Bridging the Gap Between Binary and Source Based Package Management in Spack

John Gouwar
gouwar.j@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

Gregory Becker
becker33@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Tamara Dahlgren
dahlgren1@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Nathan Hanford
nhanford@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

Arjun Guha
a.guha@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

Todd Gamblin
tgamblin@llnl.gov
Lawrence Livermore National Laboratory
Livermore, California, USA

ABSTRACT

Binary package managers install software quickly but they limit configurability due to rigid ABI requirements that ensure compatibility between binaries. Source package managers provide flexibility in building software, but compilation can be slow. For example, installing an HPC code with a new MPI implementation may result in a full rebuild. Spack, a widely deployed, HPC-focused package manager, can use source and pre-compiled binaries, but lacks a binary compatibility model, so it cannot mix binaries not built together. We present *splicing*, an extension to Spack that models binary compatibility between packages and allows seamless mixing of source and binary distributions. Splicing augments Spack's packaging language and dependency resolution engine to reuse compatible binaries but maintains the flexibility of source builds. It incurs minimal installation-time overhead and allows rapid installation from binaries, even for ABI-sensitive dependencies like MPI that would otherwise require many rebuilds.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Interoperability*; Domain specific languages.

KEYWORDS

ABI-compatibility, Dependency management, Answer set programming, Binary reuse

ACM Reference Format:
John Gouwar, Gregory Becker, Tamara Dahlgren, Nathan Hanford, Arjun Guha, and Todd Gamblin. 2025. Bridging the Gap Between Binary and Source Based Package Management in Spack . In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3712285.3759791

SC '25, November 16–21, 2025, St Louis, MO, USA, LLNL-CONF-2010453

ACM ISBN 979-8-4007-1466-5/2025/11.
https://doi.org/10.1145/3712285.3759791

1 INTRODUCTION

When developing and deploying high-performance software there is an inherent trade off between building from source and using pre-compiled binaries. Building from source provides full control over micro-architectural optimizations and conditional features offered in bleeding-edge scientific software. However, this customizability is often complex and time consuming, especially given the large numbers of dependencies that must be compatibly built for HPC. The more supported features and dependencies on third party libraries a package has, the greater the difficulty in configuring and building the software completely from source. The ability to use pre-compiled libraries can increase the speed and ease of installation but at the cost of locking into potentially limited configuration options that may not match the user's needs.

Spack [14] eliminates that sacrifice by providing configurability with the relative ease of using pre-compiled libraries, either those previously built from source by Spack or those that already exist on the system. Spack-built libraries may be installed locally on the system or fetched from an external binary cache. For example, package updates in the Spack repository trigger creation of binaries that populate a public cache available to the Spack community[1]. Software already on the system can include pre-compiled libraries that Spack cannot build, such as vendor-specific MPI implementations, or hardware abstractions.

When using pre-compiled libraries that may have been built separately, it is important to ensure that their *application binary interfaces* (ABIs) are compatible. ABI-compatible libraries agree on the compiled names of shared symbols, the calling conventions for functions, and the size and layout of shared types. Spack only allows *one* implementation of each dependency in any installation's *directed acyclic graph* (DAG) of dependencies. This trivially ensures ABI compatibility, because every compilation that uses a given dependency will include the same headers, ensuring consistency. An installation that built with a *different* version of the dependency cannot occur in the same DAG. If the user requires a different version (or a different implementation) of some dependency, all libraries built with the prior version must be rebuilt.

[1]https://spack.io/spack-binary-packages/

On its face, this may not seem like too much of a restriction; however, it is overly conservative and prevents reuse in cases where a different build of a dependency would retain ABI-compatibility. To illustrate the consequences of this restriction, consider deploying the widely used solver suite TRILINOS [20] on an HPE Cray cluster. TRILINOS has a dependency on an MPI implementation and HPE Cray recommends using their specialized MPI implementation (CRAY MPICH) on their hardware [19]. Notably, CRAY MPICH is only available on HPE Cray systems, but is ABI-compatible with the general purpose MPICH. Given Spack's current restrictions, users would be forced to rebuild all of TRILINOS on the cluster in order to use CRAY MPICH even though it is safe to build it in advance against a compatible MPICH and simply link against CRAY MPICH. This becomes especially painful when trying to *distribute* MPI binaries. Ideally, we would build a stack of software with the publicly available MPICH and allow it to be installed without rebuilding on on any system with an ABI-compatible, optimized MPI implementation. Because Spack currently cannot describe when two packages are ABI-compatible, and because it cannot represent heterogeneous binaries built with *different* dependency implementations, it cannot effectively reuse binaries that depend on MPI, and it must rebuild.

In this paper, we present an extension to Spack called *splicing*. We augment Spack with a model of package ABI-compatibility and a model of heterogeneous dependencies that may not have been initially compiled together. Splicing allows Spack to replace dependencies of pre-compiled libraries with ABI-compatible substitutes. We add a single directive, `can_splice`, to Spack's domain specific language (DSL), which allows developers to specify when compiled configurations of libraries are ABI-compatible. We also extend Spack's dependency resolution mechanism to synthesize ABI-compatible splices when using pre-compiled libraries. These features create an additional dependency structure on the Spack ecosystem that allows for specifying the ABI compatibility of a package's compiled representations and how those can be safely replaced as the dependency of another pre-compiled library. In summary, we provide the following contributions:

(1) A formal model of *splicing*, a representation of binary relinking in Spack that maintains full build provenance;
(2) An augmentation to Spack's packaging language for specifying ABI compatibility between package configurations;
(3) An implementation of automatic splice synthesis in Spack's solver and re-linking in Spack's installer; and
(4) An empirical evaluation of scaling, performance, and correctness of dependency resolution augmented with splicing.

2 BACKGROUND

2.1 API vs. ABI Compatibility

There are two forms of application interface compatibility: program and binary. An application programming interface (API) is a source-level contract describing the interface that a library provides to others. It includes the signatures of functions and names of exported types. A package X is API-compatible with a package Y if X implements a superset of Y's interface. That is, any package which uses Y's interface may safely use X's implementation of that interface, even though X may provide additional functionality. For C libraries, this information is encapsulated in header files, and API-compatible packages implement equivalent headers. An application binary interface (ABI) describes the boundary between two independently compiled software artifacts. This includes the compiler-mangled names of exported symbols and the layout of exported types. A compiled package X is ABI-compatible with a compiled package Y if it is API compatible *and* its compiled representation has the same mangled names and type layouts.

While API compatibility is necessary for ABI compatibility, it is not sufficient. For example, if a user-defined type in a particular interface is opaque, like MPI's `MPI_Comm`, code that treats this type opaquely at the source level may be safely compiled against any API-compatible implementation of MPI, but once compiled it can only be safely used with code that implements the type in the *same* way that it was compiled. The C ABI for language-specified types (*e.g.*, ints, floats, structs *etc.*) is rigorously specified for any particular architecture. Opaque type incompatibility at the binary level actually arises between OPEN MPI [12] and MPICH [16]. While OPEN MPI implements `MPI_Comm` as an incomplete struct pointer, MPICH implements it as a 32-bit integer [17]. Binaries compiled against one implementation cannot safely use the other.

2.2 Package Managers

Package managers provide tools to mitigate the complexity of installing software. They can solve user-imposed constraints, handle dependencies, build packages from source, and install precompiled binaries. Most package managers have either a *binary* or *source* based deployment paradigm. Binary package managers, like those included with most major Linux distributions (Gentoo Linux being a notable exception), select and download software versions as pre-compiled binary artifacts (*e.g.*, shared libraries, executables, *etc.*). Binary package managers allow for quick installation, but they limit package versions to specific pre-compiled configurations. They also require that all expressed dependency constraints be between ABI-compatible versions, since the packages installed by a user will be compiled independently.

