ACCESS-NRI

Australia's Climate Simulator

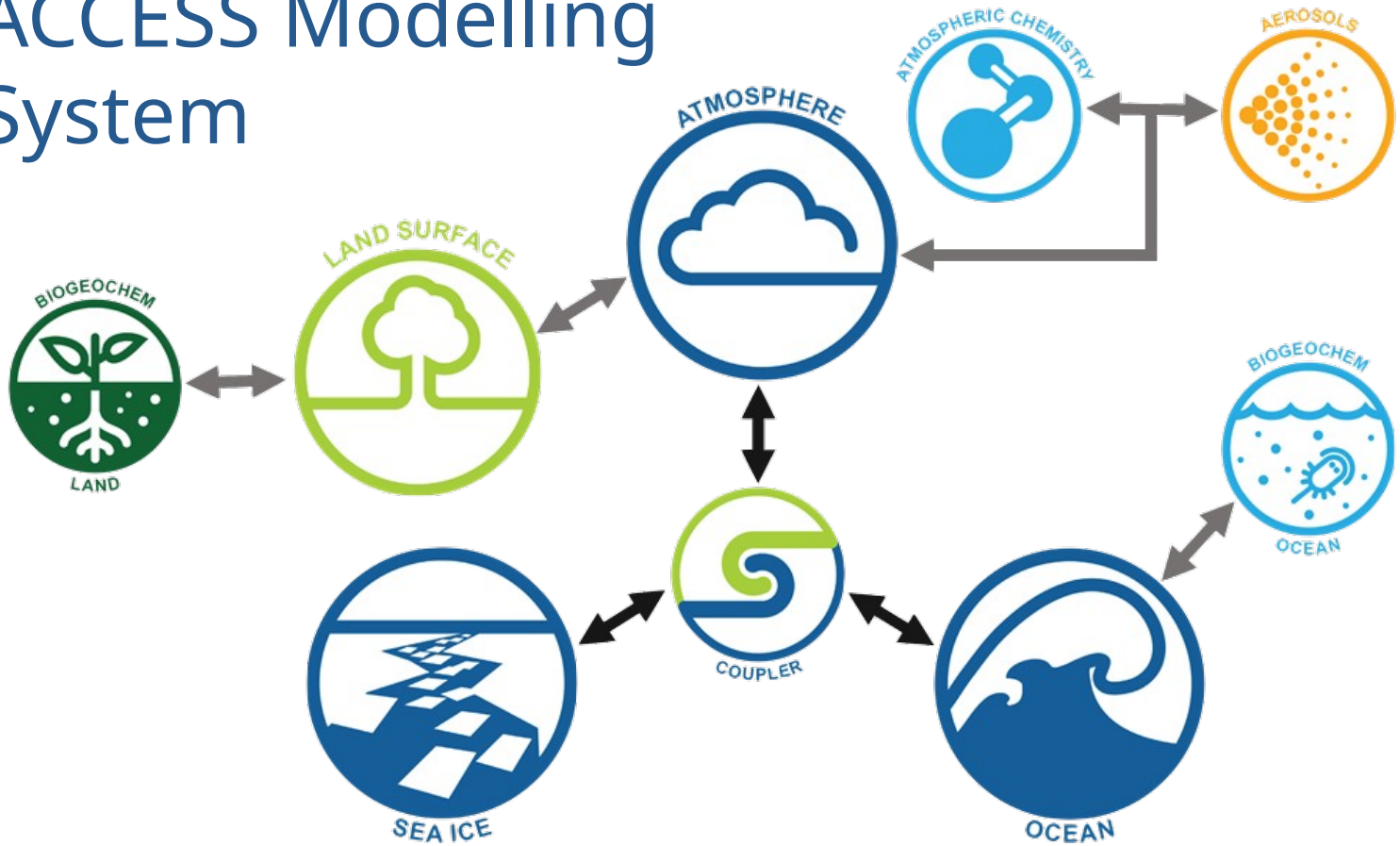
Powered by Spack

Aidan Heerdegen, Harshula Jayasuriya
Tommy Gatti, Paul Leopardi (ACCESS-NRI)

Australia's climate simulator

ACCESS
National Research Infrastructure

ACCESS Modelling System



1

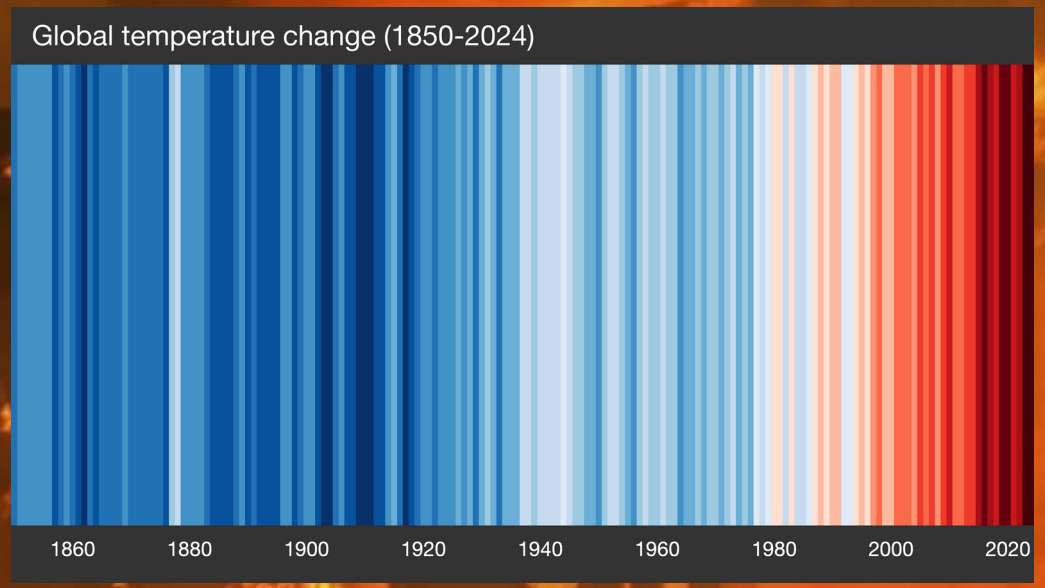
Components

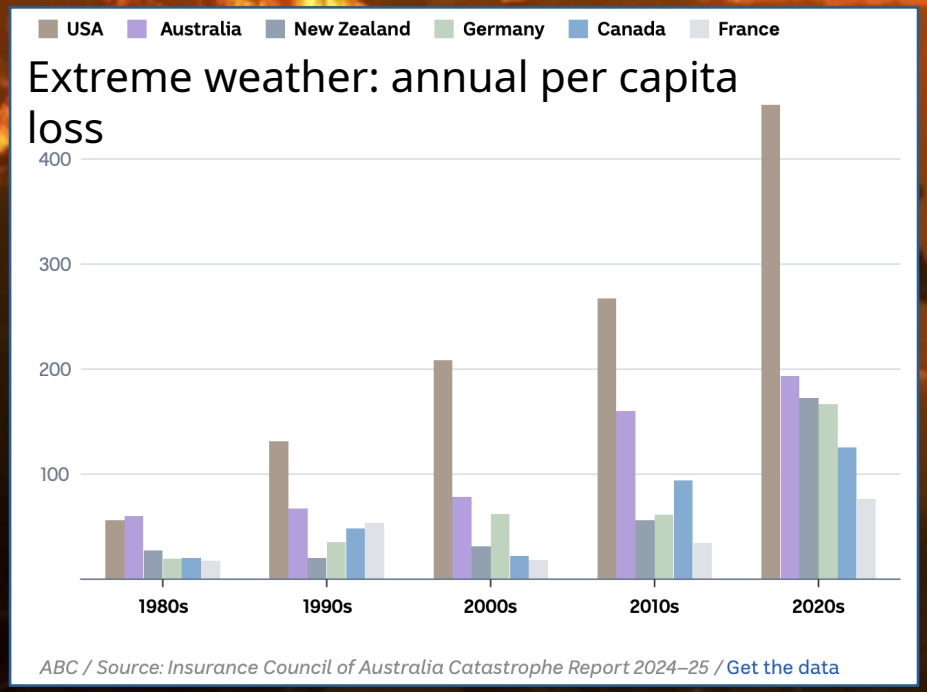
7

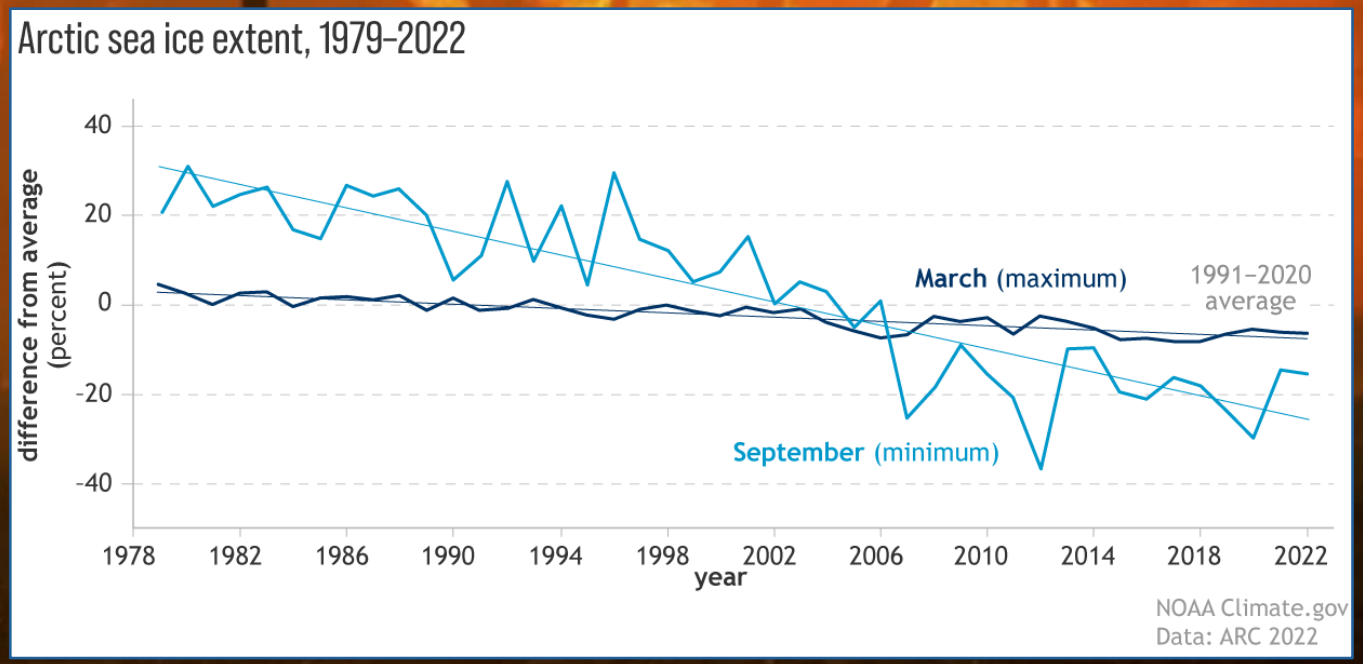
Models

Model: one or more model components & coupler

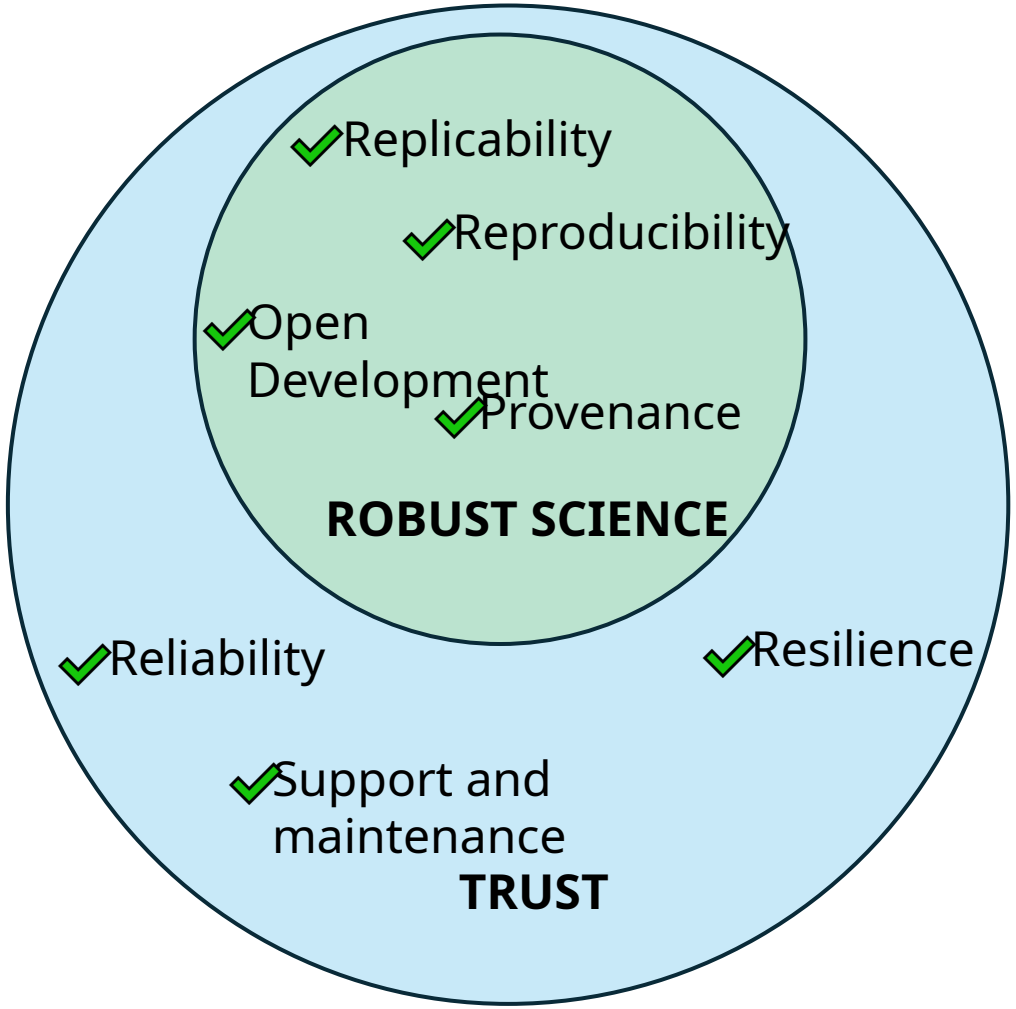
Model component: separate code base & build


Global temperature change (1850-2024)
1860
1880
1900
1920
1940
1960
1980
2000
2020


USA
Australia
New Zealand
Germany
Canada
France
Extreme weather: annual per capita loss
400
300
200
100
1980s
1990s
2000s
2010s
2020s
ABC / Source: Insurance Council of Australia Catastrophe Report 2024–25 / Get the data


Arctic sea ice extent, 1979–2022
difference from average (percent)
40
20
0
-20
-40
March (maximum)
1991–2020 average
September (minimum)
1978
1982
1986
1990
1994
1998
2002
2006
2010
2014
2018
2022
year
NOAA Climate.gov
Data: ARC 2022

