Clang
Cray NVHPC hipcc

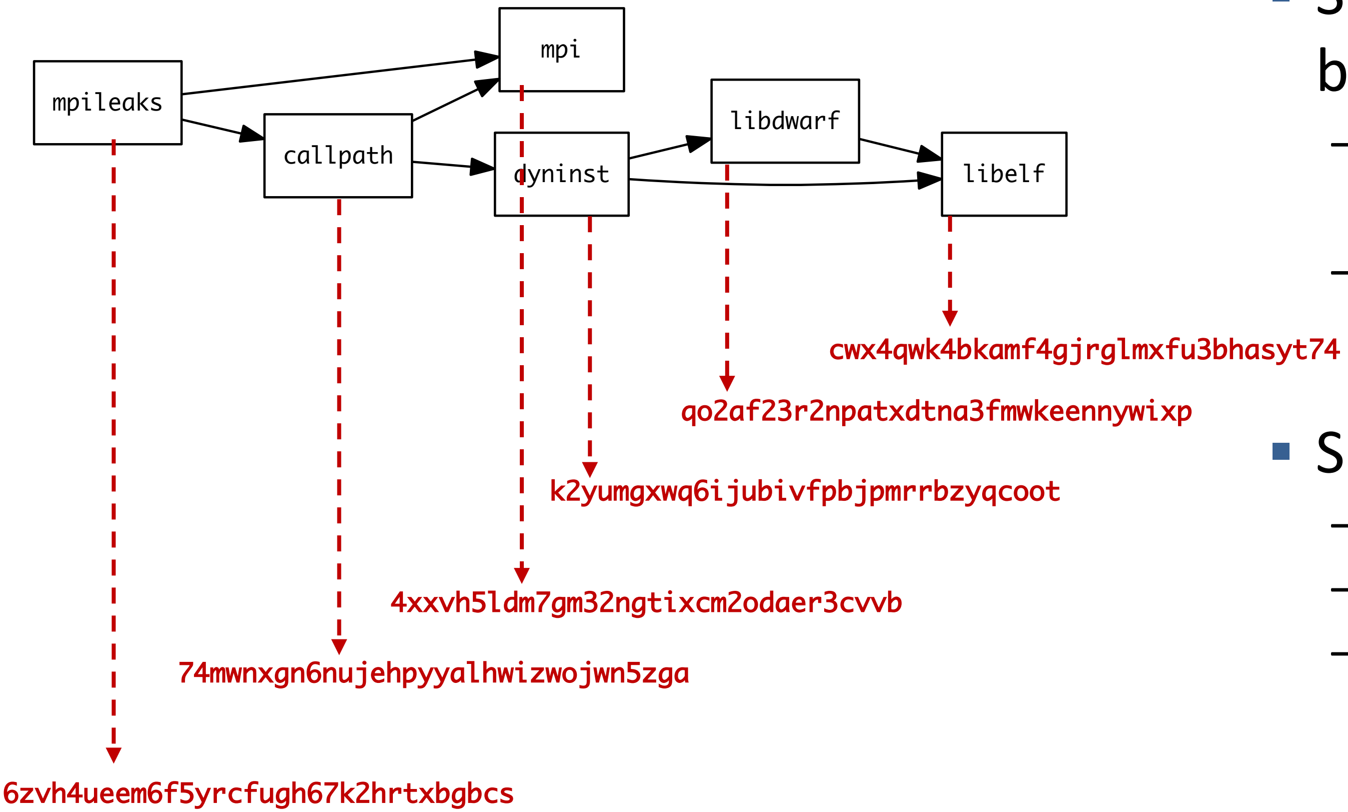
x

Oh, and 2-3 versions of each package

= **~7,500** combinations

We want an easy way to quickly sample the space, to install configurations on demand!

The Spack Software Deployment Model

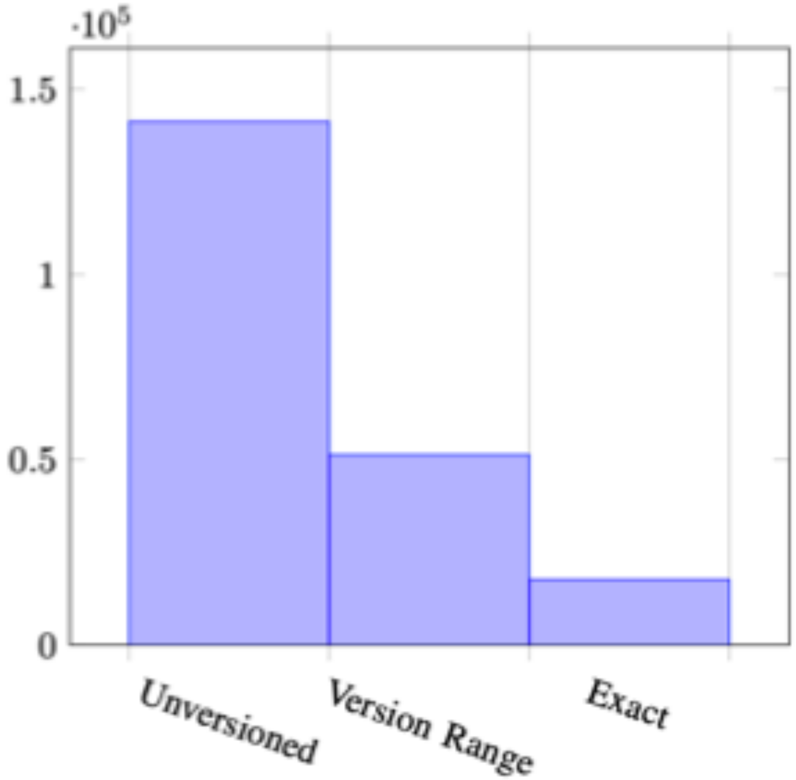


- Spack hashes represent unique *from-source* builds
  - Multiple versions of a package with built with multiple options (variants) can coexist
  - Each build gets a *Merkle hash* representing its build configuration, including all dependencies
- Spack is strict about dependencies
  - One version of any dependency per graph
  - You *must* deploy with the hashes you built with
  - If you want to change a dependency, you must rebuild, and the parent will have a new hash
- This makes it hard to swap in new binaries