Source-based package managers download source code, build it, and deploy the resulting software. In addition to installing files, they must manage the associated build or interpretation. Generally, all dependent packages of a source build will be compiled consistently, i.e., only a single version of any dependency will be used, and flags are kept consistent across all builds. Package developers can thus express dependencies at the API level. ABI compatibility follows by virtue of the consistent compilation model. While the lack of ABI modeling allows for more expressiveness, updates to packages with many dependents trigger long cascades of recompilation. Updating to an ABI incompatible version of a package in an existing dependency tree will require all *dependent packages* to rebuild against the new version. These rebuilds in the dependent packages can then require rebuilds of their own dependent packages, with required rebuilds potentially cascading to the whole dependency tree. This need to *rebuild the world* can have a chillling effect on a developer's willingness to update a single dependency. Furthermore, not all source code is open and developers may have to rely on some software that is only available as a binary, especially in the HPC space. This means that source based package managers will also build

Spec Sigil	Example	Meaning
`@`	`hdf5@1.14.5`	Require version
`+`	`hdf5+cxx`	Require variant
`~`	`hdf5~mpi`	Disable variant
`^`	`hdf5 ^zlib`	Depends on (link-run)
`%`	`hdf5%clang`	Depends on (build)
`key=value`	`hdf5 target=icelake` `hdf5 api=default`	Require variant or target to hold value

Table 1: Examples of spec syntax, and their meaning.

with binary packages out of necessity, even if the source code was never available to the package manager.

3 SPACK

Spack is a source *and* binary package manager, widely used in HPC. This section describes Spack's *packages*, *specs*, *concretizer*, and *binary relocation*, core components of configuration and dependency resolution that provide needed context for splicing.

3.1 Specs

Specs are the core of Spack; they allow users to concisely specify build configuartions and constraints. Each spec has six attributes: (1) the package name; (2) the version to build; (3) variant values (*i.e.*, compile-time options); (4) the target operating system; (5) the target microarchitecture; (6) and the specs of the packages it depends on (and their attributes). Spack provides a concise syntax for specs, and Table 1 shows a subset of simple examples needed to understand this paper. A user typically provides an incomplete description, or *abstract spec*, by not explicitly constraining all attributes. In contrast, a spec with all attributes set is called a *concrete spec*. Concrete specs can be built and installed, as they contain all information the build environment might need to query in order to configure and build. The rest of the paper is concerned mainly with concrete specs.

Specs are recursive in that dependencies (attribute (6) above) are specs themselves. They are therefore represented in memory as directed acyclic multigraphs. Nodes correspond to packages with their attributes, a directed edge (A, B) denoting that package A *depends on* package B, and two edge sets corresponding to build, and link-run dependencies. Taking the union of all edge sets and forming a single DAG admits a performant hashing scheme on specs, allowing for efficient reasoning about equality between specs.

3.2 Packages

A Spack *package* defines the build process for a software product using available options. It defines a combinatorial space of build options that can include software versions, optional features, compile-time flags, dependencies on other packages, potential conflicts with other packages, and environmental constraints (*e.g.*, requiring an x86_64 system, requiring a CUDA installation, *etc.*). The software is described in a DSL embedded in Python that uses a Python class to represent every build configuration, a *parameterized* build process, and the artifacts (*e.g.*, tarballs and patches) needed to build. Figure 1 shows a simple Spack package. Package configuration features, such as `version`, `depends_on`, and `variant`, are specified with *directives*. Directives serve to define the configuration space

```
1  class Example(Package):
2      """Example depends on zlib, mpi and optionally bzip2"""
3      # This package provides two versions
4      version("1.1.0")
5      version("1.0.0")
6
7      # Optional bzip support
8      variant("bzip", default=True)
9
10     # Depends on bzip2 when bzip support is enabled
11     depends_on("bzip2", when="+bzip")
12     # Version 1.0.0 depends on an older version of zlib
13     depends_on("zlib@1.2" when="@1.0.0")
14     # Version 1.1.0 depends on a newer version of zlib
15     depends_on("zlib@1.3", when="@1.1.0")
16     # Depends on some implementation of MPI
17     depends_on("mpi")
18
19     ###
20     # example@1.1.0 can be spliced in for example@1.0.0
21     can_splice("example@1.0.0", when="@1.1.0")
22     # example@1.1.0+bzip can be spliced in for
23     # example-ng@2.3.2+compat
24     can_splice("example-ng@2.3.2+compat", when="@1.1.0+bzip")
```

Figure 1: An example `package.py` file demonstrating constraints in Spack's embedded DSL. This package also contains our new augmentation to the packaging language, `can_splice`, which we introduce in section 5.2.

of a package and may introduce constraints on the package or its dependencies.[2] In contrast to most package managers, where a package description specifies a single software configuration, Spack packages are *conditional*, and most directives accept a `when` argument. For example, the `depends_on` directive on line 13 constrains the `zlib` dependency to version `1.2` *when* the example package is at version `1.0.0`. Similarly, the directive on line 15 says that `example` version `1.1.0` requires a newer `zlib` version: `1.3`. Constraints in package directives are specified using the spec syntax (Section 3.1). Directives below `###` are part of our DSL extension (section 5.2).

For our purposes, there are two major classes of dependencies in Spack: build and link-run. Build dependencies are packages that a node needs to execute its build process; these can be compilers (*e.g.*, GCC, or CLANG), (meta-)build systems(*e.g.*, AUTOTOOLS, CMAKE, or NINJA), or even interpreters for build-time glue code (*e.g.*, PYTHON or PERL). Link-run dependencies are the packages that a node needs either at compile-time for linking (*i.e.*,shared object files for dynamic linking and assembly code for static linking), or at runtime to use either as a subprocess or as a dynamically loaded library.

3.3 The Concretizer

Spack's dependency resolver, or *concretizer*, takes abstract specs requested by a user and produces concrete specs that are valid according to constraints in package files. Since specs are recursive, this also includes resolving concrete specs for all of a package's dependencies. Consider the following abstract spec for the package from fig. 1, `example@1.0.0`; one such concretization could be:

```
example@1.0.0 +bzip arch=linux-centos8-skylake
    ^bzip2@1.0.8 ~debug+pic+shared arch=linux-centos8-skylake
    ^zlib@1.2.11 +optimize+pic+shared arch=linux-centos8-skylake
    ^mpich@3.1 pmi=pmix arch=linux-centos8-skylake
```

[2]Some directives simply attach metadata, such as the Github usernames of the maintainers, licensing information, or websites associated with the package.

Dependency resolution when considering only compatible versions is known to be NP-complete [5, 8], and Spack must also decide compatible build options, operating systems, and microarchitectures. Spack attempts to maximize its reuse of already-built components, either previously installed locally or present in some remote cache, which we collectively call *reusable specs*. To ensure completeness and optimality (*i.e.*, always finding the optimal solution if one exists) while maintaining tractability, Spack implements its concretizer using the Answer Set Programming (ASP) system Clingo [13, 15]. ASP is a logic programming paradigm, similar to Datalog, where problems are specified as first-order logic programs (*i.e.*, facts and deductive rules with variables) extended with a non-deterministic choice construct. ASP is distinguished from other logic programming paradigms by its lack of an operational semantics; instead, ASP programs are first *grounded* into propositional logic programs without variables, and then solved for their *stable models* using techniques from the SAT/SMT solving community such as the classic Davis–Putnam–Logemann–Loveland (DPLL) algorithm along with modern extensions like Conflict-Driven Clause Learning (CDCL) [6, 7, 18]. Since a given program may have many models in the presence of non-deterministic choices, Clingo allows optimization objectives to be defined over models of a program.