Constellation of Concerns

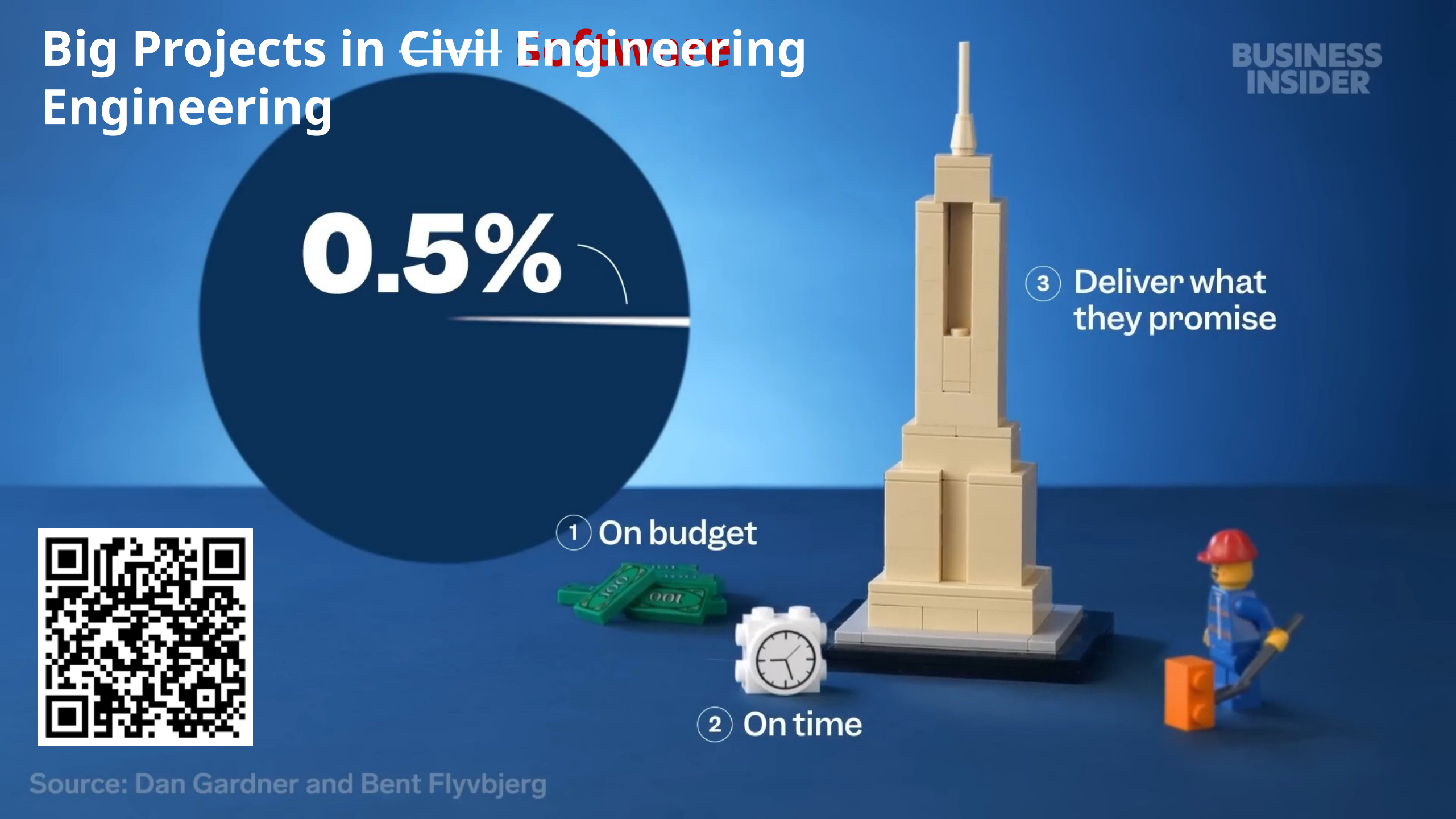


Spack

- Extensive package library (6000+)
- Full build provenance
- Build reproducibility
- Cross platform

Big Projects in ~~Civil Engineering~~ Engineering

BUSINESS INSIDER



Source: Dan Gardner and Bent Flyvbjerg

Spack package

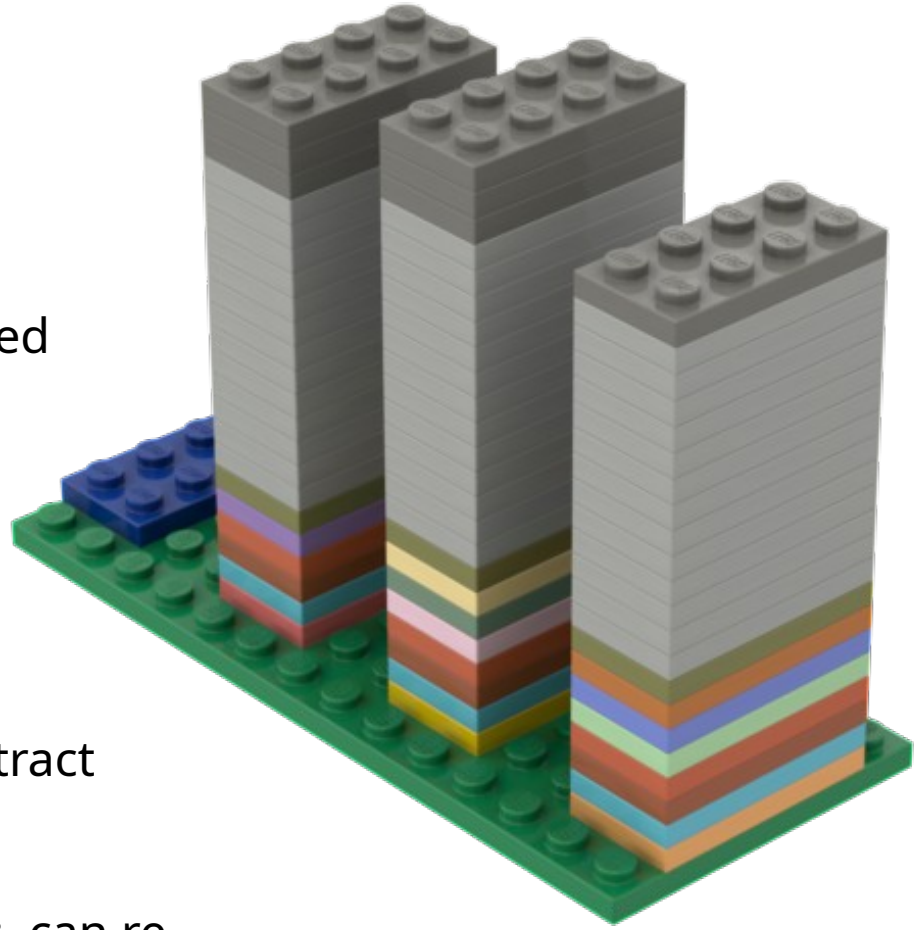
- Spack package is our foundation ”lego” for modular build
- Written in python
- Can build itself & knows its dependencies
- Supports common build systems

```
class MppnccombineFast(CMakePackage):
    """mppnccombine-fast is a fast version of mppnccombine, a
       tool for combining multiple netCDF files into a single
       file. It is designed to be used with large datasets and
       provides significant performance improvements over the
       standard mppnccombine."""

    homepage = "https://github.com/coecms/mppnccombine-fast"
    git = "https://github.com/ACCESS-NRI/mppnccombine-fast.git"
    url = "https://github.com/ACCESS-NRI/mppnccombine-fast/archive/refs/tags/2025.07.000.tar.gz"
    maintainers("dougiesquire")

    license("Apache-2.0", checked_by="dougiesquire")

    version("2025.07.000", sha256="d74ef9b47aa6a6aac2d2f802f146b59d585104a780b65571fe7fda78e69af553")

    depends_on("cmake@3.10:", type="build")
    depends_on("mpi")
    depends_on("hdf5")
    depends_on("netcdf-c")
```

Spack environment

- Environment defined by manifest file
- YAML format
- Modular: utilises spack packages
- Second piece of ”lego”

- Abstract dependencies: specify only what is required



- Concretization: determine dependencies that satisfy abstract dependencies
- Build everything
- Lockfile provides provenance: can re-

```
spack:
  specs:
    - access-esm1p6@2025.09.002
  packages:
    mom5:
      require:
        - '@2025.05.000'
    cice5:
      require:
        - '@2025.09.000'
    um7:
      require:
        - '@2025.10.000’
    openmpi:
      require:
        - '@5.0.8’
    oasis3-mct:
      require:
        - '@2025.03.001’
    gcom4:
      require:
        - '@2025.08.000’
    access-fms:
      require:
        - '@2025.08.000'
    access-generic-tracers:
      require:
        - '@2025.09.000'
    access-mocsy:
      require:
        - '@2025.07.002’
    netcdf-c:
      require:
        - '@4.9.2'
    netcdf-fortran:
      require:
        - '@4.6.1’
     hdf5:
      require:
        - '@1.14.3'
    # Preferences for all packages
    all:
      require:
        - '%oneapi@2025.2.0'
        - 'target=x86_64_v4'
```

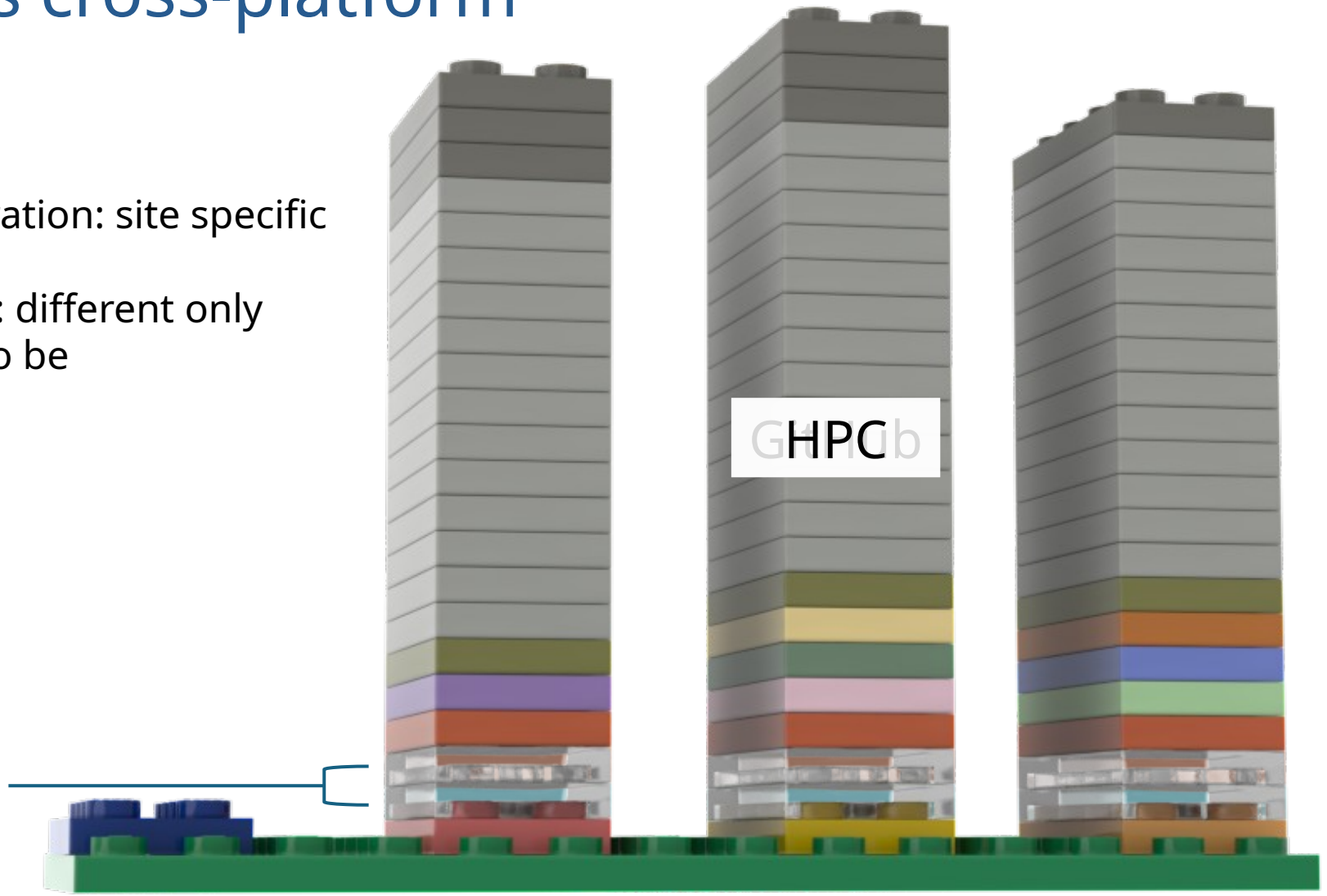
Spack is cross-platform

Recipe: same

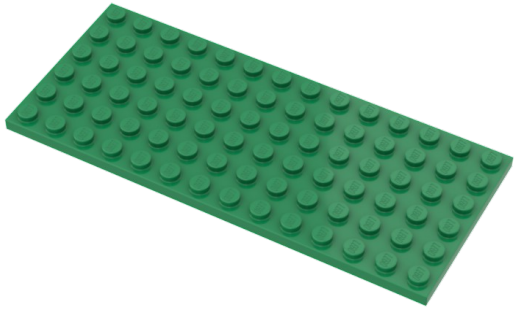
Spack configuration: site specific

Concretization: different only where needs to be

Virtual (external) dependencies



Generic Workflows: build-cd



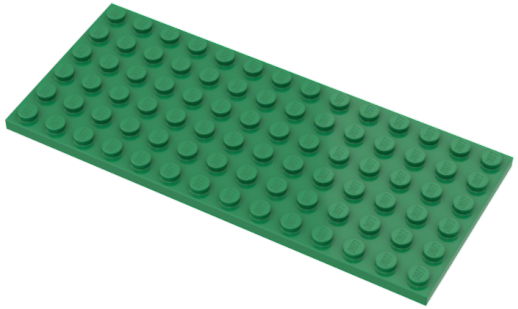
- Deploy models from repo that is just an environment file
- No code
- Simple to add pull request (PR) deployment workflow:

```yaml
jobs:
  cd:
    name: CD
    uses: access-nri/build-cd/.github/workflows/cd.yml@v5
    with:
      model: ${{ vars.NAME }}
      root-sbd: access-esm1p6
    permissions:
      contents: write
      pull-requests: write
    secrets: inherit
```

Workflow features:

- Individual isolated pre-release builds for every push to PR (on-prem HPC)
- Every build is accessible easily through `module load`
- On PR merge: auto-tag, deployment to release (on-prem HPC), upload provenance to DB
- Generic: re-used for general purpose software deployment

Generic Workflows: build-ci



- Model component code repositories: check code changes compile
- Add build correctness testing for model components:

