archspec: A library for detecting, labeling, and reasoning about microarchitectures

Massimiliano Culpo*, Gregory Becker†, Carlos Eduardo Arango Gutierrez‡, Kenneth Hoste§, Todd Gamblin†

*NP-complete S.r.l. — Mantova, Italy
massimiliano.culpo@gmail.com
†Lawrence Livermore National Laboratory — Livermore, CA, USA
{becker33,tgamblin}@llnl.gov
‡Red Hat and Universidad del Valle — Cali, Colombia
carlos.arango.gutierrez@correounivalle.edu.co
§HPC-UGent — Ghent University, Ghent, Belgium
kenneth.hoste@ugent.be

***Abstract*—Optimizing scientific code for specific microarchitectures is critical for performance, as each new processor generation supports new, specialized vector instructions. There is a lack of support for this in package managers and container ecosystems however, and users often settle for generic, less optimized binaries because they run on a wide range of systems and are easy to install. This comes at a considerable cost in performance. In this paper we introduce archspec, a library for reasoning about processor microarchitectures. We present the design and capabilities of archspec, which include detecting and labeling of microarchitectures, reasoning about microarchitectures and comparing them for compatibility, and determining the compiler flags that should be used to compile software for a specific microarchitecture. We demonstrate the benefits that archspec brings by discussing several use cases including package management, optimized software stacks, and multi-architecture container orchestration.**

1. Introduction

With Moore's law waning, there has been an explosion of new processor designs aimed at extracting every last bit of performance out of modern numerical workloads. Intel has added a host of 512-bit vector instructions (AVX-512) to its processor line, AMD processors heavily utilize 256-bit AVX2 instructions, and ARM has introduced a set of so-called Scalable Vector Extensions (SVE) for handling vectors up to 2048 bits wide. In addition to vector instructions, specialized instructions for cryptography, persistent memory, and other features are also beginning to appear.

While the core Instruction Set Architecture (ISA) (e.g., `x86_64` or `aarch64`) typically remains stable, new extensions can emerge with each new design or *microarchitecture*, and it is becoming increasingly difficult to track which processors support which extensions. To get the best performance, we *must* optimize for specific microarchitectures, but there is a fundamental tension between optimization and portability. The more we optimize for a specific machine, the less likely the code is to run on another (especially older) model. Those with newer processors benefit, but shipping optimized binaries restricts the set of potential users.

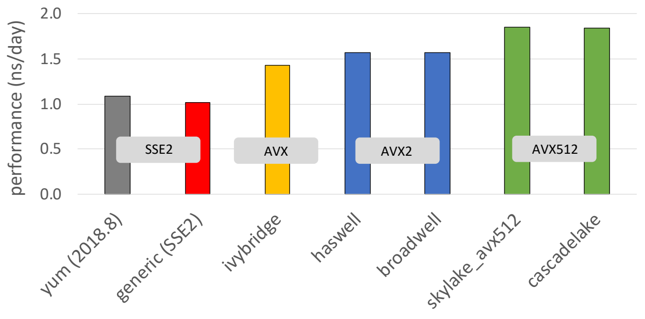

Figure 1. Performance of GROMACS 2020.1 built for different generations of CPUs.[2] Vertical axis shows performance expressed in `ns/day`, a GROMACS-specific measure of simulation speed (higher is better).

Tooling around packaging and containers has not kept up with hardware. The system packages that underlie most Linux distributions and container images are nearly always built without optimization in mind [1]. Binaries are built without any vector extensions, so that they will run almost everywhere. Users who want to take advantage of the newest hardware must either build the software themselves, or know which specific, optimized packages to install. The average user knows very little about the microarchitecture or supported vector instructions of their machine, and all too frequently users end up installing unoptimized software when optimized builds are available. For optimized binaries to be widely accessible, they must be easy to distribute.

2. GROMACS version 2020.1 was run on a dual-socket system with two Intel Xeon Gold 6420 processors (36 cores in total), using Test Case B from the PRACE Unified European Applications Benchmark Suite [2]. GROMACS was installed using EasyBuild [3] version 4.3.0 and the `foss/2020a` compiler toolchain (including GCC 9.3.0 and FFTW 3.3.8). For each installation the entire stack was built on an appropriate system using `-march=native`, and then run on the Intel Xeon Gold 6420 test system. For GROMACS the appropriate `-DGMX_SIMD` configuration option was used to compile it for a specific microarchitecture (for example `-DGMX_SIMD=AVX_512`).