Spack's concretizer is implemented in three stages. First, constraints from the package classes, reusable specs, and the user-provided abstract specs are compiled to an encoding of facts in ASP. Then, these facts are included in a logic program describing the formal constraints of spec concretization (*e.g.*, every spec must be resolved to exactly one version, every variant must have a chosen value, *etc.*) and optimization objectives (*e.g.*, maximize the reusable specs, use the newest satisfying version of every package, *etc.*), which is then run with Clingo to solve for stable models. Finally, the optimal model is processed by an interpreter to construct concrete specs for further use by Spack. Notably, this output includes which specs are reused (*i.e.*, those which have already been built), and which ones must be built from source. Further details on the concretizer implementation, its optimization criteria, and the implementation of package reuse can be found in prior work [13].

3.4 Binary Relocation

Once Spack builds a package, it needs to ensure that each executable and library knows where to find its dependencies. On systems where there is only one version of each library, managed by the system package manager, the search path for libraries is usually stored either in global system configuration (*e.g.*, `ldconfig` or `ld.so.conf`), or in an environment variable (*e.g.*, `LD_LIBRARY_PATH`). A naive install algorithm would simply append installation paths for Spack-built binaries to system search paths. This could lead to unintended crashes in both system libraries and Spack-built libraries, since each could attempt to access unexpected, incompatible libraries. Taking inspiration from the Nix package manager [10, 11], Spack instead uses RPATHs [14]. RPATHs allow for embedding the location of shared libraries directly into a binary without environment modification. All Spack packages are installed in a user-defined prefix and all dependencies are embedded as RPATHs.

Unlike Nix, Spack supports *binary relocation*. Spack build caches can be installed in any location, including user home directories.

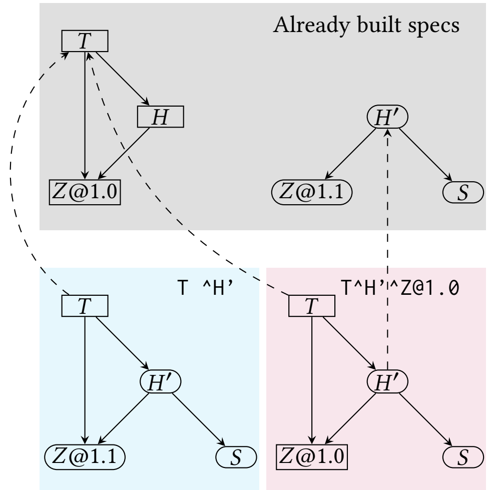

Figure 2: An example of how splicing combines spec-DAGs while maintaining build provenance. The specs in gray, `T ^H ^Z@1.0` (rectangular nodes) and `H' ^S ^Z@1.1` (rounded nodes), are already built. A spec conforming to `T ^H'` can be satisfied with a splice of the already built `H'` into already built `T` (blue background). A spec conforming to `T ^H' ^Z@1.0` can be satisfied by splicing `Z@1.0` back into the previous spec (red background). The build provenance for spliced specs are denoted with dashed lines.

The deployment locations of a binary package and its dependencies may *not* be the same at runtime as at build time. To ensure correct execution, Spack rewrites references to installation locations in installed files. It creates a mapping from binaries' original install locations to new locations, and it patches all occurrences of the old locations. For simple cases, where the install location is shorter than the build location, Spack uses simple patching logic. For more complex installs, it can use the `patchelf` tool to lengthen paths [9]. Spack can build binaries on a node's local filesystem, relocate them to a network file system, push them to a remote server for hosting reusable binaries, and install them again on a separate cluster.

4 SPLICING

Supporting ABI compatibility presents two key package management opportunities. The first is to avoid the unnecessary cascade of rebuilds for transitive dependent packages when swapping ABI-compatible dependencies. Similarly, deploying a package binary built with a reference library (like MPICH) onto systems with custom ABI-compatible implementations should not trigger rebuilds.

As discussed in section 2.2, updating a dependency in a source build can cause a series of package rebuilds for everything that directly and transitively depends on it even when the updated package is ABI-compatible with the version of the package that it is updating. Instead, only the updated package needs to be rebuilt and the RPATHs of all dependents updated to the new binary.

The deployment scenario is even simpler. Recall from section 1 the problem of deploying a package on a cluster using Cray MPICH when the package binary was built against a general implementation MPICH on a build server. Notably, Cray MPICH is usually a binary that exists only on the cluster. Assuming the general implementation conforms to the Cray MPICH ABI, deployment of the package binary should only require that it point to Cray MPICH.

These scenarios introduce two challenges for Spack:

(1) How should Spack represent binaries built against one implementation of an interface and deployed against another?
(2) How should developers specify ABI-compatibility information to Spack such that these solutions are automated?

In this section we present *splicing* as a solution to the first challenge and focus on its mechanics and representation. We also present a solution to the second challenge of automating splicing in section 5. Splicing is a model of describing package installations built against one shared library and deployed against another, while maintaining full build provenance.

4.1 The Mechanics of Splicing

We present the mechanics of splicing through a stripped down synthetic scenario, as depicted in fig. 2: There are two pre-compiled packages, `T` and `H'`, conforming to `T ^H ^Z@1.0` (rectangular nodes) and `H' ^S ^Z@1.1` (rounded nodes). The user is requesting a package installation conforming to `T ^H'`, and Spack knows that `H` and `Z@1.0` are ABI-compatible with `H'` and `Z@1.1`, respectively. Currently, the best Spack could do to satisfy the request would be to recompile `T` against `H'`; however, our goal is to satisfy the user's request without recompiling.

At the level of spec DAGs, the solution is as simple as replacing the `T` dependency on `H` with the full spec for `H'`, which we call splicing; however, there are a few subtleties to acknowledge. The first is the handling of the shared `Z` dependency. Simply copying in the full spec for `H'` would leave two copies of the `Z` dependency at different versions. Since there can only be one version of any dependency in the link-run graph for a spec, splicing must have a way to "break ties." In general, we consider splices to be *transitive*, namely that splicing a dependent spec also replaces all shared dependencies between the root and the spec being spliced in.

If the user instead requested `T ^H' ^Z@1.0` (*i.e.*, explicitly constraining `Z`), this can be solved with another splice. Splicing `Z` back into the the initial transitively spliced spec (blue background in fig. 2), would produce an *intransitive* splice (red background in fig. 2), where `Z@1.1` dependency for `H'` is replaced with `Z@1.0`, and the dependency for `T` is restored.

The second subtlety is related to the *build provenance* of spliced specs. Note that `T ^H' ^Z@1.1` *could* have been how the related binaries were built, and thus a naive implementation attempting to reproduce this spec would build it directly. Yet, this is not actually how the binaries corresponding to this spec were built. First, `T ^H ^Z@1.0` and `H' ^Z@1.1` were built, and then they were spliced together. In order to maintain reproducibility, Spack needs to distinguish spliced specs from non-spliced specs and should be able reproduce the whole process of how a spliced spec was constructed by building the initial specs and then splicing them together, rather than just building the resulting spec. Therefore, we augment spliced specs with a reference to the spec from which they are derived that we call the *build spec* (denoted by the dashed line in fig. 2).