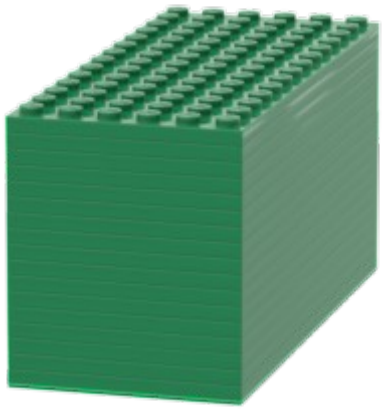
```
jobs:
  ci:
    name: CI
    uses: access-nri/build-ci/.github/workflows/ci.yml@v2
    with:
      spack-manifest-path: .github/build-ci/spack.yaml
      spack-manifest-data-path: .github/build-ci/data/standard.json
      allow-ssh-into-spack-install: false
    secrets:
      spack-install-command-pat: ${{ secrets.SPACK_INSTALL_CMD_PATH }}
```

- Using self-hosted runners (NECTAR Kubernetes), can use generic GitHub runners
- Build caching ensures good performance
- Option to SSH into runner container for debugging
- Generic: reused for CI in ACCESS-NRI spack-packages repository

ARDC Nectar Research Cloud

Scaling: model deployment with build-cd

Released models available to community on gadi@NCI



2025

Pre-release builds: testing features and release candidates

2025 Q3

Summary

- Spack is an API for building software
- Spack provides full build provenance and build reproducibility
- Modular design enables generic tools for testing and deployment and efficient code re-use
- Spack and modularity enables scale: zero marginal costs of builds, cross platform

Credits

- Tommy Gatti (ACCESS-NRI): CI/CD build infrastructure
- Harshula Jayasuriya (ACCESS-NRI): spack infrastructure
- Paul Leopardi: spack package recipes
- Pictures:
  - Lake Canjola 2020 Matthew Abbott / New York Times / Redux / eyevine
  - https://showyourstripes.info/ CC-BY-4.0
  - Coral bleaching: Brett Monroe Garner/Greenpeace
  - Sea Ice: NOAA Climate.gov ARC 2022
  - Annual per capita economic loss ABC Source: Insurance Council of Australia Catastrophe Report 2024–25

Australia's climate simulator

Documentation: https://spack.readthedocs.io/en/latest/

Video and textual tutorials: https://spack-tutorial.readthedocs.io/en/latest/

Communication with Spack core-developers and community via Slack: https://slack.spack.io/

e.g. A very active #help channel

Weekly 1 hour video chats for real-time discussion and trouble-shooting

One of the monthly meeting times was moved to AU business hours to facilitate attendees from Asia-Pacific region. https://forum.access-hive.org.au/t/building-a-spack-community-in-australia/3833

Meeting minutes containing Q&A. e.g. https://github.com/spack/spack/wiki/Telcon:-2025-10-07

First Spack User Meeting held at HPSF Conference 2025

https://spack.io/spack-user-meeting-2025/

https://www.youtube.com/watch?v=mjcjK0JLPpo&list=PLRKq_yxxHw29-JcpG2CZ-xKK2U8Hw8O1t

Code engagement

- New Spack Package recipe accepted within 3 business days: https://github.com/spack/spack-packages/pull/1840
- Issues about Spack-core raised and quickly fixed. e.g. https://github.com/spack/spack/issues/51167

Manging PE Updates @ Pawsey

Dr Emily Kahl, Dr. Deva Kumar Deeptimahanti
Pawsey Supercomputing Research Centre
eResearch Australasia 2025

OFFICIA
L

Pawsey Supercomputing Research Centre

- Headquarters located in Perth, Western Australia
- Offers critical support to radioastronomy research around the Square Kilometre Array (SKA).
- Support ~4000 users from a large number of different science domains with a wide variety of workflows

Pawsey

The problem

- Supercomputing software stacks are becoming increasingly complex:
  - Increasing number of applications and libraries from a variety of fields of science.
  - Multiple CPU architectures, compilers and supporting libraries.
  - Multiple configurations for the same software based on user requirements.
  - Constant evolution of the software provided by HPE Cray.
- Need automation of the deployment process to improve deployment times, reduce human errors and *reduce the workload on staff*

Pawsey's Approach

- Pawsey has automated the deployment process to improve deployment times, reduce human errors and reduce the workload on staff.
  - Built around Spack and Singularity HPC (SHPC).
  - Integrates with the HPE Cray PE to provide multiple builds.
  - Uses the Lmod module system to expose software deployed with Spack, SHPC and custom build methods.
- The automation package can be downloaded from

github.com/PawseySC/pawsey-spack-config.git

Design

- Software stack components and organization
- Software stack has three levels of deployment (and associated support):
  - System-wide: accessible by all and maintained by Pawsey.
  - Project-wide: maintained by the project and accessible by project members.
  - User-private: individual Pawsey users installations.
- It consists of:
  - Optimized bare-metal installations accessed through modules.
  - Containers accessed through modules.
  - The module files providing software are organized in informative, user-friendly categories (e.g., applications, libraries, containers, etc.).
- Supported versions follow a general 3 +1 rule (legacy, stable, latest)

Design

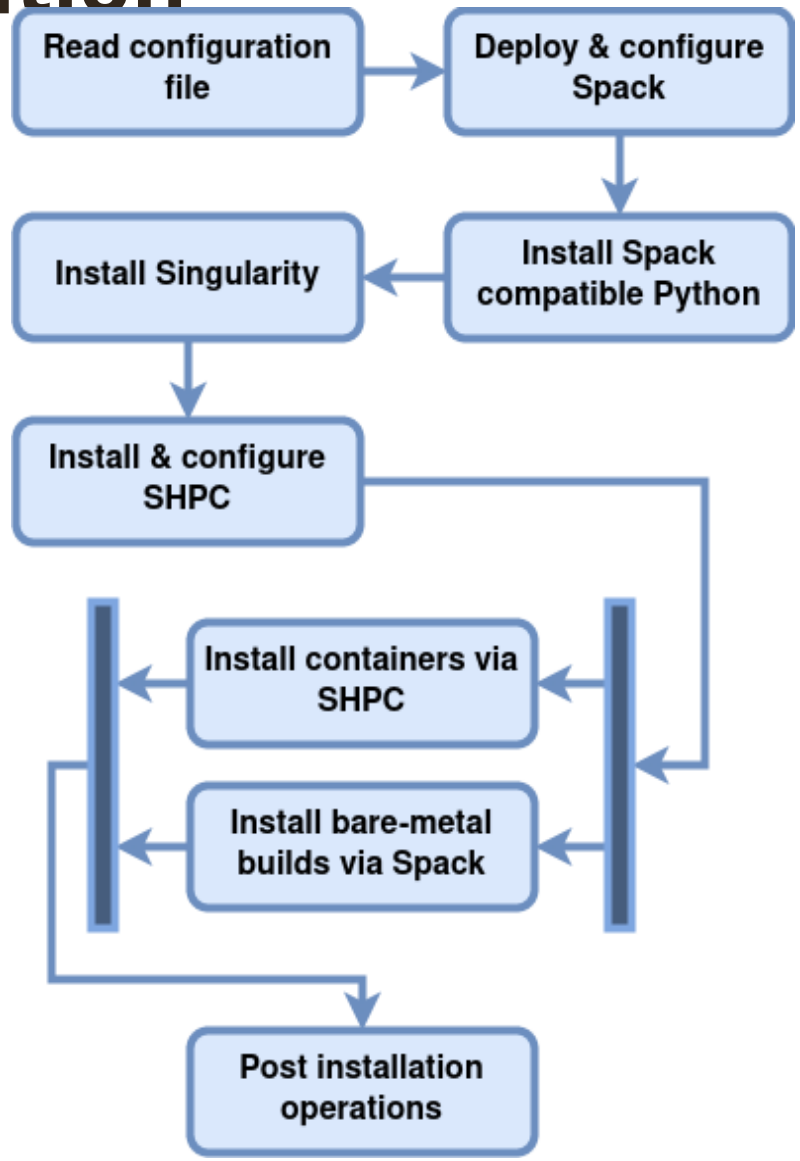
- Cray’s customized Lmod installation implements an undocumented custom way of handling hierarchies in substitution for the standard Lmod way.
  - Compiler, CPU architecture and MPI library modules scan the shell environment for a specific variable that sets additional module paths.
  - Paths are (un)set when the relevant module is (un)loaded (unloaded). For example, modules for a GCC compiler will look for LMOD_CUSTOM_COMPILER_GNU_8_0_PREFIX.
- We implement this process for Spack-built software packages through a auto-loaded module, pawseyenv.lua

System-specific configuration

```
.
├── configs
│   ├── project
│   │   ├── config.yaml
│   │   ├── modules.yaml
│   │   └── packages.yaml
│   ├── site
│   │   ├── compilers.yaml
│   │   ├── concretizer.yaml
│   │   ├── config.yaml
│   │   ├── modules.yaml
│   │   ├── packages.yaml
│   │   └── repos.yaml
│   └── spackuser
│       ├── config.yaml
│       └── modules.yaml
├── environments
│   ├── env_apps
│   │   └── spack.yaml
│   ├── env_num_libs
│   │   └── spack.yaml
│   :
├── settings.sh
└── templates
    └── modules
        └── modulefile.lua
```

Setonix

Automated Deployment Execution



Spack:
The road to 1.0

eResearch Australasia
October, 2025

THE LINUX FOUNDATION

Spack provides a spec syntax to describe build options

```
$ spack install mpileaks                      unconstrained
$ spack install mpileaks@3.3                  @ custom version
$ spack install mpileaks@3.3 %gcc@4.7.3       % custom compiler
$ spack install mpileaks@3.3 %gcc@4.7.3 +threads   +/- build option
$ spack install mpileaks@3.3 cppflags="-O3 –g3"    set compiler flags
$ spack install mpileaks@3.3 target=cascadelake    set target microarchitecture
$ spack install mpileaks@3.3 ^mpich@3.2 %gcc@4.9.3  ^ dependency constraints
```

- Each expression is a ***spec*** for a particular configuration
  - Each clause adds a constraint to the spec
  - Constraints are optional – specify only what you need.
  - Customize install on the command line!
- Spec syntax is recursive
  - Full control over the combinatorial build space

Not shown: patches, resources, conflicts, other directives.

Spack packages are parametrized using spec syntax

```
from spack import *

class Kripke(CMakePackage):
    """Kripke is a simple, scalable, 3D Sn deterministic particle transport mini-app."""

    homepage = "https://computation.llnl.gov/projects/co-design/kripke"
    url      = "https://computation.llnl.gov/projects/co-design/download/kripke-openmp-1.1.tar.gz"

    version('1.2.3', sha256='3f7f2eef0d1ba5825780d626741eb0b3f026a096048d7ec4794d2a7dfbe2b8a6')
    version('1.2.2', sha256='eaf9ddf562416974157b34d00c3a1c880fc5296fce2aa2efa039a86e0976f3a3')
    version('1.1', sha256='232d74072fc7b848fa2adc8a1bc839ae8fb5f96d50224186601f55554a25f64a')

    variant('mpi',    default=True, description='Build with MPI.')
    variant('openmp', default=True, description='Build with OpenMP enabled.')

    depends_on('mpi', when='+mpi')
    depends_on('cmake@3.0:', type='build')

    def cmake_args(self):
        return [
            '-DENABLE_OPENMP=%s' % ('+openmp' in self.spec),
            '-DENABLE_MPI=%s' % ('+mpi' in self.spec),
        ]

    def install(self, spec, prefix):
        mkdirp(prefix.bin)
        install('../spack-build/kripke', prefix.bin)
```

Base package (CMake support)

Metadata at the class level

Versions

Variants (build options)

Dependencies (same spec syntax)

Install logic in instance methods

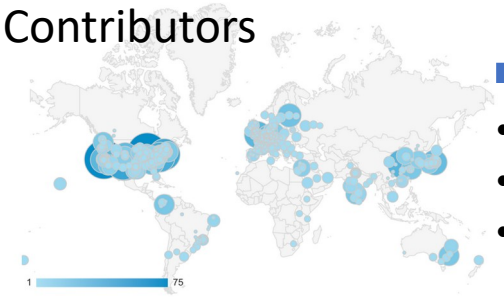
Don't typically need install() for CMakePackage, but we can work around codes that don't have it.

***One* package.py file per software project**

Spack's concretizer fills in the details

This part is NP-hard!

Contributors



- new versions
- new dependencies
- new constraints

package.py repository

concretizer

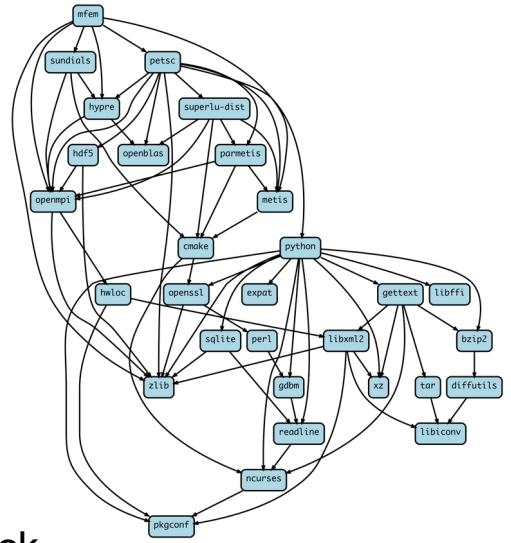
spack developers — default config packages.yaml

admins, users — local preferences config packages.yaml

users — environment config spack.yaml

users — Command line constraints

```
spack install hdf5@1.12.0 +debug
```

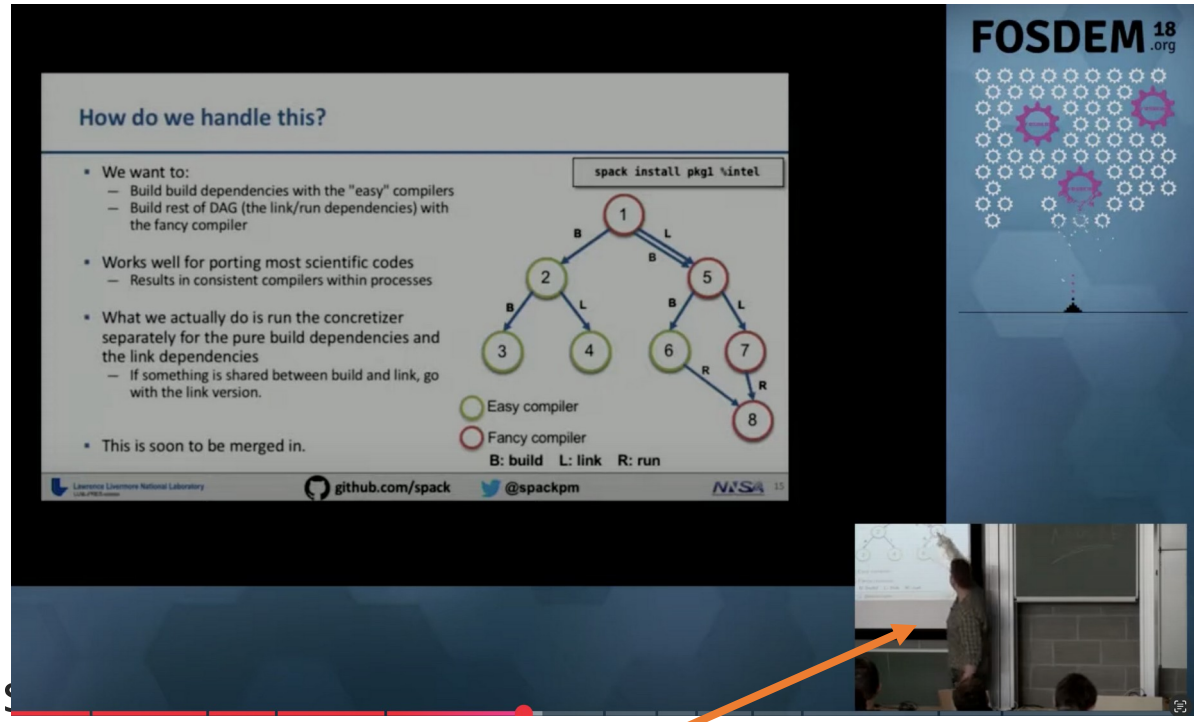


Concrete spec is fully constrained and can be built.