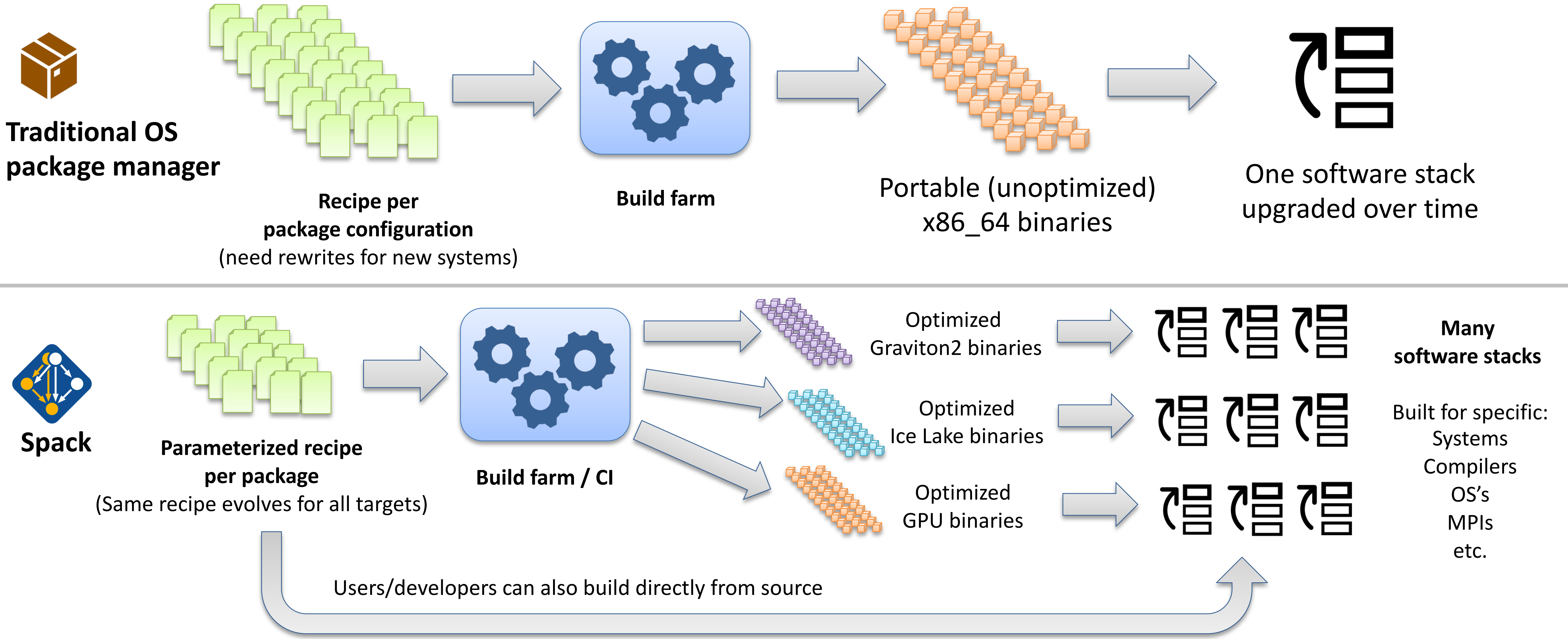
Existing package managers resolve ABI issues semi-manually

- Distro developers make sure that underlying ABI for a given OS version is stable.
- Channel authors and repo maintainers rely on build-farms to ensure that ABI doesn't break within an OS version.
  - Some (RHEL) use static analysis, like libabigail, to detect and avoid ABI breaks
  - Testing also avoids ABI breaks
- Distro maintainers apply extensive knowledge of software ecosystem and past ABI issues to decide which package versions to hold back until a new release.
- Distributed binary packages (e.g., RPM, deb) typically don't contain detailed provenance information
  - Direct dependency requirements (mostly unversioned)
  - No transitive dependency information
  - No build environment information
- **Cannot safely share an RPM across RPM-based distros** (e.g., RHEL/Fedora/SUSE)
  - Distro is a curated set of compatible packages
  - Package managers can't actually tell *what's compatible*

Debian package dependencies by type



Spack binary packages model full provenance



Traditional OS package manager

Recipe per package configuration
(need rewrites for new systems)

Build farm

Portable (unoptimized) x86_64 binaries

One software stack upgraded over time

Spack

Parameterized recipe per package
(Same recipe evolves for all targets)

Build farm / CI

Optimized Graviton2 binaries

Optimized Ice Lake binaries

Optimized GPU binaries

Many software stacks

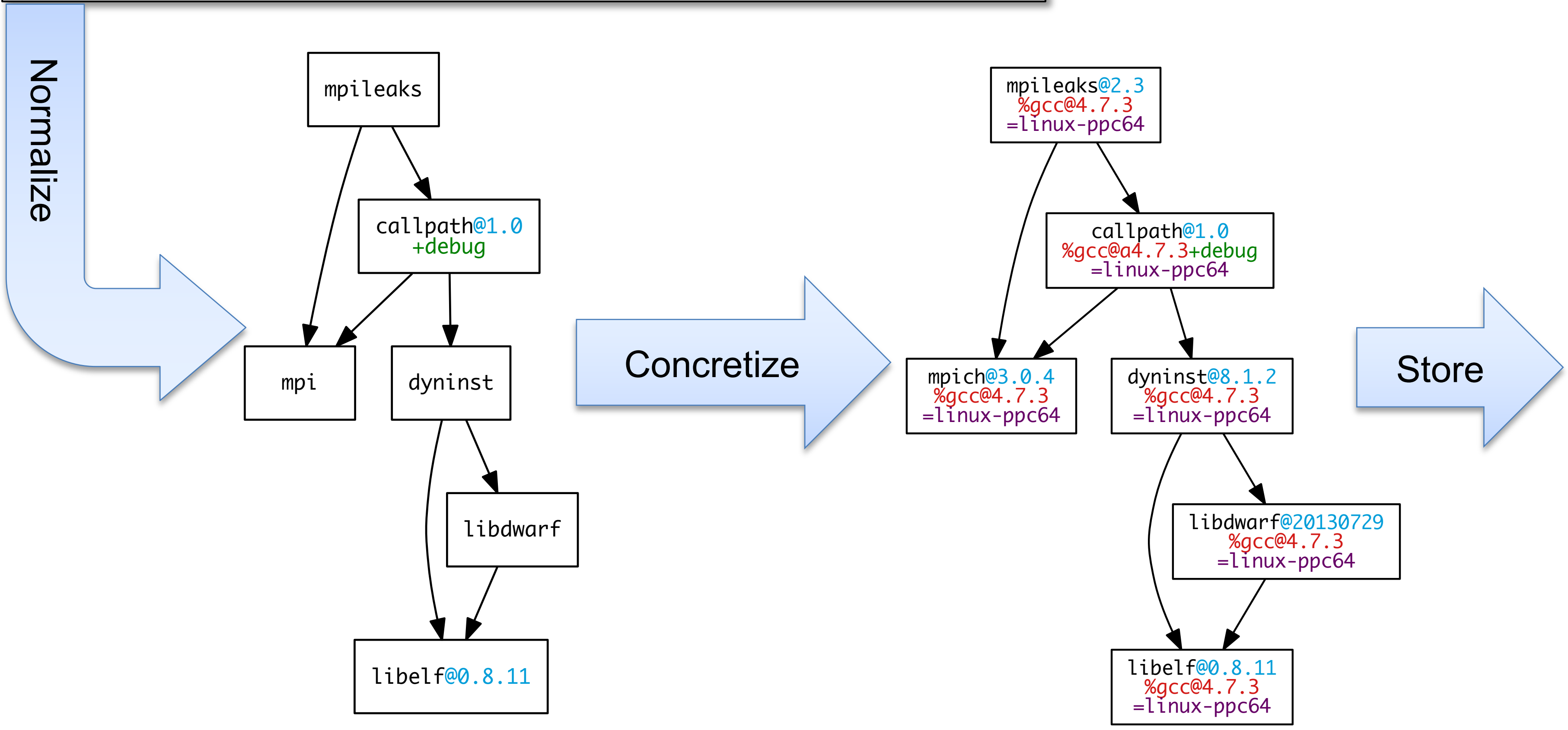
Built for specific:
Systems
Compilers
OS's
MPIs
etc.