The final subtlety is the handling of build dependencies (not pictured). Splicing is only a consideration for concrete specs which have already been built (*i.e.*, there is an existing binary to be relinked). On the other hand, recall that build dependencies need not be present *after* being used to build the spec's binary. Therefore, build dependencies are removed from the spec whose dependency has been spliced, since they are no longer relevant to the runtime representation. By keeping around the aforementioned build spec, removing these build dependencies from the spliced spec does not sacrifice reproducibility.

4.2 Patching Spliced Binaries

Recall from section 3.4 that all Spack requires for binary relocation is a mapping from the current installation paths of shared libraries to the desired relocation paths of those libraries. This machinery can easily be reused for patching binaries from spliced specs by generalizing *relocation* to *rewiring*. Instead of changing the paths of the same shared library to different locations, the shared library of the old node in the spec before it was spliced is patched to be the new shared library of the analogous node after it was spliced. This rewiring demonstrates the need for the build spec in a spliced spec, since it allows for Spack to determine the mapping between how the binary was initially installed to how it needs to be patched in order to conform to the spliced spec. Build specs also make it simple to determine whether a spec has been spliced at all, since only spliced specs will have an associated build spec.

5 AUTOMATIC SPLICING

In this section we describe the implementation of automatic splicing in Spack. The concretizer synthesizes spliced solutions from pre-installed specs when package developer-specified declarative constraints on splice validity are satisfied. As described in previous sections, this allows specs from a buildcache to be built using general implementations of common interfaces, like MPICH, and deployed without user intervention against specialized implementations, such as CrayMPICH. It also enables updating dependencies in a way that does not require *rebuilding the world*.

Several changes needed to be made to Spack. A new directive was added to the packaging language for expressing ABI-compatibility (section 5.2). The ASP encoding of pre-installed specs needed to be changed to allow for splicing (section 5.3). Finally, logic was added to the concretizer to synthesize and execute splices (section 5.4). Section 5.1 provides an introduction to the ASP encoding and techniques used in Spack's concretizer.[3]

5.1 ASP in the concretizer

Recall from section 3.3 that the ASP portion of Spack's concretizer takes four inputs: compiled ASP facts of the user's requested spec, compiled ASP facts from all package definition classes, an encoding of the concrete specs that it can reuse, and a logic program implementing Spack's concretization semantics. Specs are encoded through two relations: `node` and `attr`. The unary `node` relation simply takes the name of the package that it is encoding to indicate its presence. The higher-order, variable arity `attr` relation

[3]Most ASP examples are simplified to the core logic necessary to understand the approach in this work. Spack's full concretizer has a number of indirections and augmentations to support a diverse array of features that are orthogonal to splicing and binary reuse.

describes how various attributes are attached to nodes. The first argument of `attr` is always the name of the attribute, the second is the node to which it applies, and all other arguments depend on the particular attribute. By using a higher-order `attr` relation, rather than individual relations for each potential attribute, common patterns that relate to all of the potential attributes can be reused (such as `impose` later in this section). The following is the encoding of `example+bzip ^bzip2`:

```
node("example").
node("bzip2").
attr("depends_on",node("example"),node("bzip2"),"link-run").
attr("variant",node("example"),"bzip","True").
```

Notably, the spec encoded above is not yet concrete, since not all attributes have been set (*e.g.*, the versions of the packages). The logic program implementing the concretizer describes how to take user provided abstract specs and the compiled package definitions to set *all* attributes on the nodes. Attributes range not only over the six for concrete specs described in section 3.1, but also can describe various integrity properties of the whole spec DAG as it relates to a particular node. There are over twenty different attributes considered in the concretizer in all.

The directives from package definitions are compiled to analogous ASP facts describing constraints on the solution space. The following is an example of the facts related to versions for the package from fig. 1, declaring two unconditional versions for the package `1.1.0` and `1.0.0`:

```
pkg_fact("example",version_declared("1.1.0")).
pkg_fact("example",version_declared("1.0.0")).
```

The logic program then has the related rule for selecting versions which can be read as "choose exactly one version for each node from the declared versions.":

```
1 { attr("version",node(Name),Ver)
    : pkg_fact(Name, version_declared(Ver)
 } 1 :-
  attr("node",node(Name)).
```

There are analogous corresponding package facts and concretizer rules for each associated attribute for concrete spec (*e.g.*, variant values, dependencies, *etc.*).

5.1.1 Handling conditional constraints. Many package directives have a `when` argument for expressing conditional constraints. These require a little more machinery to express in ASP. Each condition is given a unique identifier, the requirements for the condition to be satisfied, and the constraint imposed on the resulting models if those conditions hold. Below is the compiled representation of the conditional dependency from line 11 of the package in fig. 1. It declares a condition with unique identifier `"x153"`, with conditions that there is a node `"example"` with variant `"bzip"` set to `"True"`. The condition then imposes the constraint that the `"example"` node also depends on `"bzip2"`.

```
condition("x153").
condition_requirement("x153","node","example").
condition_requirement("x153","variant","example","bzip","True").
imposed_constraint("x153","depends_on","example","bzip2")
```

The `condition_requirement` follows a similar pattern to `attr`, in that it is a higher-order, variable arity relation to allow for uniform handling of the various requirements.

In order to attach attributes to nodes conditionally (*i.e.*, when a `when` spec is satisfied), the concretizer includes a set of rules for conditionally imposed constraints on nodes. These rules hinge on two relations, `impose` and `imposed_constraint`. The latter is a variable-arity relation taking advantage of the higher order encoding of `attr` indexed by a unique identifier, while the former controls constraints associated with that identifier that are imposed on a particular node. What follows is an example of how these facts are used in the logic program for the concretizer:

```
attr(Attr, node(Name), Arg1) :-
  impose(ID, node(Name)),
  imposed_constraint(ID, Name, Arg), node(Name).
attr(Attr, node(Name), Arg1, Arg2) :-
  impose(ID, node(Name)),
  imposed_constraint(ID, Name, Arg1, Arg2), node(Name).
```

Note that there are similar rules at every potential arity of the `attr` relation. `imposed_constraint` is used to encode various package facts, especially for reusable specs and `impose` is derived when the conditions associated with the `ID` are present. In particular to this work, the `ID` considered will be that of the hash of a reusable spec.

5.1.2 Reusing concrete specs. In order to reuse concrete specs, the concretizer must be aware of installed specs indexed by their unique hash) and which constraints would be imposed in their reuse. This is encoded directly by providing an `installed_hash` relation that ties package names to the hashes of concrete specs and `imposed_constraint` facts for all of the attributes of the concrete spec. What follows is the encoding of the concrete spec from section 3.3:

```
installed_hash("example", "abcdef1234").
imposed_constraint("abcd1234","version","example","1.1.0").
imposed_constraint("abcd1234","variant","example","bzip","True").
imposed_constraint("abcd1234","node_os","example","centos8").
imposed_constraint("abcd1234","node_target","example","skylake").
imposed_constraint("abcd1234","depends_on","example","bzip2").
imposed_constraint("abcd1234","hash","bzip2","1234abcd").
```

We will update this encoding in section 5.3 with some indirection to better facilitate splicing.