Is stored in spack.lock file after solve.

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
  - ✅ 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 its inception



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

First challenge: Dependency resolution is NP-complete

- Write our own in Python (original implementation)
  - Incomplete, couldn't handle complexity
- SAT: Boolean Satisfiability
  - Hard to model the problem space with just True and False to work with
  - Optimization is very hard to implement on top (and slow)
- SMT: Satisfiability modulo theories
  - Support for integer math, implications, higher level logic operations
  - Support for multi-criteria optimization
  - Traction in formal verification, compiler communities

- ASP: Answer Set Programming
  - Clingo (from the Potassco project), very actively developed, very fast
  - Looks like Prolog
  - Easy to read, declarative modeling language
  - Support for multi-criteria optimization

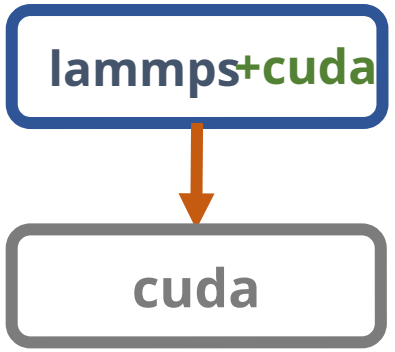
We ended up selecting ASP (from potassco.org)

Spack's concretizer is implemented using Answer Set Programming (ASP)

ASP looks like Prolog but is converted to SAT with optimization

Facts describe the graph

```
node("lammps").
node("cuda").
variant_value("lammps", "cuda", "True").
depends_on("lammps", "cuda").
```



First-order rules (with variables) describe how to resolve nodes and metadata

```
node(Dependency) :- node(Package), depends_on(Package, Dependency).
```

```
node("mpi")
```

```
node("hdf5").
depends_on("hdf5", "mpi").
```

Ground Rule

ASP: integrity constraints and choice rules

Integrity constraints say what ***cannot*** happen

```
path(A, B) :- depends_on(A, B).
path(A, C) :- path(A, B), depends_on(B, C).

% this constraint says "no cycles"
:- path(A, B), path(B, A).
```

ASP: version selection

Choice rules give the solver freedom to choose from possible options:
Inputs from package definition:

```
possible_version("hdf5", "1.13.1").
possible_version("hdf5", "1.12.2").
possible_version("hdf5", "1.12.1").
possible_version("hdf5", "1.12.0").
```

```
% if a package is in the graph, solver must choose exactly one version
% out of that package's possible versions
1 { version(Pkg, V) : possible_version(Pkg, V) } 1 :- node(Pkg).
```

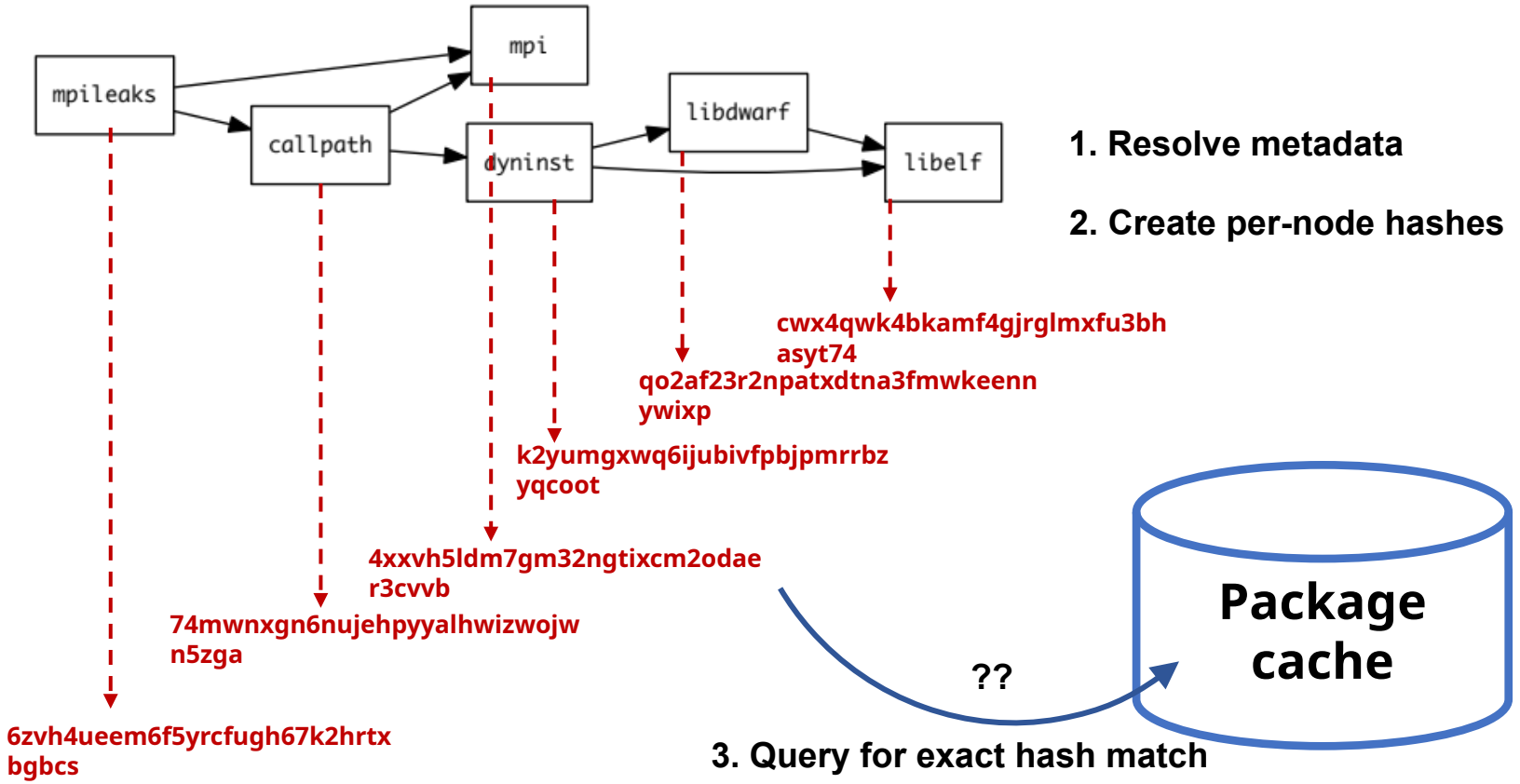
ASP: optimization

We can associate ***weights*** with versions:

```
possible_version("hdf5", "1.13.1”, 0).
possible_version("hdf5", "1.12.2", 1).
possible_version("hdf5", "1.12.1", 2).
possible_version("hdf5", "1.12.0", 3).
```

```
% assign a weight to each selected version
version_weight(Pkg, Weight)
  :- version(Pkg, V), possible_version(Pkg, V, Weight).

% minimize the sum of all version weights
#minimize{ Weight@3,Pkg : version_weight(Pkg, Weight) }.
```

Second challenge: Spack's original concretizer did not reuse existing installations



- Hash matches are very sensitive to small changes
- In many cases, a satisfying cached or already installed spec can be missed
- Nix, Spack, Guix, Conan, and others reuse this way

The new concretizer allows us to be more aggressive about reusing packages.

- First, we need to tell the solver about all the installed packages!
- Add constraints for all installed packages, with their hash as the associated ID:

```
installed_hash("openssl","lwatuuysmwkhuahrncywvn77icdhs6mn").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node","openssl").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","version","openssl","1.1.1g").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_platform_set","openssl","darwin").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_os_set","openssl","catalina").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_target_set","openssl","x86_64").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","variant_set","openssl","systemcerts","True").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_compiler_set","openssl","apple-clang").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","node_compiler_version_set","openssl","apple-clang","12.0.0").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","concrete","openssl").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","depends_on","openssl","zlib","build").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","depends_on","openssl","zlib","link").
imposed_constraint("lwatuuysmwkhuahrncywvn77icdhs6mn","hash","zlib","x2anksgssxsxa7pcnhzg5k3dhgacglze").
```

Telling the solver to minimize builds is surprisingly simple:
leverage the *impose* half of a generalized condition.

1. Allow the solver to *choose* a hash for any package:

```
{ hash(Package, Hash) : installed_hash(Package, Hash) } 1 :- node(Package).
```

2. Choosing a hash means we impose its constraints:

```
impose(Hash) :- hash(Package, Hash).
```

3. Define a build as something *without* a hash:

```
build(Package) :- not hash(Package, _), node(Package).
```

4. Minimize builds!

```
#minimize { 1@100,Package : build(Package) }.
```

With and without reuse optimization

Note the bifurcated optimization criteria

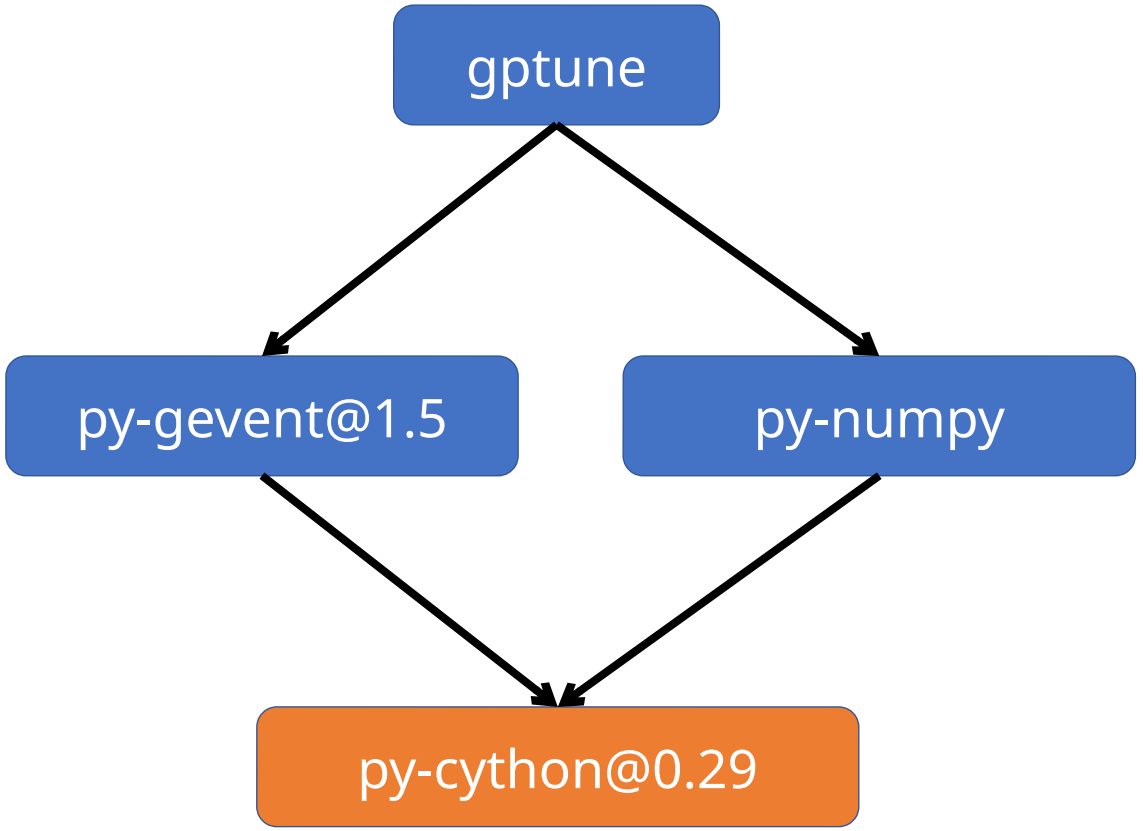
```
(spackle):solver> spack solve -Il hdf5
==> Best of 9 considered solutions.
==> Optimization Criteria:
  Priority  Criterion                                                 Installed  ToBuild
  1         number of packages to build (vs. reuse)                           -       20
  2         deprecated versions used                                          0        0
  3         version weight                                                    0        0
  4         number of non-default variants (roots)                            0        0
  5         preferred providers for roots                                     0        0
  6         default values of variants not being used (roots)                 0        0
  7         number of non-default variants (non-roots)                        0        0
  8         preferred providers (non-roots)                                   0        0
  9         compiler mismatches                                               0        0
  10        OS mismatches                                                     0        0
  11        non-preferred OS's                                                0        0
  12        version badness                                                   0        2
  13        default values of variants not being used (non-roots)             0        0
  14        non-preferred compilers                                           0        0
  15        target mismatches                                                 0        0
  16        non-preferred targets                                             0        0

  -   zzngfs3  hdf5@1.10.7%apple-clang@13.0.0~cxx~fortran~hl~ipo~java+mpi+shared~szip~threadsafe+tools api=default b
  -   nsylovq      ^cmake@3.21.4%apple-clang@13.0.0~doc+ncurses+openssl+ownlibs~qt build_type=Release arch=darwin-bi
  -   xdbaqeo          ^ncurses@6.2%apple-clang@13.0.0~symlinks+termlib abi=none arch=darwin-bigsur-skylake
  -   kfureok              ^pkgconf@1.8.0%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   5ekd4ap          ^openssl@1.1.1l%apple-clang@13.0.0~docs certs=system arch=darwin-bigsur-skylake
  -   xz6a265              ^perl@5.34.0%apple-clang@13.0.0+cpanm+shared+threads arch=darwin-bigsur-skylake
  -   xgt3tls                  ^berkeley-db@18.1.40%apple-clang@13.0.0+cxx~docs+stl patches=b231fcc4d5cff05e5c3a4814
  -   65edjf6                  ^bzip2@1.0.8%apple-clang@13.0.0~debug~pic+shared arch=darwin-bigsur-skylake
  -   662adoo                      ^diffutils@3.8%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   fu7tfsr                          ^libiconv@1.16%apple-clang@13.0.0 libs=shared,static arch=darwin-bigsur-skyla
  -   vjg67nd                  ^gdbm@1.19%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   tjceldr                      ^readline@8.1%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   xevvljj                  ^zlib@1.2.11%apple-clang@13.0.0+optimize+pic+shared arch=darwin-bigsur-skylake
  -   xelfobh      ^openmpi@4.1.1%apple-clang@13.0.0~atomics~cuda~cxx~cxx_exceptions+gpfs~internal-hwloc~java~legacy
  -   zruns75          ^hwloc@2.6.0%apple-clang@13.0.0~cairo~cuda~gl~libudev+libxml2~netloc~nvml~opencl~pci~rocm+sha
  -   ib4fnkf              ^libxml2@2.9.12%apple-clang@13.0.0~python arch=darwin-bigsur-skylake
  -   dwiv2ys                  ^xz@5.2.5%apple-clang@13.0.0~pic libs=shared,static arch=darwin-bigsur-skylake
  -   blitnbl          ^libevent@2.1.12%apple-clang@13.0.0+openssl arch=darwin-bigsur-skylake
  -   h7jalyu          ^openssh@8.7p1%apple-clang@13.0.0 arch=darwin-bigsur-skylake
  -   7v7bqx2              ^libedit@3.1-20210216%apple-clang@13.0.0 arch=darwin-bigsur-skylake
```