Users/developers can also build directly from source

How Spack defines the build space

```
mpileaks ^callpath@1.0+debug ^libelf@0.8.11
```

User input: *abstract* spec with some constraints



spec.json

```
{
  "spec": {
    "_meta": {
      "version": 2
    },
    "nodes": [
      {
        "name": "mpileaks",
        "version": "1.0",
        "arch": {
          "platform": "linux",
          "platform_os": ”centos7",
          "target": {
            "name": ”power9le",
            "vendor": ”IBM",
            "features": […],
            "generation": 9,
            "parents": [”power8le"]
          }
        },
        "compiler": {
          "name": "gcc",
          "version": "10.2.1"
        },
        "namespace": "builtin",
        "parameters": {
          "stackstart": "0",…
        "dependencies": [
            "name": "callpath",
            "build_hash":
"ayrvk72jdt5d4wznd5wubsqzvl5ffb5p",
            "type": ["build", "link"]
          },
          {
            "name": ”mpich",
…
```

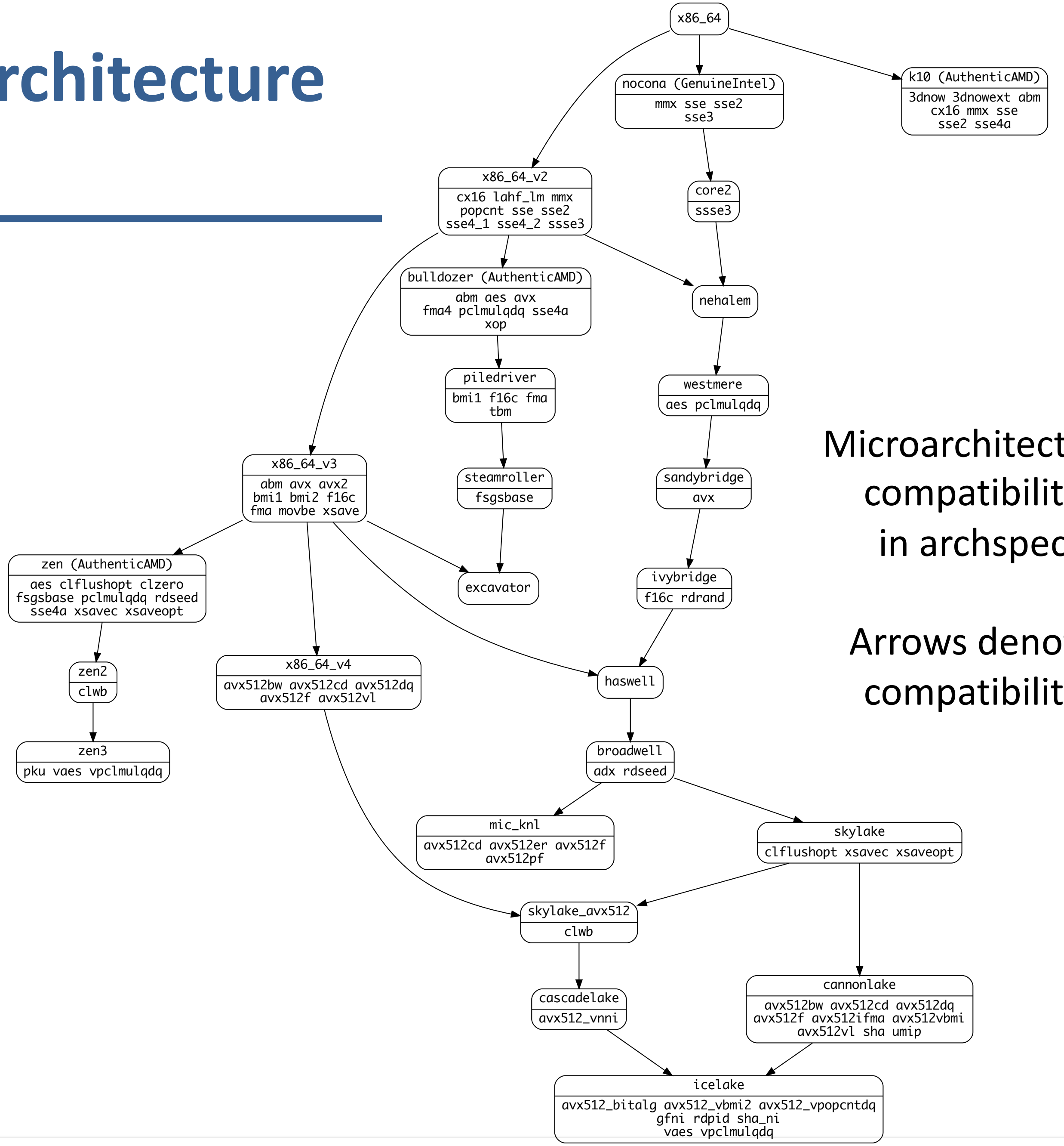
Abstract, normalized spec with some dependencies.