In order to select reusable specs, the concretizer may select at most one spec for every node with a potential candidate (*i.e.*, the `installed_hash` fact is present for a node with that name). If chosen, the concretizer imposes all constraints associated with that concrete spec. Any node which is not selected for reuse is then marked to be built instead. Then, the highest optimization objective in Spack's concretizer is to reuse as many specs as possible, or equivalently to minimize the number of specs needing to be built:

```
{ attr("hash", node(Name), Hash): installed_hash(Name, Hash) } 1
  :- node(Name).
impose(Hash, Node) :- attr("hash", Node, Hash), Node.
build(Node) :- Node, not attr("hash", Node, Hash).
#minimize { 100, PackageNode : build(PackageNode) }
```

5.2 Augmenting Spack's Package Language

5.2.1 Representation in the packaging DSL. We add a single directive to Spack's packaging language for specifying ABI-compatibility relationships between packages, `can_splice`. The bottom of fig. 1 demonstrates `can_splice` directives for our example package. The first spec argument describes the constraints on the target of the splice (*i.e.*, the spec that would be replaced by the splice) and the `when` spec argument describes the constraints on the package for the splice to be valid. Both arguments support the full spec syntax. Note that splices can be specified between two versions of the same package as well as different packages. `can_splice` is a natural extension to the packaging language since it utilizes the same API as most other Spack directives.

This directive inverts the dependency structure of Spack when considering binary distributions of packages. While the `depends_on` relationship is given by the *depending* package, `can_splice` allows *dependent* packages to state when it would not be ABI-breaking for them to be swapped for another package. One subtlety to note is that packages state which specs they *can replace* rather than by which they *can be replaced*. This is an acknowledgment of the fact that developers of a common ABI, like that of MPICH, will not necessarily be aware of every ABI-compatible replacement, but developers of these replacements will know that their package is ABI compatible.

5.2.2 Encoding in ASP. Unlike other directives in Spack, rather than being encoded as facts to describe the constraints imposed by the `can_splice` directive, we choose to encode the constraints from `can_splice` as a specialized rule. Figure 4a demonstrates our ASP encoding for the second `can_splice` directive in fig. 1. It is a single rule that derives the fact `can_splice`. This fact represents that the package node in its first argument can replace a reusable spec with the name of the second argument and the hash of the third argument, and can only be derived if there is a package node satisfying the `when` spec and a pre-built spec satisfying the target. It is a near-direct translation of the `can_splice` directive to ASP. Being able to match node attributes to pre-built spec attributes is one of factors that motivated the change in encoding of reusable specs to use `hash_attr`.

5.3 Changing the Encoding of Reusable Specs

Recall from section 5.1.2 that reusable specs were encoded by directly as `imposed_constraint` facts for every attribute of the concrete spec, indexed by its hash. Since splicing changes the dependencies of a concrete spec, the concretizer needs to be able to change which dependencies are imposed by the spliced spec. Yet, in the previous encoding, all constraints from the original spec are imposed immediately, and ASP does not have a mechanism for removing derived facts from a model. Therefore, we changed the encoding of pre-installed specs to add a layer of indirection between the encoding of the attributes of reusable specs, and when those attributes are actually imposed. Figure 3a demonstrates how we introduce this indirection through `hash_attr` facts and fig. 3b shows how we recover the semantics from the previous encoding. Note that the initial imposition of `"hash"` and `"depends_on"` depends on the presence of `can_splice`. These attributes are where splices can be introduced and this encoding allows the concretizer to make decisions about imposition and reuse of either the original dependency or a spliced dependency. Without this `can_splice`, there are no valid splices and thus the concretizer defaults to its previous behavior of always reusing the dependencies of any pre-built spec whenever it reuses the spec itself.

```
installed_hash("example", "abcdef1234").
hash_attr("abcd1234","version","example","1.1.0").
hash_attr("abcd1234","variant","example","bzip","True").
hash_attr("abcd1234","node_os","example","centos8").
hash_attr("abcd1234","node_target","example","skylake").
hash_attr("abcd1234","depends_on","example","bzip2").
hash_attr("abcd1234","hash","bzip2","1234abcd").
```

(a) The new encoding of reusable specs.

```
imposed_constraint(Hash, Attr, Name) :- hash_attr(Hash, Attr, Name).
imposed_constraint(Hash, Attr, Name, Arg1) :-
  hash_attr(Hash, Attr, Name, Arg1),
  Arg1 != "hash".
imposed_constraint(Hash, Attr, PackageName, Arg1, Arg2) :-
  hash_attr(Hash, Attr, PackageName, Arg1, Arg2),
  Attr != "depends_on".
imposed_constraint(ParentHash,"hash",ChildName,ChildHash) :-
  hash_attr(ParentHash,"hash",ChildName,ChildHash),
  ChildHash!=ParentHash,
  not can_splice(node(Name),ChildName,ChildHash).
imposed_constraint(ParentHash,"depends_on",ParentName,ChildName) :-
  hash_attr(ParentHash,"depends_on",ParentName,ChildName),
  not can_splice(node(Name),ChildName,ChildHash).
```

(b) Rules that recover the `imposed_constraint` facts from the original encoding of reusable specs.

Figure 3: The new encoding of reusable specs introduces indirection into the of `imposed_constraint`, this indirection is key for splicing to allow for splicing to change the dependencies of reusable specs.

```
% can_splice("example-ng@2.3.2+compat",when="@1.1.0+compat)
can_splice(node("example"),"example-ng",Hash) :-
  installed_hash("example-ng",Hash),
  attr("node",node("example")),
  hash_attr(Hash,"version","example-ng","2.3.2"),
  attr("version",node("example"),"1.1.0"),
  hash_attr(Hash,"variant","example-ng","compat","True"),
  attr("variant",node("example"),"compat","True").
```

(a) Compiled ASP encoding of the `can_splice` directive included as a comment on the first line of the fragment

```
1  { imposed_constraint(ParentHash,"hash",DepName,DepHash) } :-
2    hash_attr(ParentHash,"hash",DepName,DepHash),
3    can_splice(node(SpliceName),DepName,DepHash).
4  attr("splice",ParentNode,node(SpliceName),DepName,DepHash) :-
5    attr("hash",ParentNode,ParentHash),
6    hash_attr(ParentHash,"hash",DepName,DepHash),
7    can_splice(node(SpliceName),DepName,DepHash),
8    not imposed_constraint(ParentHash,"hash",DepName,DepHash).
9  imposed_constraint(Hash,"depends_on",ParentName,SpliceName,Type) :-
10   hash_attr(Hash,"depends_on",ParentName,DepName,Type),
11   hash_attr(Hash,"hash",DepName,DepHash),
12   attr("splice",node(ParentName),node(SpliceName),DepName,DepHash).
```

(b) The core logic implementing splicing. This fragment selects whether to execute a splice based on the presence of the `can_splice` fact, and then imposes the new spliced dependency instead of the pre-installed spec's original dependency.

Figure 4: The splicing implementation in ASP portion of Spack's concretizer.

5.4 Splicing in the Concretizer

Figure 4b demonstrates the core fragment of the concretizer implementing splicing. The rules can be read top to bottom as implementing the fairly simple decision procedure for splicing. The rules beginning on lines 1 and 4 describe an exclusive-or condition for how to impose the dependencies of the spec that is being chosen for reuse provided it has a potential candidate for a splice. When reusing a concrete spec, the concretizer may impose its original dependency (the choice on line 1). If it does not (the condition on line 8), it must instead splice in a compatible dependency (the conclusion on line 4). If it decides to splice in a new dependency (the condition on line 12), that new dependency is imposed instead of the original one (the conclusion on line 9).

Perhaps it is surprising how little logic is required to implement the core splicing in the concretizer's logic program. We owe this elegance to the change in encoding of reusable concrete specs. The small indirection added through `hash_attr` facts provides the perfect hook for changing the dependency when splicing. By refactoring machinery already present in the concretizer (*i.e.*, `impose` and `imposed_constraint`), we are still able to take advantage of the logic that maintains the integrity of the whole spec-DAG (*e.g.*, ensuring compatible microarchitectures among all specs). After making this encoding change, Spack core developers have found subsequent uses in the concretizer for indirection in imposing concrete specs[4].