The potential performance gains of optimization are quite significant. Figure 1 shows the performance of different GROMACS [4], [5] builds on a dual-socket Intel Xeon Gold 6240 system (Intel Cascade Lake microarchitecture). A generically optimized GROMACS installation that runs on any system supporting SSE2 is significantly slower than installations that exploit more recent SIMD instructions. Using AVX and AVX2 results in speedups of about 32% and 44% respectively. Compiling GROMACS for architectures that can exploit the AVX-512 instructions supported by the Intel Cascade Lake microarchitecture gives an additional 18% performance improvement relative to using AVX2 instructions, with a speedup of about 70% compared to a generic GROMACS installation with only SSE2.

The fundamental issue is that packaging tools do not know how to *reason* about compatibility. Most container tools can select an image for the base ISA, e.g., `x86_64`, but they do not optimize for specific microarchitectures (e.g., Cascade Lake or Skylake) on top of it. Similarly, in package managers like RPM a user can request the AVX-512 version of a package, usually by specifying a particular RPM name, but the user must know that they need it. There are many system tools that can tell *what* type of chip a host is, but the information they provide is often too detailed for humans to reason about. For example, the Linux `/proc/cpuinfo` filesystem reports a model number like `Intel(R) Xeon(R) CPU E5-2695 v4`, but it does not tell us that this is one of many models supporting the Broadwell microarchitecture.

To enable optimization-aware tooling, we have developed a library called `archspec`, which can detect, label, and reason about microarchitectures and their compatibility. A user can ask for the microarchitecture of the current machine and compare it to a label on a binary to determine whether they are compatible. Users can ask whether a particular microarchitecture supports certain features, and they can ask what flags to use for a particular compiler to build a binary specifically for a microarchitecture. We have designed `archspec` to be easy to contribute to—its database of processor information is a simple JSON file, and it is easy to write bindings for different languages using this file. We have so far implemented complete bindings for Python and experimental bindings in Go. The main contributions of this work are:

1) a conceptual framework for reasoning about compatibility as a directed acyclic graph;
2) a file format and schema for describing microarchitecture information;
3) an implementation of our framework in the `archspec` library; and
4) three detailed use cases showing how `archspec` can be used in practice.

In the remainder of this paper, we describe the concepts behind `archspec`, its implementation, and we describe several real-world use cases for the tool. Our aim is for `archspec` to be used in packaging and container tools, so that they may easily distribute and use optimized binaries.

2. The `archspec` library

In this section we discuss the design and capabilities of `archspec`. Its core feature is the ability to detect, label and compare microarchitectures at the granularity with which humans reason about them. We have chosen a set of intuitive microarchitecture labels, e.g., `skylake` or `thunderx2`, based on their commonly used names. We chose this level of granularity (as opposed to specific *models* of the same microarchitecture) in order to model processors at the granularity that their instruction set varies. This is what determines compatibility. We can say that a binary is compatible with `haswell` and will run on any later Intel chip, and all major compilers have flags that set the allowed instructions at this granularity. This lets us optimize for specific processor designs *and* reason about compatibility, whereas if we had picked a finer granularity like the specific processor *model*, there would be many models with essentially the same extended ISA. As such, it is coarser than model-specific details that one can collect from system tools but more fine-grained than the simple distinction between ISA families, like `x86_64` or `aarch64`.

2.1. The Microarchitecture Database

At a high level, `archspec` is composed of two parts. All the knowledge of microarchitecture names, features, compiler support and compiler flags is stored in a JSON file, which we call the *database*. On top of this static information, `archspec` provides language bindings (thus far, Python and Go) with logic to detect, query and manipulate microarchitecture objects. We chose this architecture because it is easy to contribute to—adding new microarchitectures requires only a new entry in the JSON database—and it allows us to easily write new language bindings on top of the JSON database file. JSON support is widely available across many languages, and, we plan to implement more language bindings in the future.