Concrete spec is fully constrained and can be passed to install.

Detailed provenance is stored with the installed package

Spack can reason about microarchitecture and OS compatibility

- Uses archspec to reason about target compatibility.
- Specs also include information about the build OS
  - mostly a proxy for libc, given that we build *most* dependencies in Spack.
  - if we model libc in Spack we can likely start omitting this.
- Spack's solver currently uses this to ensure that we don't *build* a dependency that's incompatible with a dependent.
  - e.g., we currently don't allow a haswell build to depend on an icelake build



Microarchitecture compatibility in archspec

Arrows denote compatibility

How do we reason about *software* compatibility?

We want the package manager to know statically whether two packages will work together.

Two sources of information seem practical:

1. **Match symbol and type information (binary analysis)**
   - Keep (some) symbol information around, and make the solver aware
   - Solver searches for configuration with guaranteed-compatible symbols
   - Not all compatibility is in the symbols (particularly package semantics)

← Focus of another collaboration with U. Wisconsin

2. **Get the package maintainers to tell us (with a DSL)**
   - We already record a lot of provenance in the Spack package
   - What else do we need to express the ABI surface over time?
   - Compatibility could be conditional on version, features/variants, flags, usage of package, etc.
   - How do we design a DSL for this that people will bother writing?

← This work

We frequently want to swap in a new MPI in HPC

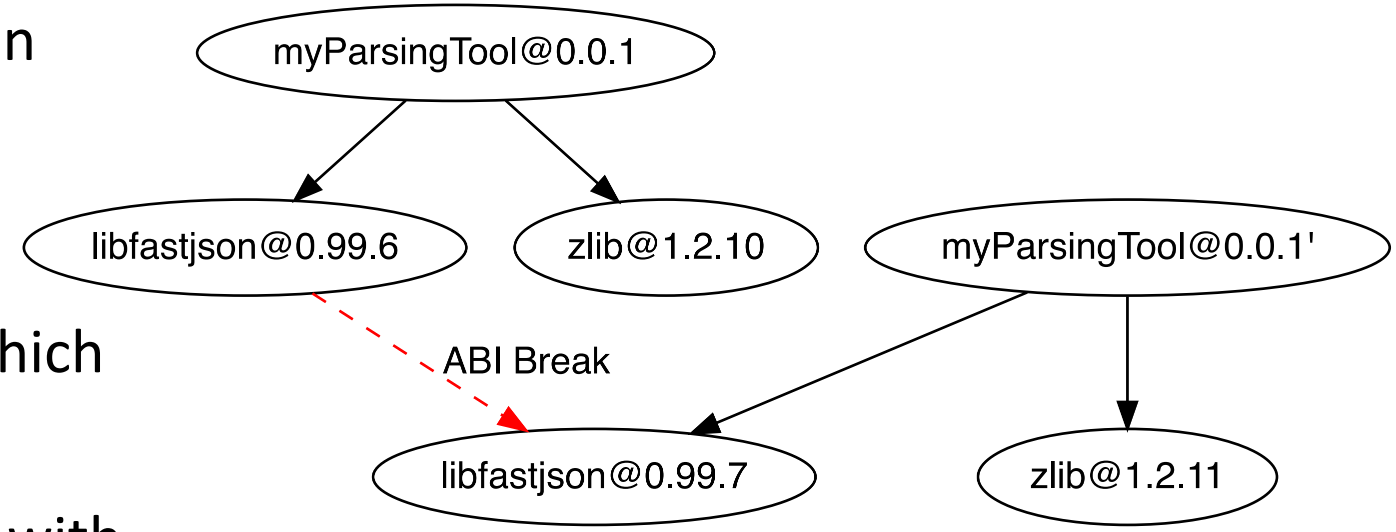
- Running against a new MPI
  - OpenMPI package maintainers tell Spack that OpenMPI 4.0.7 is ABI-compatible with OpenMPI 4.1.2
  - OpenMPI 4.1.2 satisfies all symbols present in the 4.0.7 version.
  - Therefore, users will know that software built against OpenMPI 4.0.7 will run against OpenMPI 4.1.2, regardless of the symbols used.

- Running in a container
  - User built their application with MPICH in a container
  - needs to run with MVAPICH2 from the host for performance
  - bind-mount host MPI into the container

- Spack deployment?
  - We have an HDF5 binary built with MVAPICH2 2.3.1
  - Can we deploy it against MVAPICH2 2.2.0 from the host system?

The External Dependency Problem

- Suppose a Spack package depends on some underlying piece of system software
  - (Called “externals” in Spack parlance)
- Then a system update is required, which includes updating this dependency.
- If the new version is ABI-compatible with the existing version, how do we tell this explicitly to Spack so we don’t have a “rebuild the world” situation?
- What if the dependency *should* trigger rebuilds?



We are working on better External Dependency Representation

- This will be continuously informed by our binary analysis work
- In many cases, the user just needs to manually tell Spack where to go looking for an external library dependency, etc.
- We need a reliable, automatic way to keep up with OS updates.
    - Eventually, our ABISpec filtering algorithm will also be able to determine if we care about ABI surface changes.

We are working on compilers as dependencies

- Compiler objects currently support C, C++, Fortran, and Fortran 77 as distinct languages.
- Models compilers using specs, adding attributes for targets, modules, aliases, extra RPATHs, and more.
- Most importantly, default to settings that make binary relocation possible.
- Work is ongoing to make compilers into proper Dependencies using the Spack model.