Ultimately, the concretizer produces a mapping between prebuilt specs (*i.e.*, nodes with the `"hash"` attribute) and their dependencies that will be replaced by splicing (*i.e.*, nodes with the `"splice"` attribute). This mapping can be used directly not only to build the spliced specs in a straightforward manner but also in the rewiring of spliced specs, as described in section 4.2. Finally, the splicing logic can be conditionally loaded, and thus is a fully transparent opt-in feature. We validate that our encoding change for pre-installed specs incurs minimal overhead in the next section.

6 PERFORMANCE RESULTS

We have fully implemented automatic splicing in Spack's concretizer and aim to answer 4 questions about its performance:

- RQ1. Do the changes to the encoding of prebuilt packages in the concretizer introduce bugs or performance regressions?
- RQ2. Does the concretizer produce spliced solutions when necessary?
- RQ3. What is the overhead of considering spliced solutions?
- RQ4. How does automatic splicing performance scale in relation to the number of candidate splices?

6.1 Experimental Setup

6.1.1 System Setup. All performances runs were executed on a single node with 96 Intel Xenon Gold 6342 (Icelake) processors and 1TB of memory running Ubuntu 22.04.

6.1.2 The RADIUSS stack and the mpiabi mock package. In each of our experiments we consider the concretization of specs from the RADIUSS software stack. RADIUSS is an open-source collection of HPC infrastructure (*e.g.*, Flux and LvArray), portability tools (*e.g.*, RAJA and CHAI), data management and visualization (*e.g.*, glvis and hatchet), and simulation applications (*e.g.*, ascent and sundials) created by LLNL in order to provide a unified foundation for developing HPC applications. Concretizing RADIUSS packages is a common use-case for Spack and the packages vary in both overall dependency structure and number of dependencies. Furthermore, many of the packages in the RADIUSS stack have a virtual dependency on MPI which serves as our splice target. For a splice candidate, we created a mock package, mpiabi, based on mvapich, with a single version and the ability to splice into `mpich@3.4.3`. In section 6.4, we introduce many copies of this package in order to assess the scaling factor of the number of splice candidates.

6.1.3 The local and public buildcaches. Splicing requires the existence of pre-concretized specs so we use *buildcaches* of specs. Buildcaches are Spack's way of providing reusable binaries of prebuilt specs that map concrete specs to their build targets (*e.g.*, shared libraries, executables, *etc.*) and whose contents can be reasoned about by the concretizer when deciding which specs can be reused. Our experiments rely on a local and a public buildcache. The local buildcache consists of only specs from the RADIUSS stack (and transitive dependencies), allowing us to construct a controlled environment to evaluate the correctness of our implementation, containing ~200 specs. The public buildcache, which is generated by Spack [5] as a continuous integration task, contains over 20,000 prebuilt specs that include multiple configurations of the RADIUSS stack, which provides a use-case more in line with that of the average Spack user.

6.1.4 Concretization Objectives and Configurations. We consider the concretization of each of the 32 individual specs in the RADIUSS stack and there are three axes for configuration in our experiments. The first axis is the version of Spack under consideration. We compare our implementation, hence *splice Spack*, to the last commit on the main Spack Github repository prior to the introduction of automatic splicing, hence *old Spack*. [6]. The second axis is the number of reusable specs the concretizer considers when producing solutions. This can be either the local RADIUSS buildcache or the public buildcache. The third axis is whether automatic splicing is enabled when testing that implementation. Recall from section 5 that *splice Spack* allows for users to conditionally enable the concretizer to consider automatic splices; however, disabling automatic splicing *does not* revert to the previous encoding of reusable specs. Therefore, experiments where the feature is disabled are to evaluate the performance of the new encoding of reusable specs without the added expressivity of allowing splicing. Enabling splicing allows the concretizer to consider solutions that involve splicing, but does not explicitly constrain it to only consider solutions with splicing. In order to address the variability inherent in heuristic nature of Spack's dependency resolution engine, we evaluate each concretization objective in each configuration 30 times.

6.2 Determining the Impact of the Changed Encoding for Reusable Specs

In order to answer RQ1, we consider whether our changes to the encoding of reusable specs and strategy for reuse introduce regressions into the concretizer. To accomplish this, we concretize each objective using both *old Spack* and *splice Spack* against both buildcache configurations. We consider only the case where the

[4] https://github.com/spack/spack/pull/45189

[5] www.cache.spack.io

[6] Git commit hash for *old Spack*: ad518d975c711c04bdc013363d8fc33a212e9194, git commit hash for *splice Spack*: 9f7cff1780d1e3e97cf957d686966a74d3840af6. Spack code is available at https://github.com/spack/spack

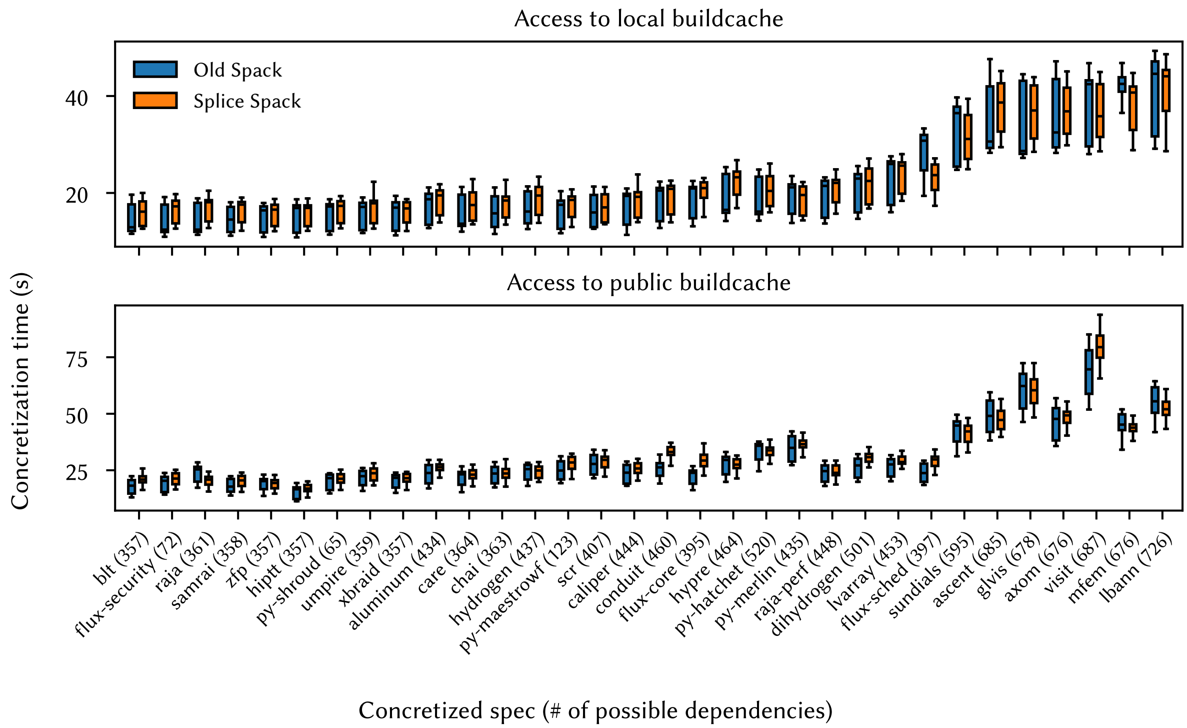

Figure 5: A comparison of the performance of *old Spack* and *splice Spack* with respect to their encoding of reusable specs over 30 runs.

automatic splicing feature is disabled, such that the only difference between the two implementations is the encoding of reusable specs.