The most important information contained in the database is the dictionary of known microarchitectures. Each entry in the dictionary maps a unique label to corresponding information on:

- The closest compatible microarchitectures
- The vendor of the microarchitecture
- The instruction sets available
- The optimization support provided by compilers

The granularity of the labels used in `archspec` follows closely that of the "machine type" employed by compilers to emit processor-specific machine instructions. In fact, resources like the GCC documentation [6] have been of primary importance to gather information for the database. The actual labels though might differ from the ones used by GCC, since names for the microarchitectures have been selected to be human readable. For instance, in `archspec` we refer to the `steamroller` microarchitecture as opposed to `bdver3` in GCC.

```
{
  "broadwell": {
    "from": ["haswell"],
    "vendor": "GenuineIntel",
    "features": [
      "sse",
      "sse2",
      "...",
      "avx2"
    ],
    "compilers": {
      "gcc": [
        {
          "versions": "4.9:",
          "flags": "-march={name} -mtune={name}"
        }
      ]
    }
  }
}
```

Figure 2. An example record from archspec's JSON database.

An entry for the broadwell microarchitecture is shown in Figure 2. The from field tells us the closest compatible labels and allows us to reconstruct a Directed Acyclic Graph (DAG) encoding binary compatibility. We can determine whether one microarchitecture is compatible with another by examining its ancestors—the compatibility relationship is transitive and one-directional. The features list contains the ISA extensions implemented by a given microarchitecture and is used by the Python bindings both for detection and for querying, as we will see later.

The compilers field has information on the optimization support different compilers provide for the microarchitecture. This allows archspec to determine how to *build* binaries specifically for this machine and whether a given compiler version will *support* a particular microarchitecture. Build systems can use this feature to query the best flags for their machine, and to pass these flags to their compiler of choice with microarchitecture-specific optimizations.

Currently version 0.1.1 of the archspec JSON database contains information on 45 microarchitectures. The majority of them is relevant in HPC contexts, due to how the project originated and the use cases treated so far, but the design of archspec is not tied in any way to serve *only* that field of application. The addition of microarchitectures used in other domains like, for example, embedded or mobile would not require any change in the current model and may be part of future releases of the project.

2.2. Microarchitecture Detection

One of the challenges of reasoning about microarchitectures is that the granularity of information available programmatically is not the granularity at which microarchitectures can be reasoned about. Microarchitectures are less granular than model numbers—many models of Intel and AMD chips have the same microarchitecture. Microarchitectures are more granular than machine names—the x64_64 machine family contains a bewildering array of different microarchitectures. The archspec library provides a mechanism to query system information and match it with microarchitecture characteristics in the JSON database.

On Linux systems, archspec uses information from /proc/cpuinfo, and on MacOS it uses information from the sysctl command. These sources give us information on available ISA extensions, or *features*. The detection method we use is dependent on the machine family.

For x86_64 systems, archspec uses the "vendor" and "flags" data from either /proc/cpuinfo or the appropriate data source for the operating system. We take the set of flags read from the OS and find the microarchitecture with the largest subset of these features. In other words, we find the compatible microarchitecture with the most features and use that name for the host. We map the detected host and compiler version information to a specific compiler in the compilers list, and we look up the build flags in this list (e.g., for GCC, the appropriate -march flag).

For aarch64 systems, archspec uses the "CPU implementor" and "Features" information from /proc/cpuinfo (or other OS source). Similarly to the x86_64 detection, within each implementor's ecosystem, the "Features" information is compared to the canonical set of features that define each microarchitecture. In the aarch64 case the full set of features is indicative of the microarchitecture.

For ppc64 and ppc64le systems, archspec uses the "cpu" information from the appropriate data source for the operating system. The specific microarchitecture can be extracted directly from the OS information on these systems.

2.3. Compatibility and Comparison

Microarchitectures in archspec are partially ordered by compatibility. We say that microarchitecture A is less than microarchitecture B if code compiled for A can run on B. There are two reasons that this is a partial ordering:

- Separate families are not comparable.
- Microarchitecture generations are not linear within a single family, or even a single family and vendor.