We need three things to make binary swapping possible in Spack

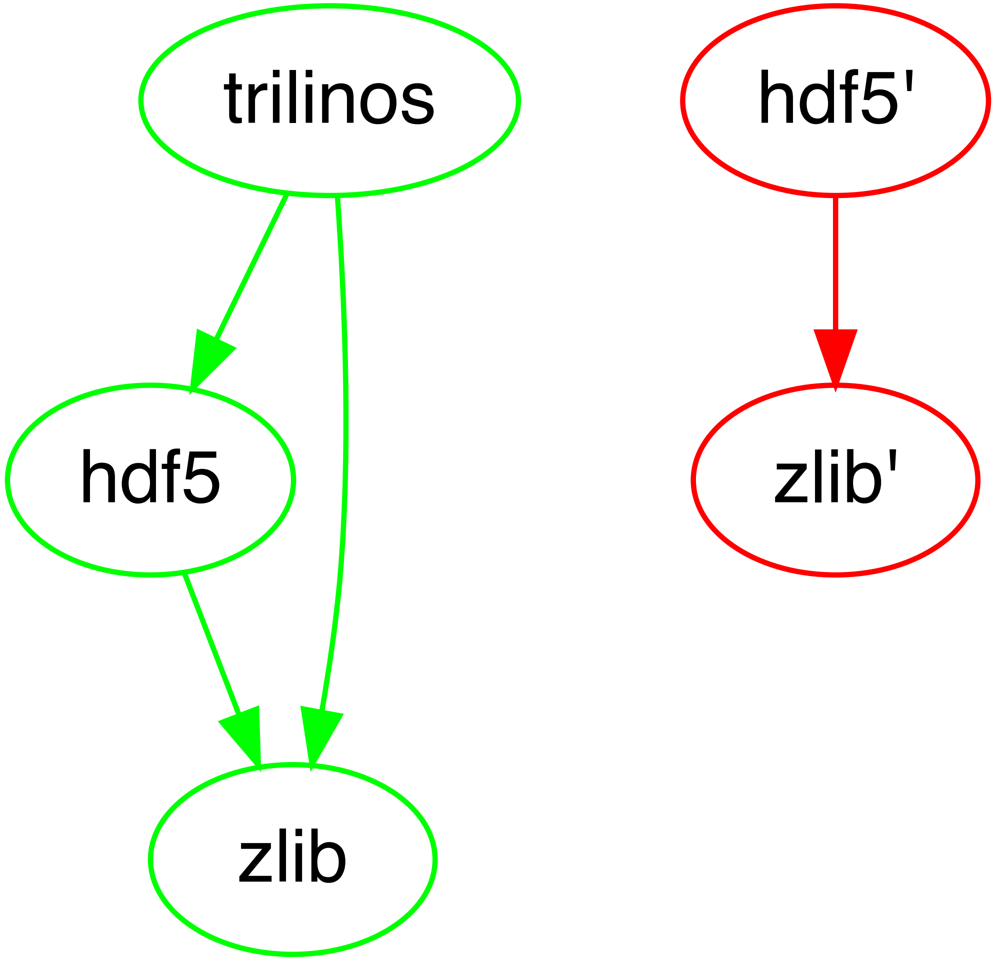
1. ***New deployment and metadata model***
   - Splicing
     - Need to be able to swap one dependency for another
     - Need to avoid losing provenance and *preserve build metadata* even when *deployment is different*
   - Rewiring
     - Need to be able to relocate package RPATH's, shebangs, etc. to point to new dependency
     - Use patchelf, binary rewriting, rewriting symlinks, etc. on installation as part of relocation

2. ***New ABI information in packages***
   - Specified with DSL by user
   - Tells you *what swaps are safe*

3. ***Solver changes***
   - Solver needs to know about ABI constraints
   - Find safe configurations

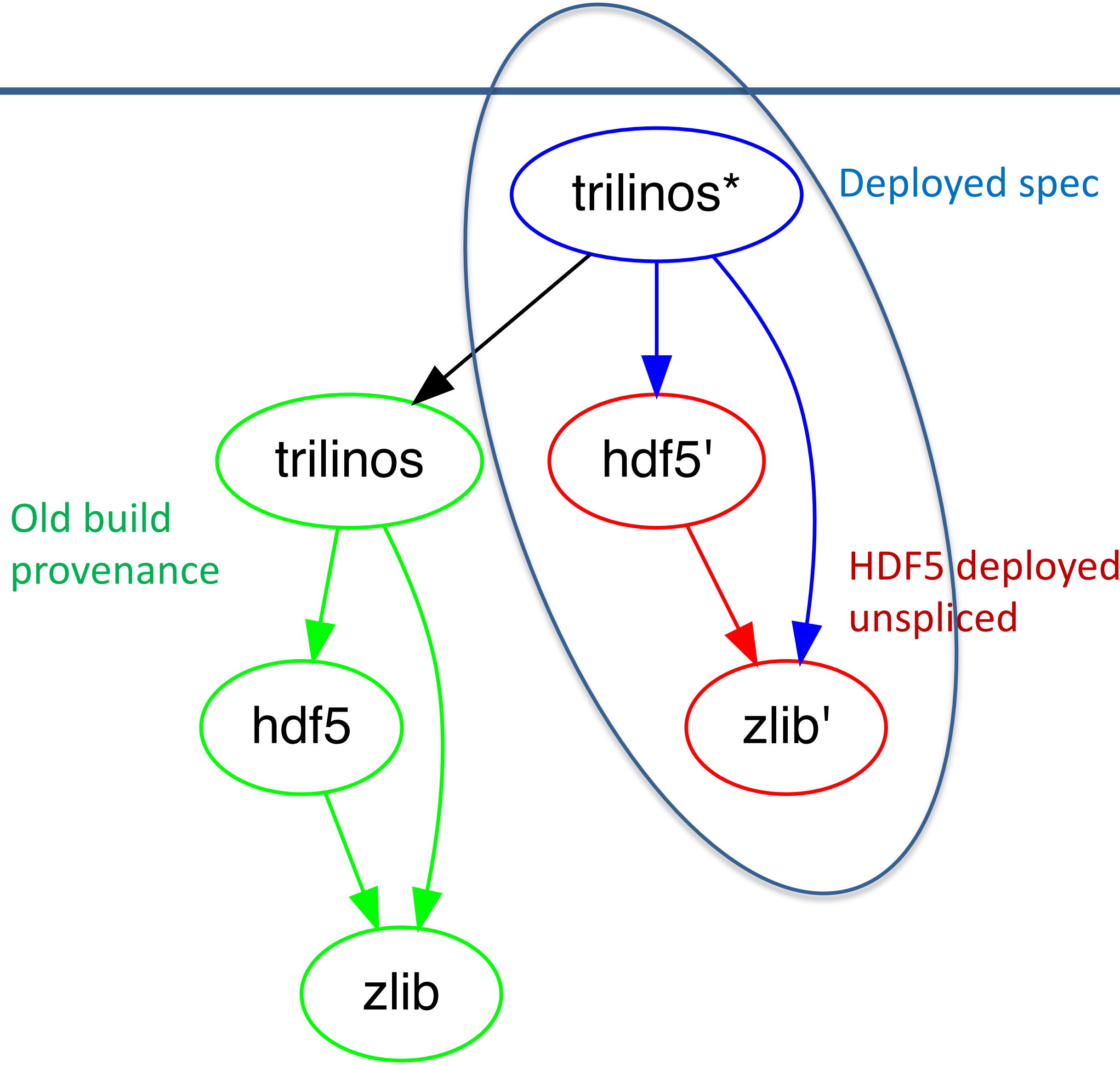
Splicing: a new deployment model for Spack