Figure 5 shows the distribution over 30 runs for concretizing the entire RADIUSS stack in the four possible configurations. Across all specs in the RADIUSS stack, we see an 4.7 percent increase in average concretization time with access to the local buildcache from our encoding change, and a 7.1 percent increase with access to the public buildcache. Therefore, we can see that the change to the encoding of resuseable specs only introduces neglible overhead into the performance of the concretizer.

6.3 Evaluating the Correctness and Performance of Automatic Splicing

In order to answer RQ2 and RQ3, we consider the subset of specs in RADIUSS that have a virtual dependency on MPI. We concretize these MPI-dependent specs separately and jointly with an explicit dependency on either `mpich` in the case of *old Spack*, or our mock MPI package (`mpiabi`) in the case of *splice Spack* using both buildcache configurations. We ensure that *splice Spack* produces spliced solutions when possible. We also include `py-shroud` in order to evaluate the overhead of enabling splicing on specs which cannot be spliced. Note that *splice Spack* is solving a harder problem than *old Spack*, since it now must consider splicing rather than simple reuse, and thus one should expect a increase in concretization time.

Figure 6 shows the distribution over 30 runs for concretizing `py-shroud` and the MPI-dependent specs in the RADIUSS stack in the four possible configurations. Across all MPI-dependent specs in the RADIUSS stack, we see a 17.1 percent increase in concretization time with access to the local buildcache from our encoding change, and a 153 percent increase with access to the public buildcache. We see virtually no difference in concretization time for `py-shroud` from our extension. Recall that the public buildcache is roughly 2 orders of magnitude larger than the local buildcace (~200 specs vs. ~20,000 specs).

Thus, while enabling splicing introduces a potentially two minute increase to concretization time in the case of specs like `visit` and `glvis` against the large public buildcache, every spliced solution could save potential hours of time spent building software. Furthermore, since splicing can be conditionally enabled, users can easily opt out if they would not benefit from splicing and would prefer faster concretization.

6.4 Scaling the Number of Spliceable Packages

The previous experiments consider splicing a single package with a single `can_splice` directive targeting `mpich@3.4.3`; however, in more realistic scenarios there may be many potential splice candidates for a given target. Therefore, in order to answer RQ4, we will consider the same concretization goals as section 6.3, but instead of varying the version of Spack, we will instead scale the number of potential splice candidates. We accomplish this by first creating 100 of copies of `mpiabi` differing only in name. We then concretize with the same objectives as the previous experiment using only the local buildcache while giving the concretizer access to increasingly large subsets of the 100 mock packages. We also

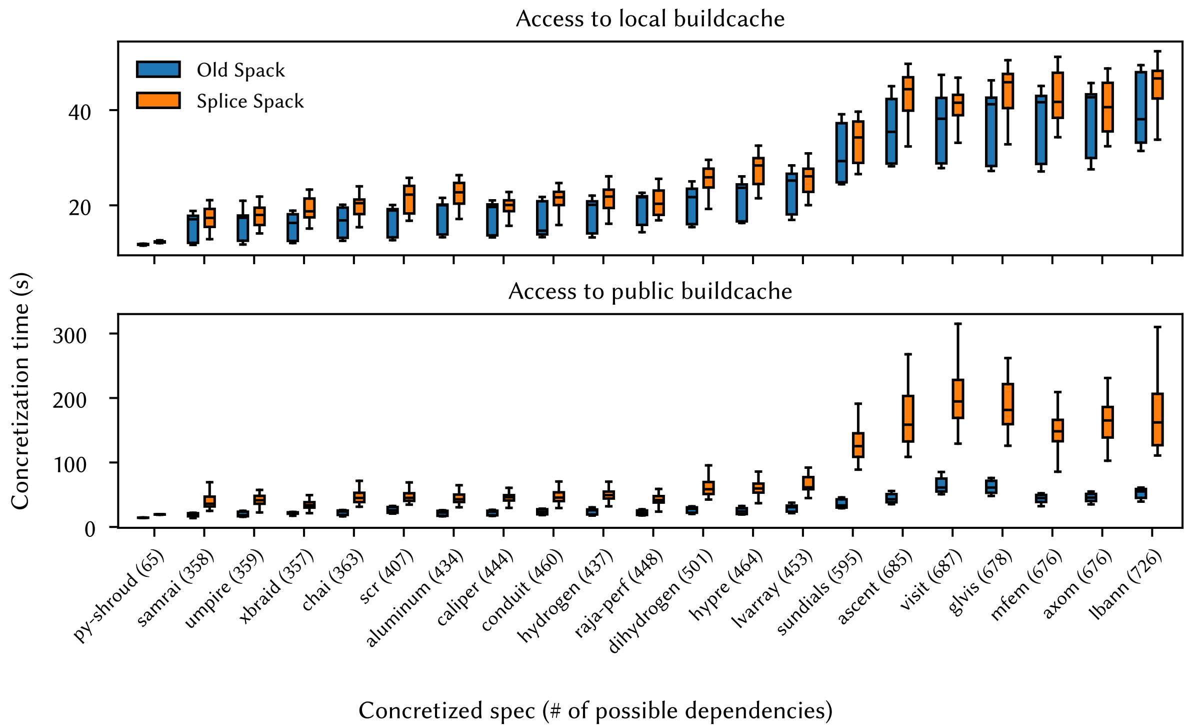

Figure 6: A comparison of the performance of a non-spliced solution for *old Spack* with *splice Spack* considering spliced solutions over 30 runs.

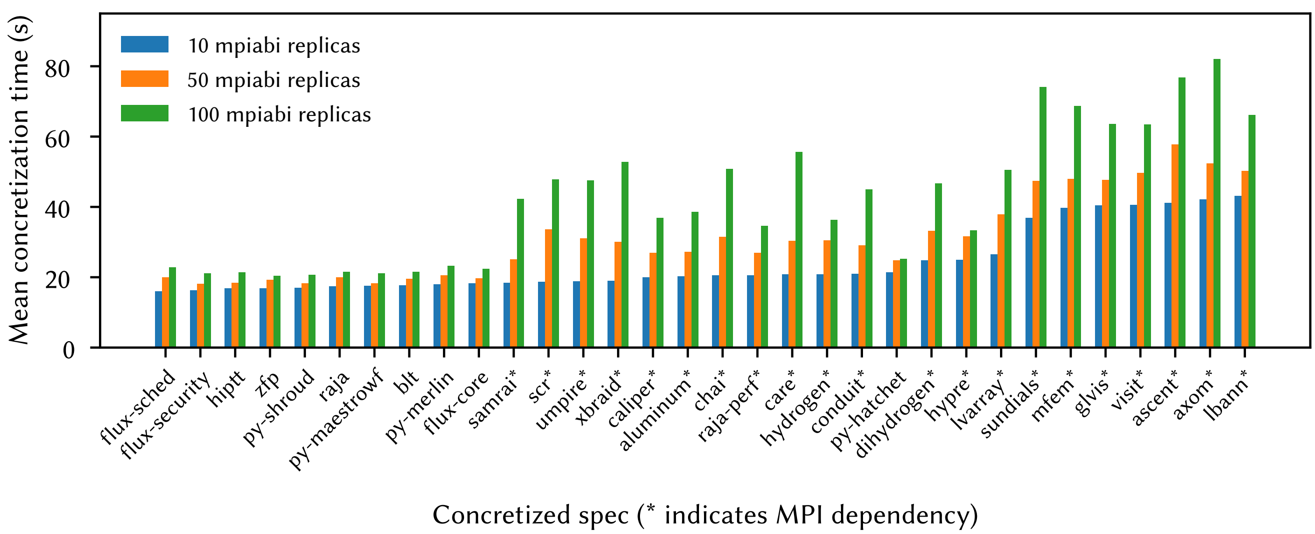

Figure 7: An evaluation of the scaling behavior of *splice Spack* when the number of potential splice candidates is increased.

require that concretized specs *do not* depend on `mpich`, but do not constrain which of the replicas the concretizer chooses.