The first point simply states that, e.g., no code compiled for an aarch64 chip will run on an x86_64 chip. The second point is more subtle. The cascadelake and cannonlake microarchitectures are not comparable, because cascadelake includes the AVX512_VNNI instruction that cannonlake does not, and cannonlake includes the AVX512_VBMI and AVX512_IFMA instructions that cascadelake does not. This means that both of the following are false:

```
cascadelake < cannonlake
cannonlake  < cascadelake
```

However, both cascadelake and cannonlake are supersets of skylake, and both are subsets of icelake, so both of the following are true:

```
skylake < cascadelake < icelake
skylake < cannonlake  < icelake
```

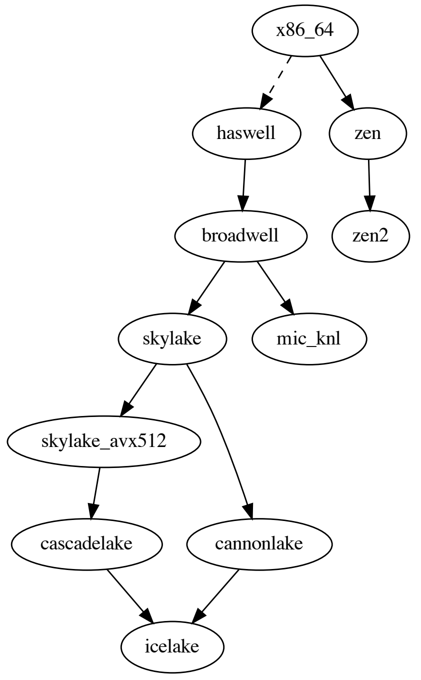

Figure 3. Subgraph showing the partial ordering between the most recent x86_64 CPU microarchitectures known to archspec at the time of writing, including Intel Haswell and its descendants, AMD Zen, and AMD Zen2. The dashed line between x86_64 and haswell indicates that Intel microarchitectures preceding Haswell have been omitted.

These relationships are shown in Figure 3 as a partial graph including the most recent CPU x86_64 microarchitectures known to archspec at the time of writing. Comparison operators make it easy for client code to determine whether a binary or container labeled with one microarchitecture can run safely on a system labeled with another.

2.4. Compiler Flags

In order to label binaries or container images, developers need to be able to compile their code optimized for a specific microarchitecture. The "compilers" information associated with each microarchitecture entry in the JSON file enables archspec to return appropriate optimization flags for each combination of compiler and target.

Optimization flags are specific to both the compiler and compiler version, to support changes to compiler flags over time. For example: versions of the GNU C compiler prior to version 6 do not know about the ARM v8.1-a architecture, and versions prior to version 7 do not know about the thunderx2 microarchitecture. Therefore, there are multiple different ways to target thunderx2 for GCC, depending on the compiler version.

GCC, Intel, and LLVM Clang are the compilers currently supported by archspec, with a format that enables additional compilers to be added easily in the future. In our experience, no previous system has aggregated this information for all of the supported compilers in one place.

2.5. Feature Query

There is no standard for how software packages expose their interactions with the microarchitecture to the user. Most packages leave this entirely to the compiler and compiler flags used by the build system. However, many packages, particularly those with hand-tuned assembly, expose architecture features to the user as build options (like GROMACS, see Figure 1). To support this, Microarchitecture objects in archspec can be queried for individual architecture features. Users can ask questions such as "Does the haswell architecture contain the AVX2 instruction set?" (yes). "Does sandybridge?" (no). This allows users to fine-tune their interaction with the hardware up to the finest possible level of granularity, while maintaining the separation of concerns between knowledge of the underlying architecture (archspec) and decisions about how a specific package builds for that architecture (the package build system).

archspec handles corner cases that OS-level feature queries do not. For example, /proc/cpuinfo reports that older machines have the sse3 extension, and that newer machines have ssse3 but *not* sse3 even though ssse3 is a superset of sse3. Archspec abstracts these details from the user by reporting that machines with ssse3 *also* have sse3, enabling users to write cleaner query code.

3. Example use cases for archspec

The capabilities of archspec outlined in Section 2 are meant for a widespread use in build, packaging and containerization tools. In this section we present use cases to demonstrate the benefits of archspec for situations that involve, at different levels, labeling, comparing, querying and optimizing for specific microarchitectures.

3.1. Package Management: Spack

The archspec library grew out of the Spack project, and so we present it as a first use case. We will see how archspec can be used by package managers to optimize software for a specific target, record the full provenance of the binaries and simplify the build recipes when it comes down to microarchitecture specific choices.