Pure hash-based reuse: all misses

```
(spackle):spack> spack solve --reuse -Il hdf5
==> Best of 10 considered solutions.
==> Optimization Criteria:
  Priority  Criterion                                                 Installed  ToBuild
  1         number of packages to build (vs. reuse)                           -        4
  2         deprecated versions used                                          0        0
  3         version weight                                                    0        0
  4         number of non-default variants (roots)                            0        0
  5         preferred providers for roots                                     0        0
  6         default values of variants not being used (roots)                 0        0
  7         number of non-default variants (non-roots)                        2        0
  8         preferred providers (non-roots)                                   0        0
  9         compiler mismatches                                               0        0
  10        OS mismatches                                                     0        0
  11        non-preferred OS's                                                0        0
  12        version badness                                                   6        0
  13        default values of variants not being used (non-roots)             1        0
  14        non-preferred compilers                                          15        4
  15        target mismatches                                                 0        0
  16        non-preferred targets                                             0        0

  -   yfkfnsp  hdf5@1.10.7%apple-clang@12.0.5~cxx~fortran~hl~ipo~java+mpi+shared~szip~threadsafe+tools api=defaul
 [+]  zd4m26e      ^cmake@3.21.1%apple-clang@12.0.5~doc+ncurses+openssl+ownlibs~qt build_type=Release arch=darwin
 [+]  53i52xr          ^ncurses@6.2%apple-clang@12.0.5~symlinks+termlib abi=none arch=darwin-bigsur-skylake
 [+]  us36bwr          ^openssl@1.1.1l%apple-clang@12.0.5~docs+systemcerts arch=darwin-bigsur-skylake
 [+]  74mwnxg              ^zlib@1.2.11%apple-clang@12.0.5+optimize+pic+shared arch=darwin-bigsur-skylake
  -   3ijfnel      ^openmpi@4.1.1%apple-clang@12.0.5~atomics~cuda~cxx~cxx_exceptions+gpfs~internal-hwloc~java~lega
  -   jxexyb7          ^hwloc@2.6.0%apple-clang@12.0.5~cairo~cuda~gl~libudev+libxml2~netloc~nvml~opencl~pci~rocm+
 [+]  ckdn5zf              ^libxml2@2.9.12%apple-clang@12.0.5~python arch=darwin-bigsur-skylake
 [+]  k7auat3                  ^libiconv@1.16%apple-clang@12.0.5 libs=shared,static arch=darwin-bigsur-skylake
 [+]  k2yumgx                  ^xz@5.2.5%apple-clang@12.0.5~pic libs=shared,static arch=darwin-bigsur-skylake
 [+]  grgtlcd              ^pkgconf@1.8.0%apple-clang@12.0.5 arch=darwin-bigsur-skylake
  -   nnc66ug          ^libevent@2.1.12%apple-clang@12.0.5+openssl arch=darwin-bigsur-skylake
 [+]  63xbksk          ^openssh@8.6p1%apple-clang@12.0.5 arch=darwin-bigsur-skylake
 [+]  snhgldt              ^libedit@3.1-20210216%apple-clang@12.0.5 arch=darwin-bigsur-skylake
 [+]  qbkmtdd          ^perl@5.34.0%apple-clang@12.0.5+cpanm+shared+threads arch=darwin-bigsur-skylake
 [+]  tnvkifs              ^berkeley-db@18.1.40%apple-clang@12.0.5+cxx~docs+stl patches=b231fcc4d5cff05e5c3a4814f
 [+]  7d5woqt              ^bzip2@1.0.8%apple-clang@12.0.5~debug~pic+shared arch=darwin-bigsur-skylake
 [+]  vh6di3i              ^gdbm@1.19%apple-clang@12.0.5 arch=darwin-bigsur-skylake
 [+]  qgy3v4l                  ^readline@8.1%apple-clang@12.0.5 arch=darwin-bigsur-skylake
```

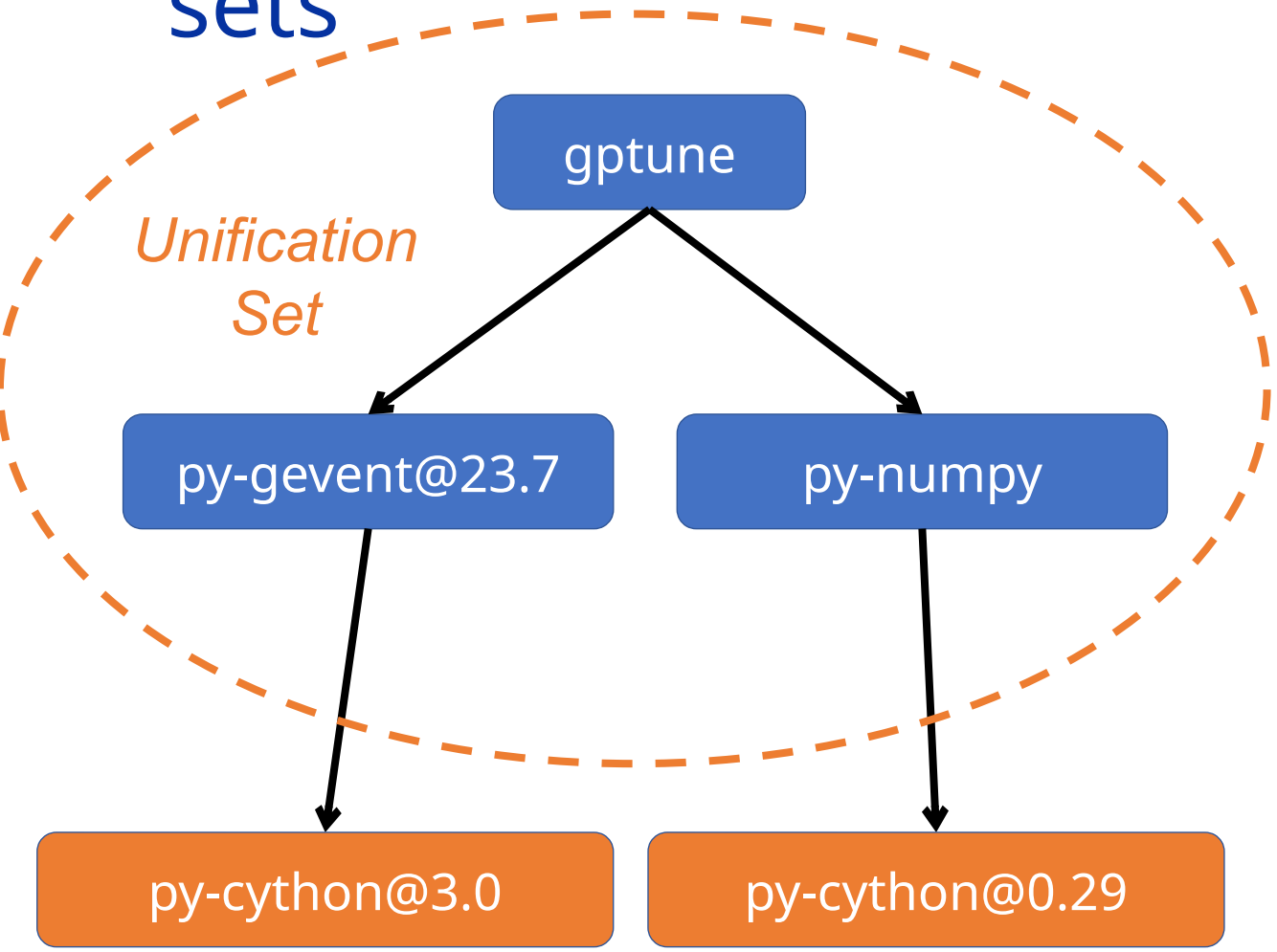
With reuse: 16 packages were actually acceptable

Third challenge: we needed to allow multiple versions of build dependencies in the DAG



- Only one configuration per package allowed in the DAG
  - Ensures ABI compatibility but is too restrictive
- Needed to enable compiler mixing with compiler dependencies
- Also needed for Python ecosystem
  - In the example py-numpy needs to use py-cython@0.29 as a build tool
  - That enforces using an old py-gevent, because newer versions depend on py-cython@3.0 or greater

Solution: Model process spaces as unification sets



- The constraint on build dependencies can be relaxed, without compromising the ABI compatibility
- Having a single configuration of a package is now enforced on unification sets
- These are the set of nodes used together at runtime (the one shown is for gptune)
- This allows us to use the latest version of py-gevent, because now we can have two versions of py-cython

We dynamically “split” nodes when needed

1. Start with deducing single dependency nodes:

```
node(DependencyName)
   :- dependency_holds(PkgName, DependencyName)
```

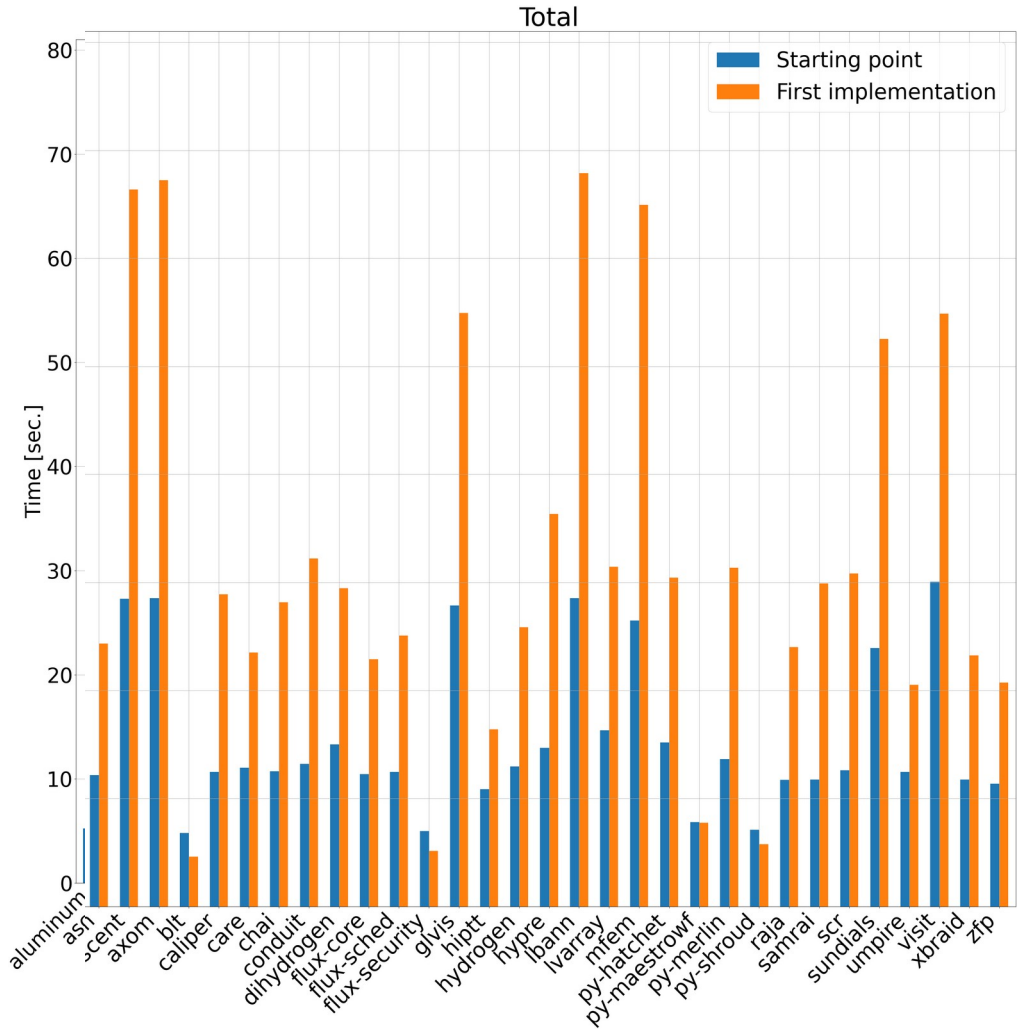
2. Allow solver to **choose** to duplicate a node:

Converted node identifier from **name** to (**name, id)**

```
  1 {
     depends_on(PkgNode, node(0..Y-1, DepNode), Type)
     : max_dupes(DepNode, Y)
  } 1
     :- dependency_holds(PkgNode, DepNode).
```