- A binary of trilinos has already been built and will be deployed on a system with its own HDF5 installation (in green).
- We need to use this system-installed HDF5 (in red).
- We we don't want to totally rebuild trilinos.
- So the system-installed HDF5 is spliced into the DAG



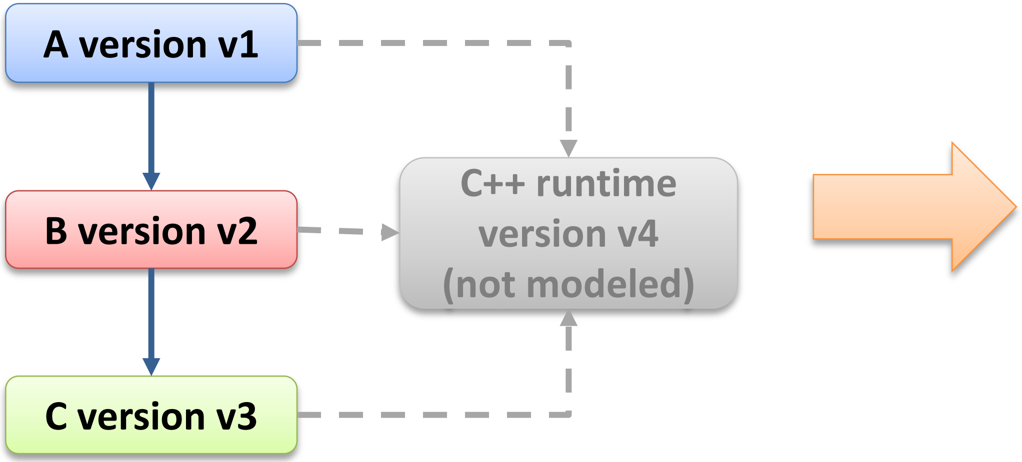
Splicing HDF5

- Trilinos* installation uses the the system-installed **HDF5**.
  - Different HDF5 than it was built with
  - RPATHs from trilinos install now point at the new HDF5
- Black arrow is a “build_spec”
  - Metadata recording original build graph
  - Records original build information
  - Can be used to check ABI compatibility later
- Trilinos now *also* uses the system-installed zlib’ that HDF5 depended on
  - We can also do “intransitive splices”
  - Would use zlib from original trilinos graph
  - Not shown here.

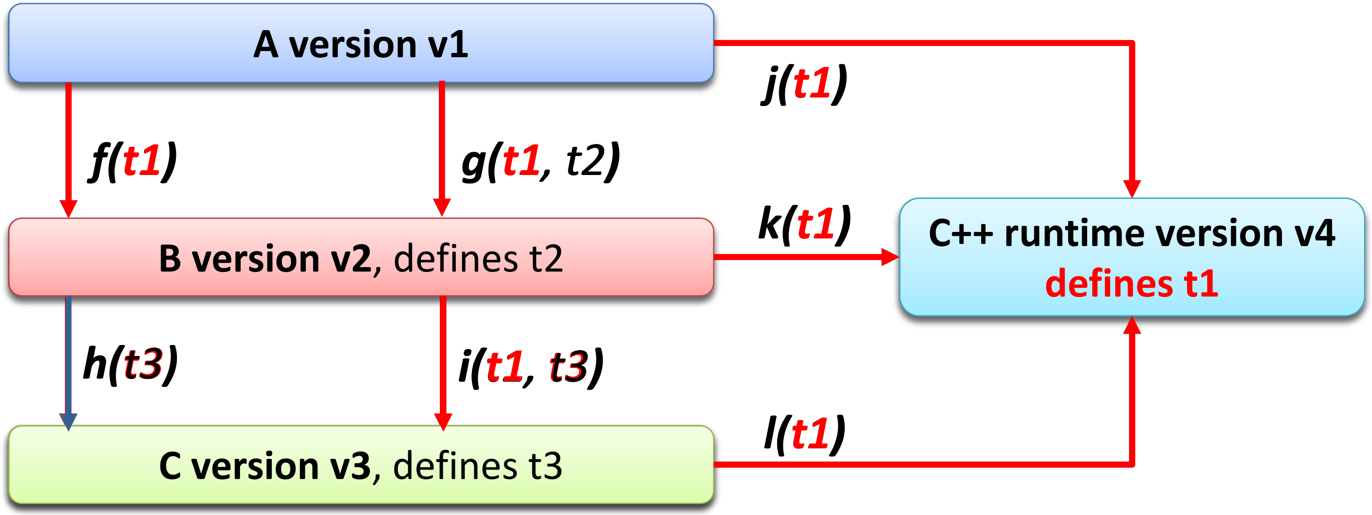


The end goal: Build fine-grained compatibility models that cover functions, data types, and other aspects of ABI

Current model is coarse



Complete model represents *how* changes affect code



- We will model libraries at call granularity:
  - Entry calls
  - Exit calls
  - Data type definitions & usage
- We will model runtime libraries behind compilers
  - C++, OpenMP, glibc
  - GPU runtimes
- We will model changes in the graph
  - "If h(t3) changes, is B still correct?
  - "If C changes, what needs to be rebuilt?"
  - We will model semantics of interfaces

This model allows us to reason about compatibility, so we can find usable packages