Figure 7 shows how the average concretization time scales for concretization of the RADIUSS stack. Across all MPI-dependent packages, the average percent increase in concretization time between 10 and 100 replicas is 74.2. Again, this increase on the order of minutes is inconsequential compared the potential hours saved by not having to rebuild packages. Note that there is very little overhead in scaling the number of replicas for specs which do not have an MPI dependency.

7 RELATED WORK

Splicing generalizes approaches to avoid rebuilds in source-based package management by allowing binary substitution across ABI-compatible implementations. Related systems either avoid this problem through rigid policies or handle it with limited, use-case-specific tooling. Systems like `rpm`, `apt`, `yum`, `dnf`, and other binary package managers, regularly resolve dependencies without rebuilding their dependents. Typical Linux distributions upgrade packages in-place, and when a dependency binary is installed, its dependents are rebuilt. Traditional distributions *trust* the maintainers to ensure that ABI compatibility is maintained. Some, like Red Hat, use tooling like `libabigail` to ensure that new packages adhere to the ABI of existing packages. Fundamentally, all of these solutions rely on the maintenance process to ensure ABI compatibility. Moreover, these systems do not reason deeply about ABI–they typically only manage version constraints, and do not reason about flags, build options, or microarchitectures as our conditional `can_splice` solution allows.

So-called functional package managers like Nix [10, 11] and Guix [4] ensure that ABI compatibility is preserved using a convention similar to Spack. They identify packages based on their own configuration *and* that of all their dependencies, and they traditionally force full rebuilds if any dependencies change. Nix mitigates this issue by providing a public (non-relocatable) binary cache full of package substitutes. Again, maintainers ensure that rebuilds happen and packagers can easily install once all rebuilds happen.

Guix has implemented a solution for the cascading rebuild problem most similar to our own; their variant is called *grafts* [1]. Their solution is intended for security updates and one-time ABI replacements. Splicing lets package authors declare ABI compatibility using spec constraints, enabling the solver to select compatible replacements based on context. Unlike grafts, these declarations live with the replacing package, not the replaced one. Guix allows developers to specify replacements for their own package that avoid cascading rebuilds [2, 3]. This solution differs from ours in that the *replaced* package specifies what can replace it. This effectively means that the replaced package must *know* about possible replacements, and that only one replacement can be specified at any given time. This makes sense for Guix's use case of security updates. If a CVE is discovered, a grafted package is replaced with a "fixed" version. However, this implementation is unsuitable for implementations like MPI, where there may be *many* possible ABI-compatible replacements that may be different depending on the host platform and the choices of the developer maintaining the software stack. We could not express in Guix, for example, that Cray MPICH, mvapich, and intel-mpi are all suitable replacements for mpich. Further, Guix has no solver, so there is no reasoning on the part of the package manager about ABI constraints – the maintainers must ensure that any specified replacement is ABI-compatible with the replaced package. Splicing instead allows for an ecosystem in which new replacements can be easily added, users can choose which replacement they use, and the solver ultimately ensures the safety of user choices.

8 CONCLUSIONS AND FUTURE WORK

There is a fundamental tension between source-based and binary-based package management trading off configurability for installation speed. This tension arises from a lack of explicitly modeling ABI-compatibility in source-based package managers, while concerns of ABI-compatibility limit configurability in binary-based package managers.

In this paper, we provide an extension to the source-based Spack package manager that allows for explicitly stating and reasoning about ABI-compatibility among packages and automatically generating solutions with a minimal number of builds by reusing and relinking ABI-compatible packages when possible. Our extension incurs minimal overhead, while greatly extending the modeling capability of Spack. We have also demonstrated that our approach scales well in the presence of many opportunities for splicing. Our hope is that this extension will increase the efficiency of Spack users, and provide a testing bed for future research on ABI compatibility.

Currently, ABI compatibility must be specified by package developers manually adding `can_splice` to their package classes. The one-time effort of modeling a package's ABI can provide massive efficiency gains for all downstream users of a package. In the future, we will develop methods for automating ABI discovery for the Spack ecosystem in order to reduce developer burden.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science under Award Number DESC0025613 and Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

REFERENCES

[1] Ludovic Courtès. 2016. *Security Updates.* https://guix.gnu.org/manual/devel/en/html_node/Security-Updates.html
[2] Ludovic Courtès. 2016. *Timely delivery of security updates.* https://guix.gnu.org/blog/2016/timely-delivery-of-security-updates/
[3] Ludovic Courtès. 2020. *Grafts, continued.* https://guix.gnu.org/blog/2020/grafts-continued/
[4] Ludovic Courtès and Ricardo Wurmus. 2015. Reproducible and User-Controlled Software Environments in HPC with Guix. In *2nd International Workshop on Reproducibility in Parallel Computing (RepPar)*. Vienne, Austria. https://hal.inria.fr/hal-01161771
[5] Russ Cox. 2016. *Version SAT: Dependency hell is NP-complete. But maybe we can climb out.* https://research.swtch.com/version-sat
[6] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. https://doi.org/10.1145/368273.368557

[7] Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. 7, 3 (July 1960), 201–215. https://doi.org/10.1145/321033.321034

[8] Roberto Di Cosmo. 2006. EDOS deliverable WP2-D2. 2: Report on Formal Management of Software Dependencies. (2006).

[9] Eecol Dolstra. [n. d.]. *patchelf*. https://github.com/NixOS/patchelf

[10] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th USENIX Conference on System Administration* (Atlanta, GA) *(LISA '04)*. USENIX Association, USA, 79–92.

[11] Eelco Dolstra and Andres Löh. 2008. NixOS: a purely functional Linux distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 367–378. https://doi.org/10.1145/1411204.1411255

[12] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 97–104.

[13] Todd Gamblin, Massimiliano Culpo, Gregory Becker, and Sergei Shudler. 2022. Using Answer Set Programming for HPC Dependency Solving. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41404.2022.00040

[14] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R De Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[15] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. 2011. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.* 24, 2 (April 2011), 107–124.

[16] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* 22, 6 (1996), 789–828. https://doi.org/10.1016/0167-8191(96)00024-5

[17] Jeff Hammond, Lisandro Dalcin, Erik Schnetter, Marc PéRache, Jean-Baptiste Besnard, Jed Brown, Gonzalo Brito Gadeschi, Simon Byrne, Joseph Schuchart, and Hui Zhou. 2023. MPI Application Binary Interface Standardization. In *Proceedings of the 30th European MPI Users' Group Meeting* (Bristol, United Kingdom) *(EuroMPI '23)*. Association for Computing Machinery, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/3615318.3615319

[18] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference* (Las Vegas, Nevada, USA) *(DAC '01)*. Association for Computing Machinery, New York, NY, USA, 530–535. https://doi.org/10.1145/378239.379017

[19] Hewlett Packard Enterprise Development. 2025 (accessed April 7, 2025). *Cray MPICH Documentation*. https://cpe.ext.hpe.com/docs/24.03/mpt/mpich

[20] The Trilinos Project Team. [n. d.]. *The Trilinos Project Website*.