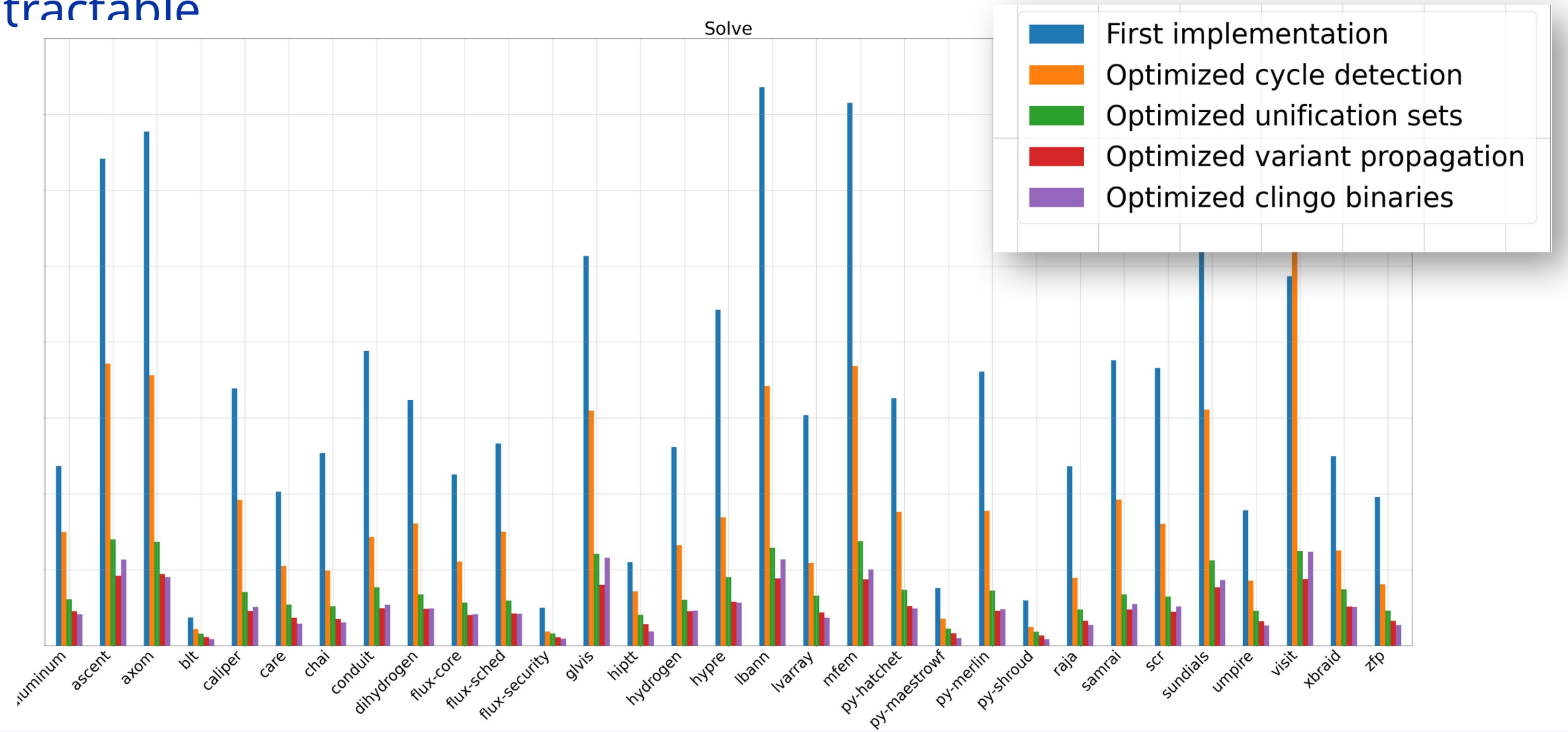
3. Re-encode package metadata so that it can be associated with duplicates

First try at allowing duplicates in a single solve



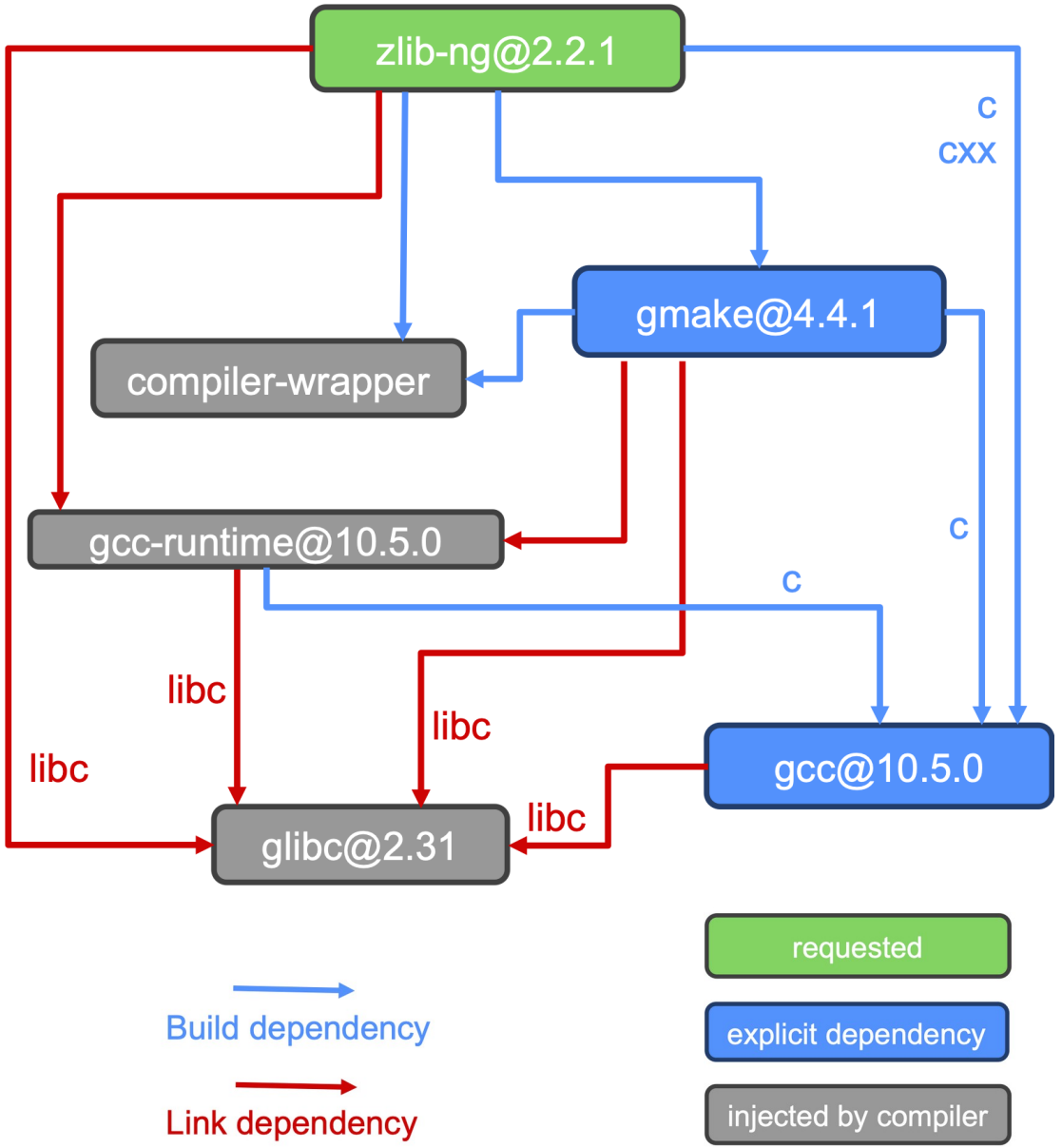
Increased runtimes by >> 2x in some cases

Through many different optimizations, we were able to reclaim enough performance to make duplicate build dependencies tractable



Compiler Dependencies

- Compilers are now build dependencies
- Runtime libraries modeled as packages
  - gcc-runtime is injected as 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:

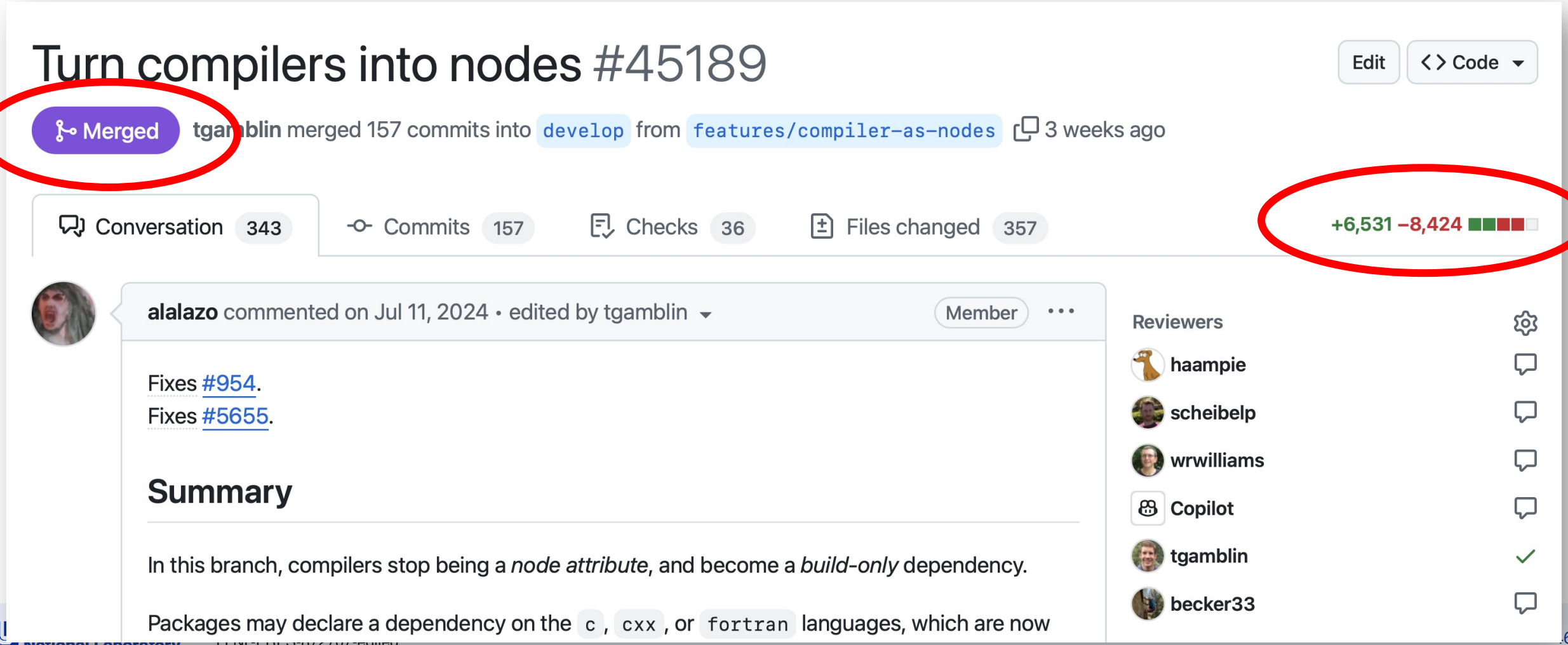
```
pkg +foo %gcc      # +foo associates with pkg
```

- 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! Spack 1.0 released in July 2025



Turn compilers into nodes #45189

Edit · Code

Merged · tgamblin merged 157 commits into `develop` from `features/compiler-as-nodes` · 3 weeks ago

Conversation 343 · Commits 157 · Checks 36 · Files changed 357 · +6,531 −8,424

alalazo commented on Jul 11, 2024 • edited by tgamblin · Member

Fixes #954.
Fixes #5655.

Summary

In this branch, compilers stop being a *node attribute*, and become a *build-only* dependency.

Packages may declare a dependency on the `c`, `cxx`, or `fortran` languages, which are now

Reviewers

- haampie
- scheibelp
- wrwilliams
- Copilot
- tgamblin ✓
- becker33

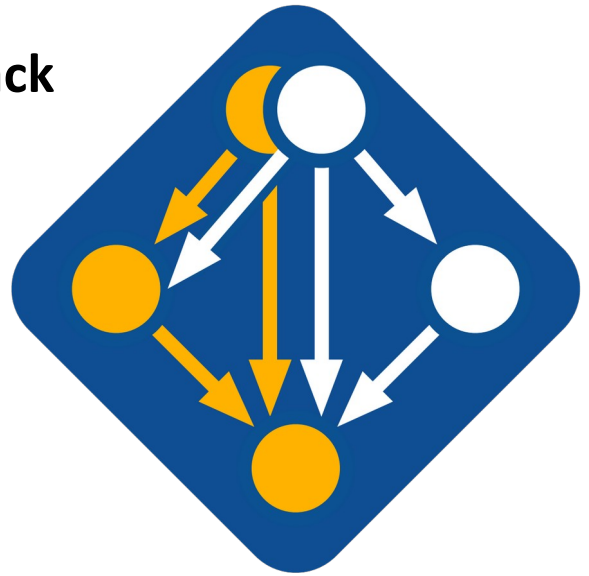
Final bits

- We added a larger concept of a “toolchain” to Spack
  - 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
- We separated the builtin package repo from Spack
  - Spack packages live in a separate GitHub repository
    - Need Spack to bootstrap this new repository
    - Will need to download automatically on first install
  - Built system classes moved into the package repository
  - Compiler wrappers are now a real package

Talk to us

- There are lots of ways to get involved with Spack!
  Contribute packages, documentation, or features at **github.com/spack/spack**
  - Contribute your configurations to **github.com/spack/spack-configs**

- Talk to us!
  - Join our **Slack channel** (slack.spack.io)
  - Submit **GitHub issues** and **pull requests**!
  - Ask us at the conference (or later) for email addresses



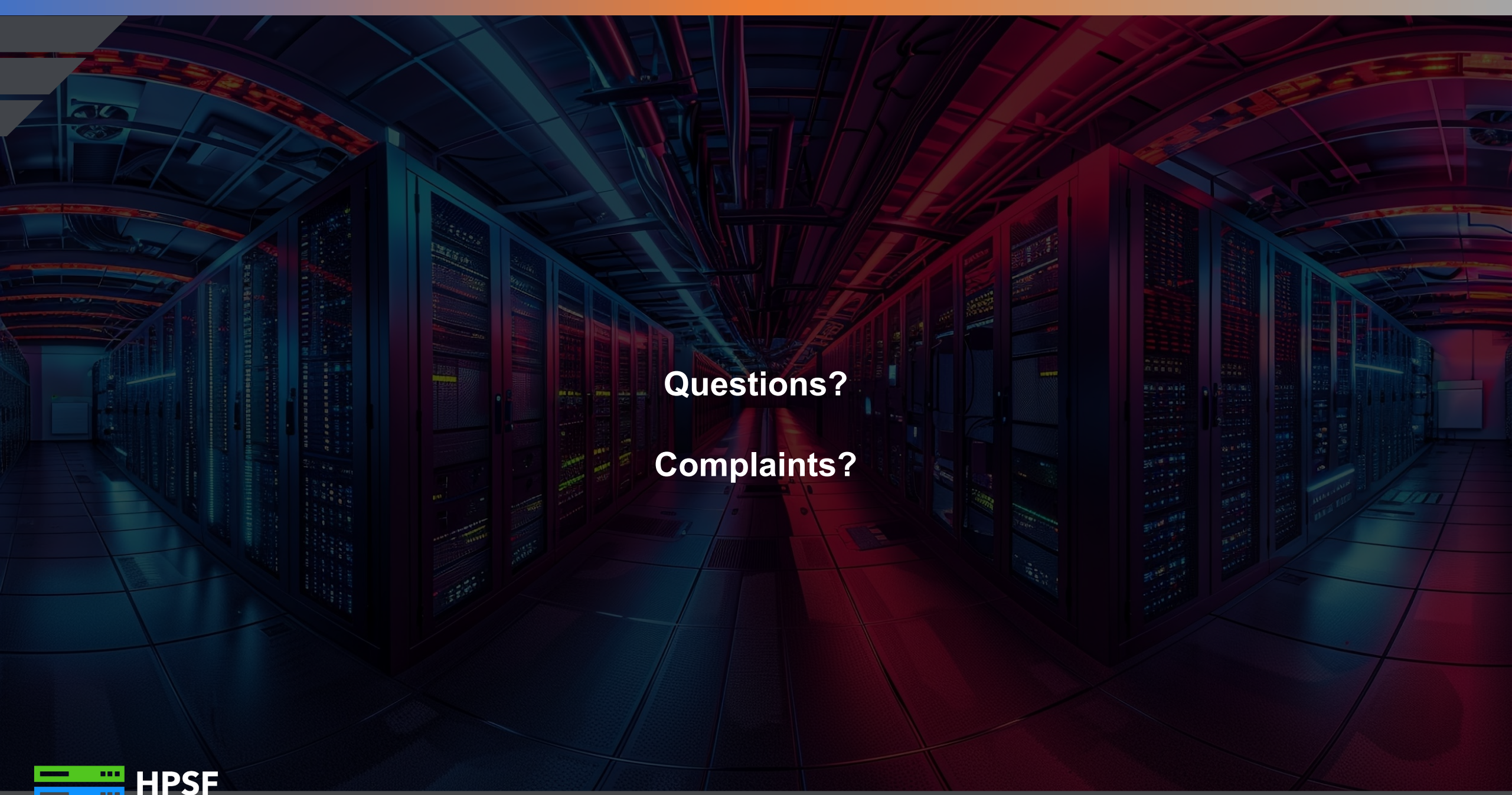
Star Spack on GitHub!

github.com/spack/spack

Follow Spack on ~~Twitter~~ X!