Our Proposed Solution in Spack: The ABISpec

- Will encapsulate relevant ABI information about a set of otherwise compatible specs.
- At first, it will just contain all the provenance of a spec with maybe just the build dependencies removed.
- However, over time, this will become more lenient in some ways, and stricter in others.

```
class ABIspec(object):
    …
    @staticmethod
    def _return_abi(os_tag, target_tag, compiler_tag, abi_version):
        platform = spack.platforms.host()
        …
        abi_tuple = …
        return ABIspec(abi_tuple)
```

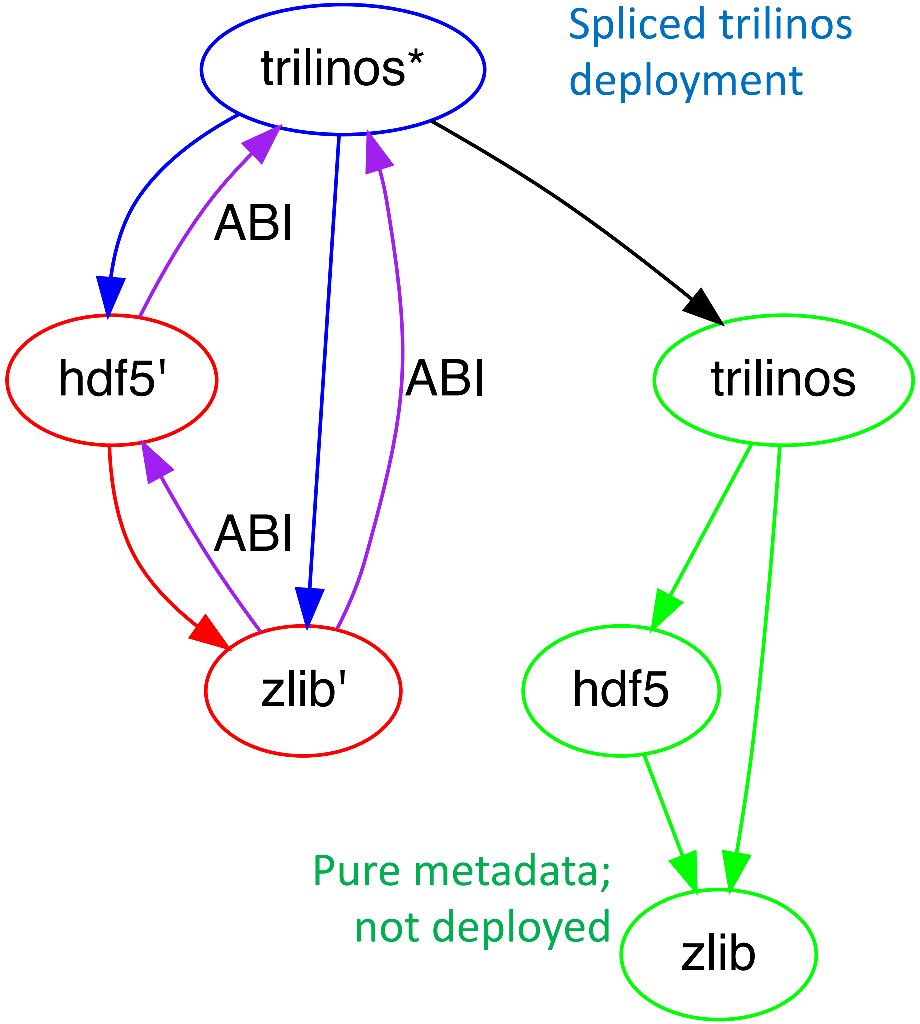
We will also present a clean API for package maintainers in package.py!

Checking ABI in a spliced graph

- ABI Specs will allow us to check whether nodes in a spliced configuration are compatible

- For each *deployed* edge A → B:
  - Check whether abispec(B) satisfies abispec(A)[B]
    - Includes DSL information from packages:
      - Version constraints
      - Enabled sub-APIs
      - Compiler flags
      - etc.
    - Can also (optionally) include binary analysis information
      - Function and symbol comparisons straight from the binary

- Future work will integrate constraints into the solver as facts and rules
  - Search for correct configurations, given a set of binaries