@spackpm

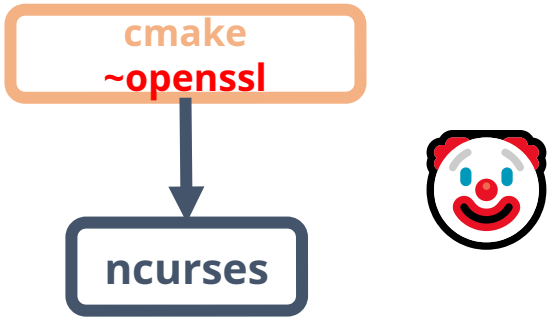
We hope to make distributing & using HPC software easy!


Questions?
Complaints?

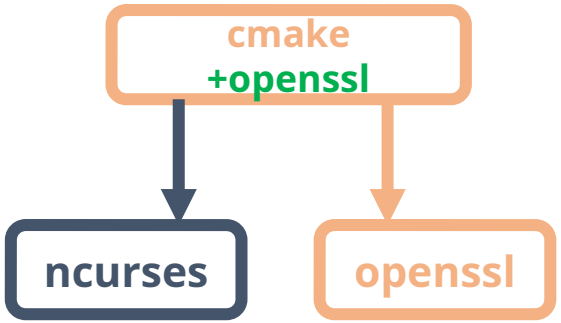
Art vs. science: reusing builds is not quite enough

- We get strange behavior when we have to build new packages
  - E.g.: Cmake depends on openssl for https
  - Minimizing builds will toggle this feature *off* to avoid a dependency
- **We want to prioritize reusing a package *if* the user already installed it**
  - Has to be more important than defaults, or we would never reuse
- **We want to prioritize package defaults *if* the package had to be built anyway**
  - Make *new* builds follow defaults

How can we do both?



minimize builds > package defaults

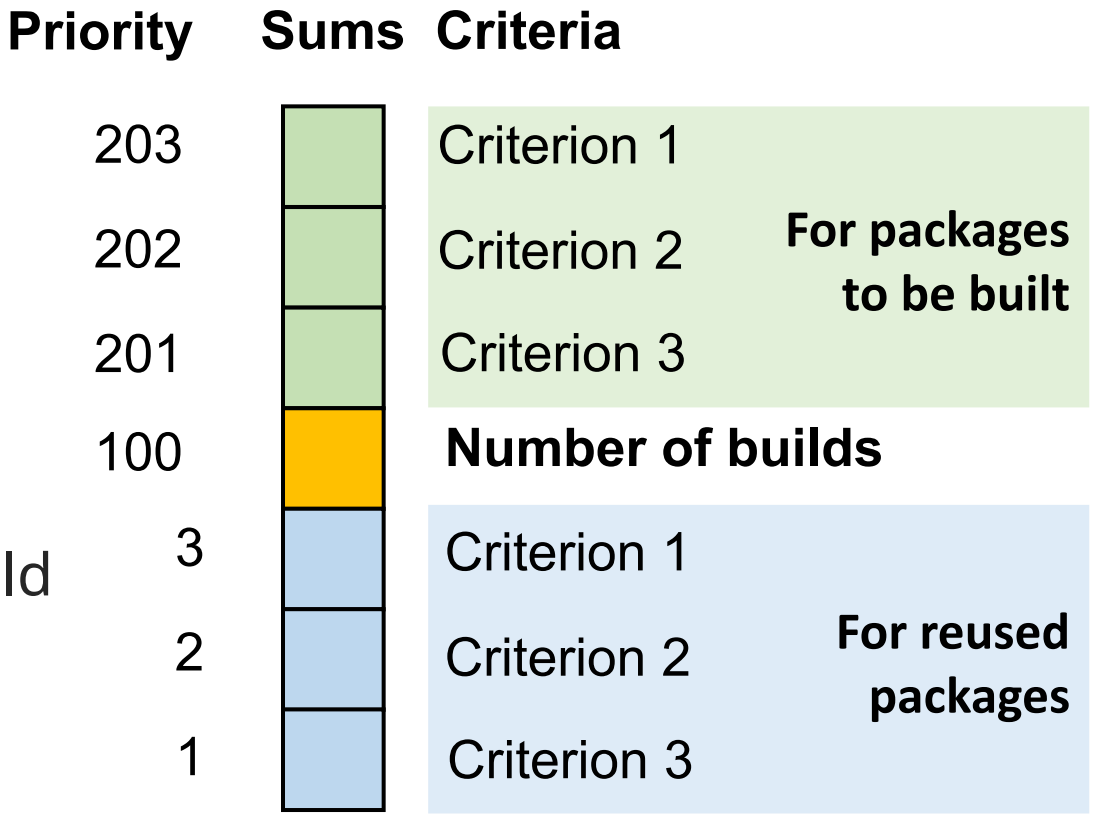


package defaults > minimize builds

We devised a two-level optimization scheme

```
build_priority(P, 200) :- build(P),     node(P).
build_priority(P, 0)   :- not build(P), node(P).

% priority + 200 IF we are building
#minimize{
   W@2+Priority,P
   : version_weight(P, W), build_priority(P, Priority)
}.
```

- Minimize builds, *unless we have to build*
  - Prioritize *defaults* for specs we *have* to build
- Last trick to get this to work:
  - All criteria must be formulated as minimizations
  - No built configuration can be “better” than a reused configuration



Objective vectors of sums are compared lexicographically from highest to lowest priority

ASP searches for *stable models* of the input program

- Stable models are also called ***answer sets***
- A ***stable model*** (loosely) is a set of true atoms that can be deduced from the inputs, where every rule is idempotent.
  - Similar to fixpoints
  - Put more simply: a *set of atoms where all your rules are true!*
- Unlike Prolog:
  - Stable models contain everything that can be derived (vs. just querying values)
  - ASP is guaranteed to complete!

What's in a solve?

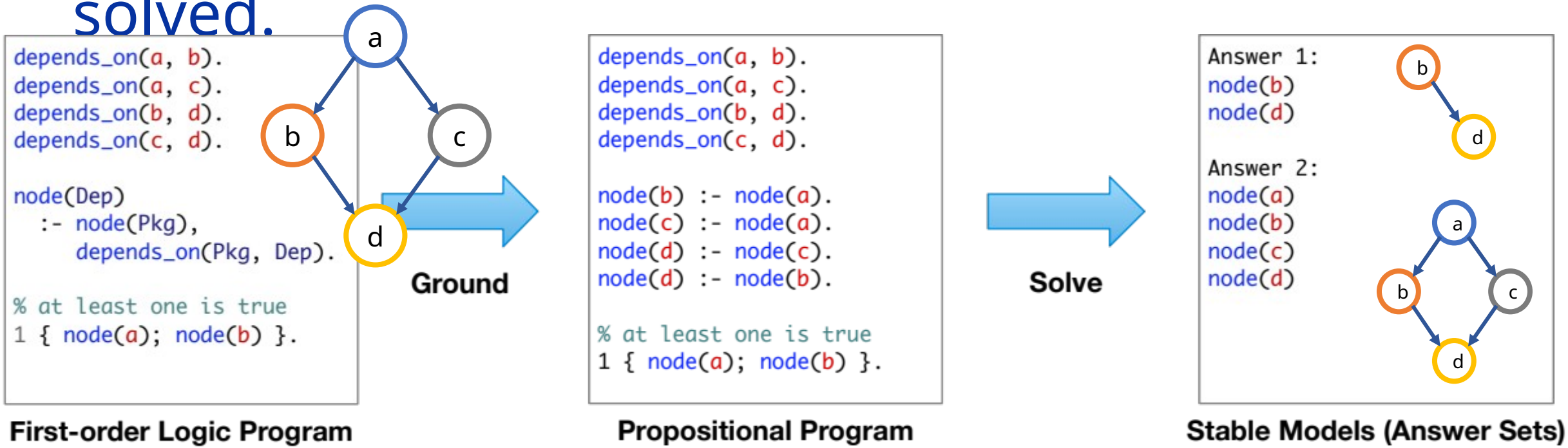
I. Setup
- Compute all possible dependencies
- Load each dependency `package.py` to get metadata
- Translate metadata from specs to ASP "facts"

II. Ground
- Instantiate first-order logic program with facts from problem instance

III. Solve
- Run grounded program

IV. Reconstruct
- Translate facts from solution back to a resolved graph

Grounding converts a first-order logic program into a propositional logic program, which can be solved.



Answer 1: Only node(b) is true

Answer 2: Both node(a) and node(b) are true

- Logic program is ~250 rules and 20-50 optimizations, around 1200+ lines of ASP
- Solves have hundreds of thousands of facts and (low) millions of grounded atoms

We use optimization to choose the “best” of all valid solutions

- Tend to be a lot of *valid* solutions
- Choosing configurations that are “intuitive” to users can be difficult
- Some criteria have to be simplified due to solver efficiency
  - Would like DAG-level precedence for optimization criteria
  - Had to settle for roots vs. others
  - Can be unintuitive in some cases due to aggregations
    - Solver choices can be apples to oranges

Priority	Criterion (to be minimized)
1	Deprecated versions used
2	Version oldness (roots)
3	Non-default variant values (roots)
4	Non-preferred providers (roots)
5	Unused default variant values (roots)
6	Non-default variant values (non-roots)
7	Non-preferred providers (non-roots)
8	Compiler mismatches
9	OS mismatches
10	Non-preferred OS’s
11	Version oldness (non-roots)
12	Unused default variant values (non-roots)
13	Non-preferred compilers
14	Target mismatches
15	Non-preferred targets

15 original optimization criteria (now there are 22)

Complicated logic can become very simple with ASP

- Example: every node in the DAG has a compiler and a target microarchitecture
  - x86_64, haswell, broadwell, skylake, cascadelake, etc.
  - Some compilers don't support generating code for some targets
  - We want to pick the best target possible for each compiler
- We must ensure that we pick a build target for which the compiler can generate code

Each node has 1 target assigned

Disallow cases where the compiler doesn't support the target.

Minimize the total weight of all targets

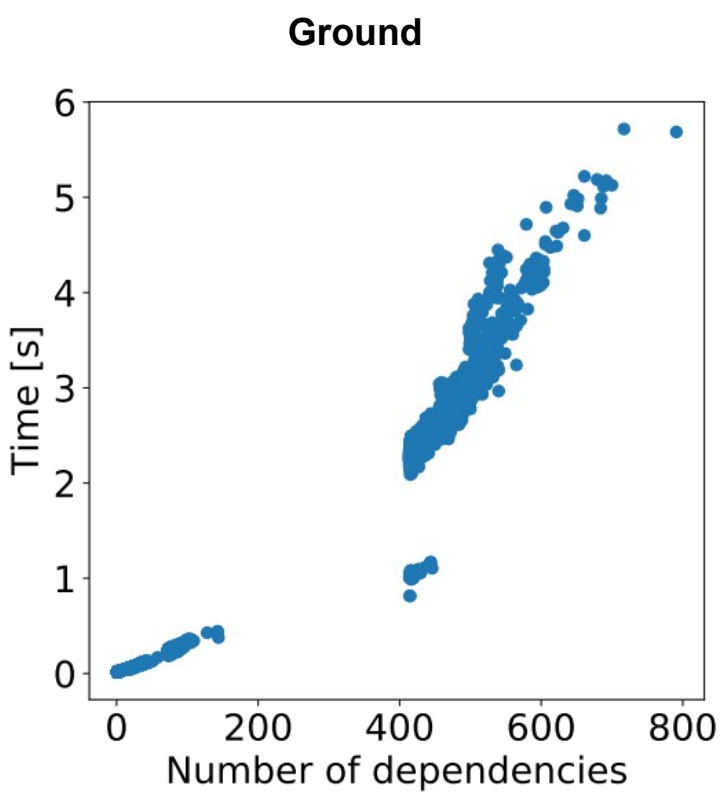
```
% one target per node -- optimization will pick the "best" one
1 { node_target(P, T) : target(T) } 1 :- node(P).

% can't use targets on node if the compiler for the node doesn't support them
:- node_target(P, T), not compiler_supports_target(C, V, T),
   node_compiler(P, C), node_compiler_version(P, C, V).

% if a target is set explicitly, respect it
node_target(P, T) :- node(P), node_target_set(P, T).

% each node has the weight of its assigned target
node_target_weight(P, N) :- node(P), node_target(P, T), target_weight(T, N).
#minimize{ N@5,P : node_target_weight(P, N) }.
```

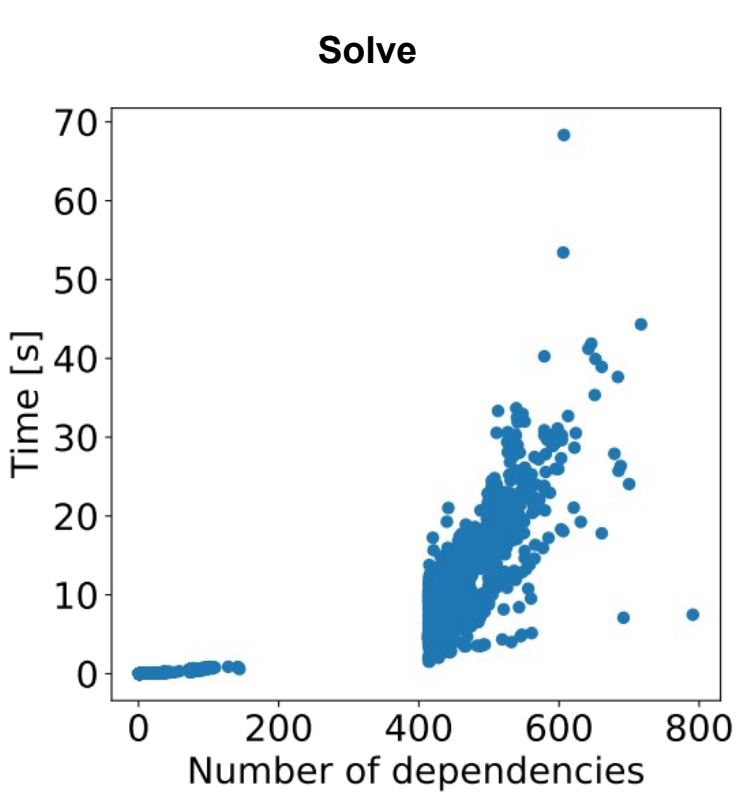
In v0.23, we were ready to add a first version of language dependencies

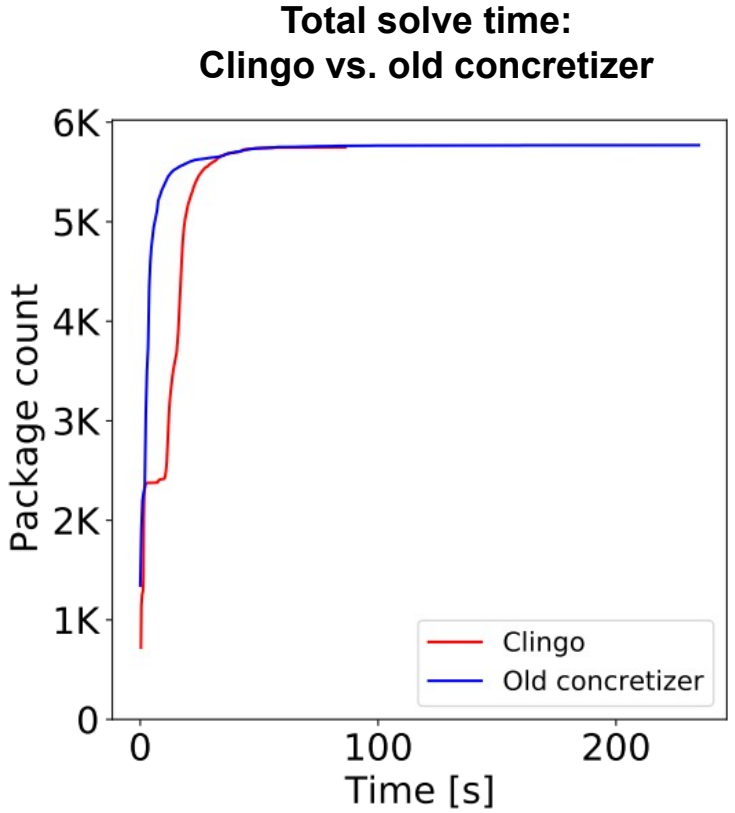
```
depends_on("c", type="build")
depends_on("cxx", type="build")
depends_on("fortran", type="build")
```

- 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)

Ground and solve times of this approach are acceptable compared to the old, incomplete, heuristic approach



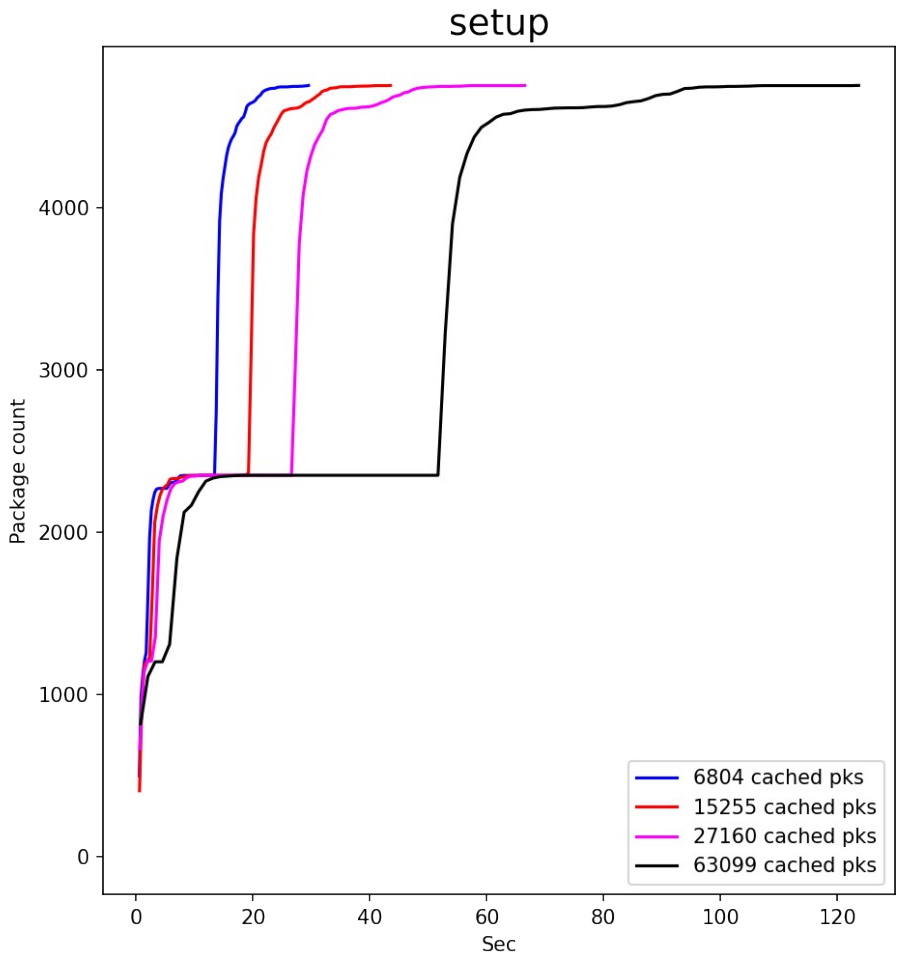




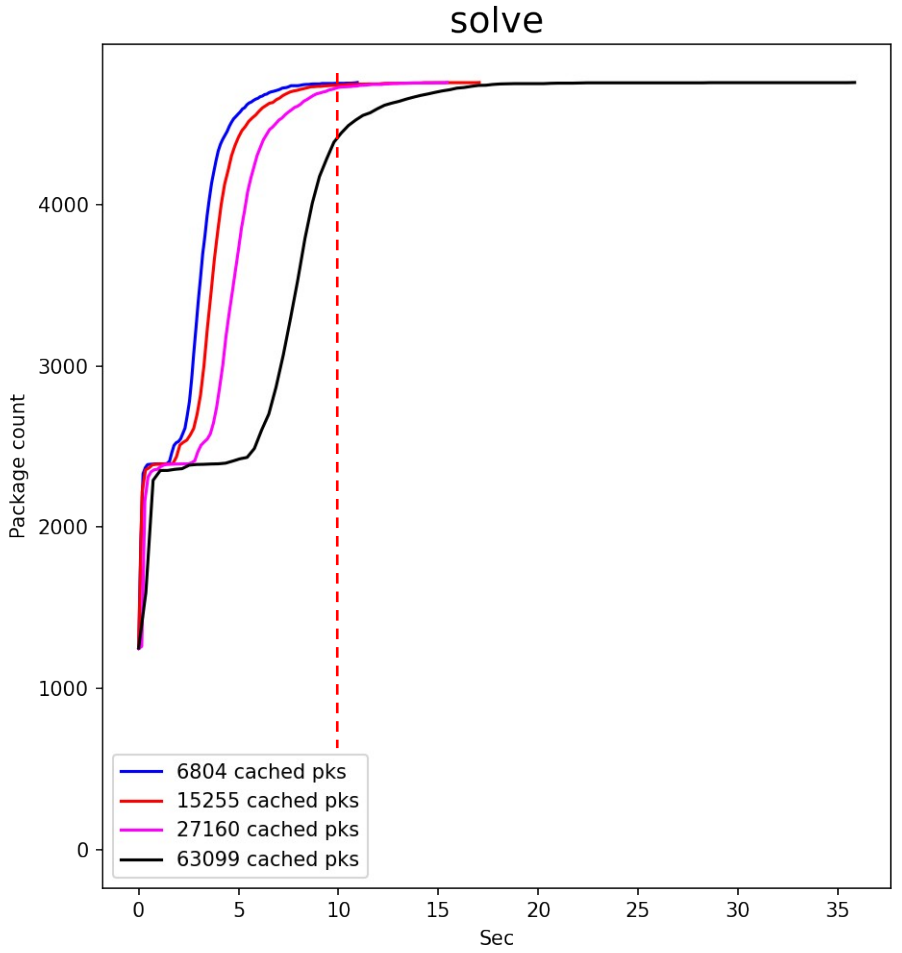
- Performance measurements from Xeon E5-2695 v4 (Broadwell) node
  - Most solves take seconds, with longer solves for very large packages
  - Jump in dependency count comes from packages that indirectly depend on MPI

We can handle *very* large problem sizes with the reusing concretizer

- Cumulative distribution of setup and solve times
- Hypothesis: we don't see big combinatorial blow-up b/c we're strict about dependency hashes
- Next: try mixed ABI, but *prefer* "pure" source-built dependencies



Most of the time is spent in setup
(reading data in Python – can be sped up w/caching)



Even with 63k packages in a repo,
nearly all package solves take < 10 sec

Cycle detection in the solver is *expensive*

```
path(A, B) :- depends_on(A, B).
path(A, C) :- path(A, B), depends_on(B, C).

% this constraint says "no cycles"
:- path(A, B), path(B, A).
```

Has to maintain path() predicate representing paths between nodes
Cycles are actually rare in solutions
    Switched to post-processing for cycle detection
    Only do expensive solve if a cycle is detected in a solution
Eventually moved this calculation *into* the solver
using some custom directives from the developers

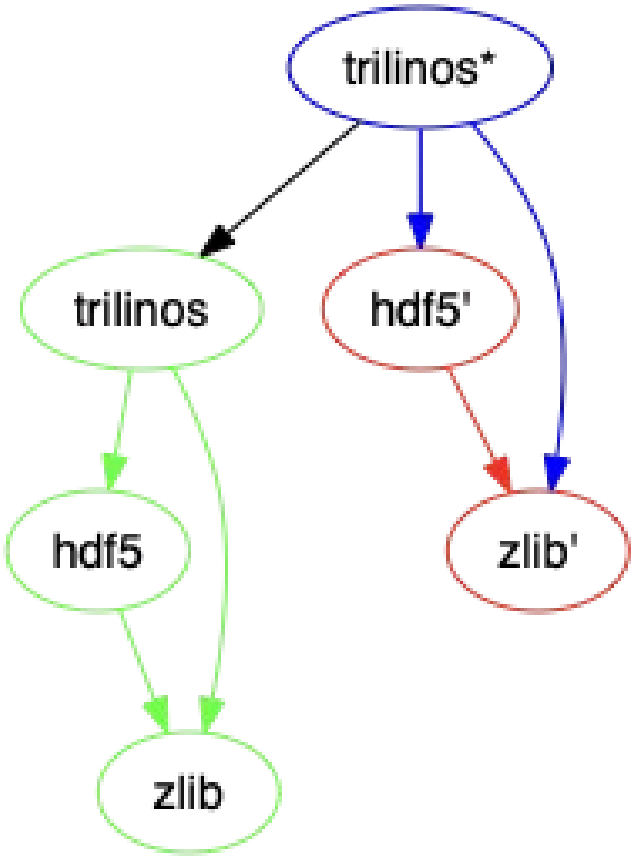
50%+ improvement in solve time

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

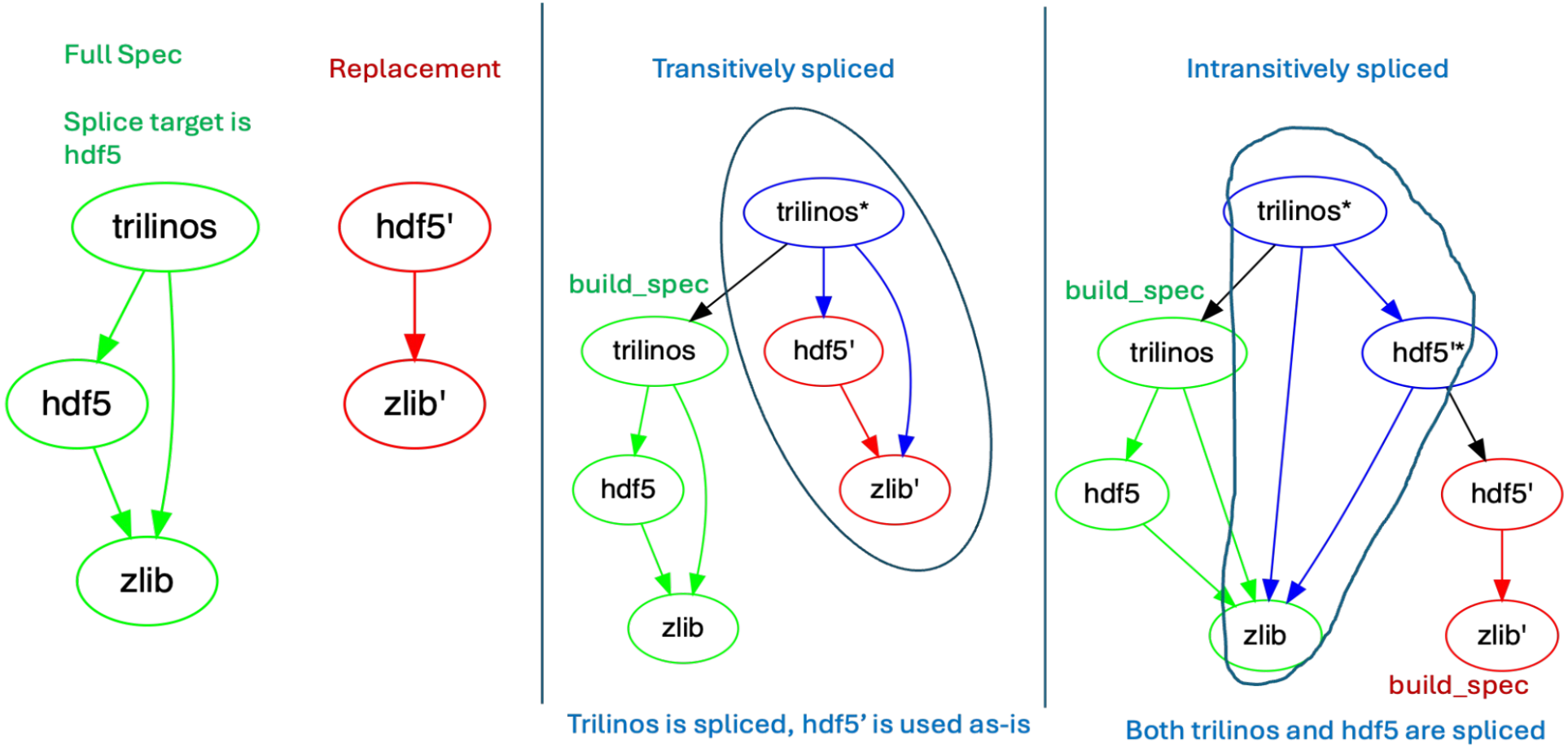
Relocation logic is repurposed for “rewiring” spec to its new configuration



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.