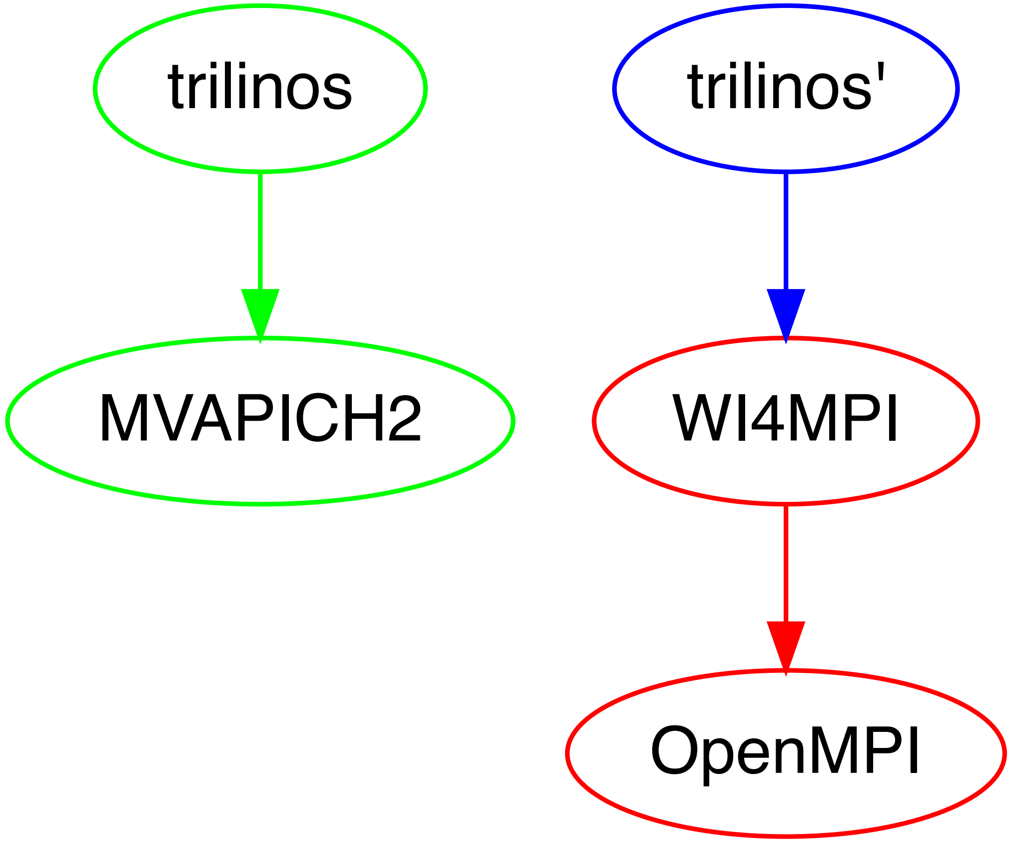


Spliced trilinos deployment

Pure metadata; not deployed

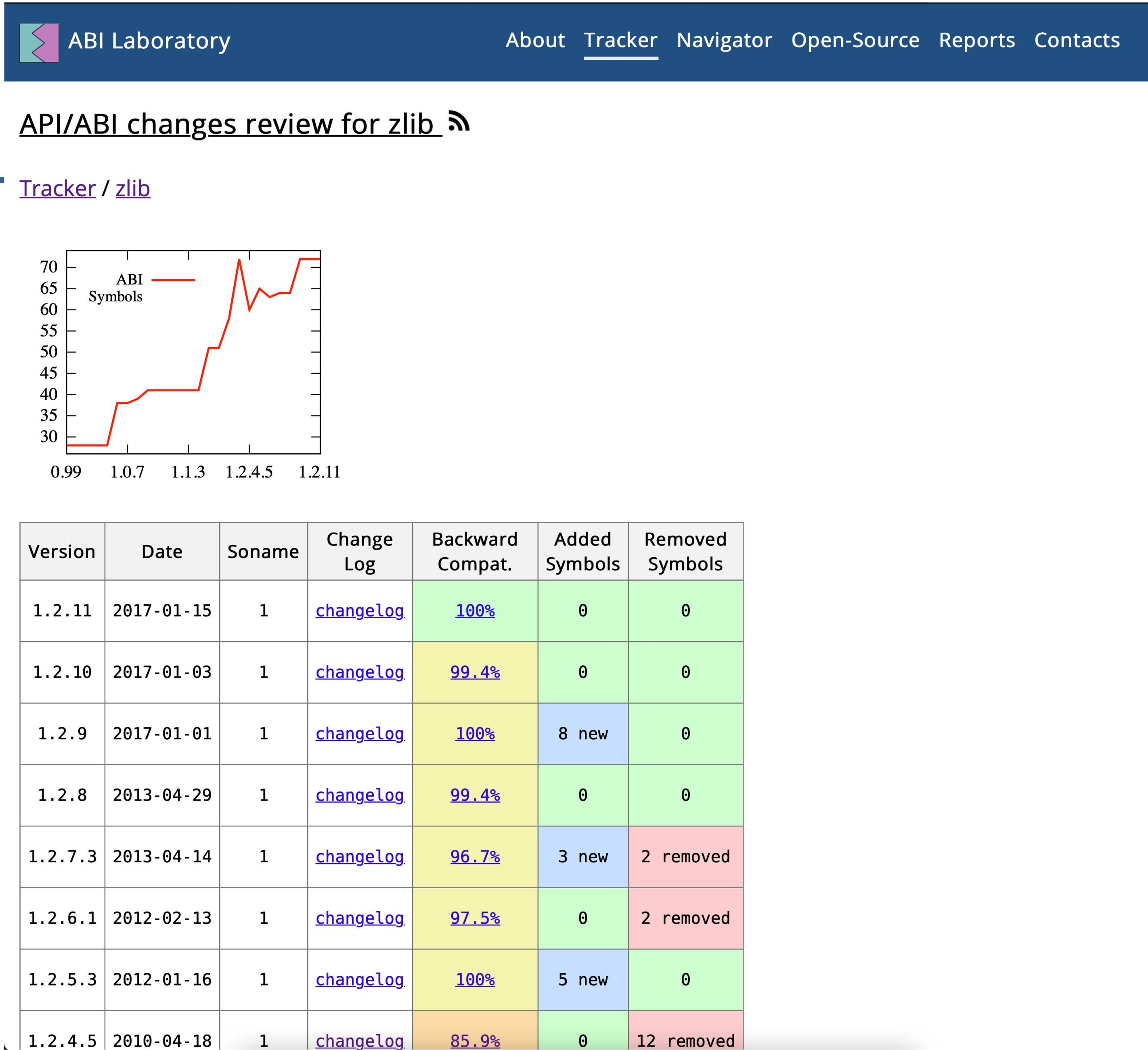
ABI Translation Shims

- WI4MPI and MPItrampoline leverage the fact that MPI implementations adhere to the MPI Standard API in order to translate between ABI-incompatible implementations.
- With WI4MPI, you can build using MPICH, and then run using OpenMPI or vice-versa.
- With either, you can also build against the "fake" MPI library and then run with any MPI library (pictured at right).
- How can we represent this in Spack?



zlib ABI stability

- Even a seemingly stable package can go through many subtle ABI changes
- (as seen on https://abi-laboratory.pro)



API/ABI changes review for zlib

Tracker / zlib

Version	Date	Soname	Change Log	Backward Compat.	Added Symbols	Removed Symbols
1.2.11	2017-01-15	1	changelog	100%	0	0
1.2.10	2017-01-03	1	changelog	99.4%	0	0
1.2.9	2017-01-01	1	changelog	100%	8 new	0
1.2.8	2013-04-29	1	changelog	99.4%	0	0
1.2.7.3	2013-04-14	1	changelog	96.7%	3 new	2 removed
1.2.6.1	2012-02-13	1	changelog	97.5%	0	2 removed
1.2.5.3	2012-01-16	1	changelog	100%	5 new	0
1.2.4.5	2010-04-18	1	changelog	85.9%	0	12 removed

Future work

- Integrate ABI specs and constraints into solver
  - Search for correct configurations
  - How many constraints and how much ABI info can we cram in a solver?
- How to avoid combinatorial explosion?
  - Allowing swaps makes the deployment space much larger (combinatorially)
  - Can we get away with preferring swaps close to the build confguration?
  - How do we prefer one binary over another if metadata is arbitrary?
  - What curation will still be necessary?
- When should you rebuild instead of reusing?
  - How do you quantify this decision?

Lawrence Livermore
National Laboratory