3.1.1. What is Spack. Spack is a flexible package manager that supports building multiple versions and configurations of software on a wide range of platforms and in a reproducible way [7]. There are three fundamental design choices in Spack that make all of the above possible:

- Spack provides a domain specific language (the "*spec syntax*") to specify custom configurations;
- Package recipes are templated and provide information on how to build a given software with many different configurations; and
- Spack employs a constraint resolution algorithm ("*concretization*") to derive all the details of a build that are not explicitly specified by user input.

Broadly speaking, the workflow when installing something with Spack starts with a user specifying a spec to be installed. Spack reads this spec as input, combines it with information from the registered repositories of package recipes, configures a unique DAG with all the details of the software being deployed, hashes it, and finally installs it with full provenance. Users interested in a more in-depth description of Spack features may refer to the tutorial [8].

3.1.2. Spack before microarchitecture support. Before support for microarchitectures was coded into Spack the granularity with which users or packagers could refer to targets was that of the "broad" architecture family (e.g., x86_64 or ppc64le etc.). Software that optimized for microarchitecture features had to either:

1) make their interface more complex and expose microarchitecture choices to the user, or
2) rely on generic or non-portable optimizations.

An example of the first issue is fftw, which introduced variants so users could choose among SIMD instruction sets:

```
$ spack install fftw simd=avx,avx2
```

On the opposite side we find software that like dealii or cp2k was using non-portable flags (e.g., -march=native) to optimize the binaries for the current host.

It is clear that both these approaches are sub-optimal and come with unwanted consequences. In the case of fftw the complexity of keeping track which architecture supports which instruction set is left to the user, while in the case of dealii or cp2k the binaries that are produced are not portable to hosts different from the one where they were built. Ultimately, the absence of a mechanism to register the target microarchitecture of a spec as part of its provenance would hinder Spack's ability to produce optimized code and package it into a binary format that could be portable across different machines.

3.1.3. Spack with archspec. archspec was designed to provide a way for Spack and other tools to reason about microarchitecture compatibility using a "shared language".

Integrating the naming scheme for microarchitectures with the spec syntax allows users to specify which microarchitecture they want to target with a specific installation:

```
$ spack install fftw target=broadwell
```

The full target provenance is recorded with each installation as shown in Figure 4. Since the target is now part of the spec, packages can query it directly in their recipes, as shown in Figure 5. The package can check itself whether the target microarchitecture is new enough to support --broadwell-flag, and exposing specific instruction set options as variants (e.g., with the simd variant shown in Section 3.1.2) is not necessary anymore.

archspec is also employed to compute, for each compiler, the flags needed to activate the generation of binary instructions optimized for the current target. This not only

```
spec:
- zlib:
    version: 1.2.11
    arch:
      platform: linux
      platform_os: ubuntu18.04
      target:
        name: broadwell
        vendor: GenuineIntel
        features:
        - adx
        - aes
...
```

Figure 4. Target provenance stored in spec.yaml for the zlib package.

```
def config_args(self):
  configure_args = []
  # If the target is compatible
  # with Broadwell
  if self.spec.target >= 'broadwell':
    configure_args.append('--broadwell-flag')
```

Figure 5. Querying target compatibility in a Spack python recipe.

eliminates any need to have package recipes that inject non-portable flags like -march=native, but also allows Spack to warn the user if the compiler being used is too old to optimize for the current microarchitecture (in which case the target will be demoted to the best fit) or to error out if the incompatible target request is explicit.

Recording the full target provenance in a binary package allows public binary repositories to distribute optimized binaries, see for instance [9]. From a user standpoint, this will combine both the installation speed of binary package managers and the optimization levels that can be obtained from package managers installing from sources.

Finally, the semantic used by archspec to model microarchitecture compatibility will be leveraged in the future to allow different strategies to pull binary packages from public repositories. Users might choose for instance to always prefer the best compatible target for which a binary package is already available or configure Spack to never go below a given target and install from sources anything that is not already packaged as a binary. The base to implement any of this is the capability to reason about different microarchitectures that archspec provides.

3.2. Optimized Software Stacks: EESSI

archspec is a key component in the European Environment for Scientific Software Installations (EESSI) project [10], where the main goal is to provide a collection of optimized scientific software installations that can be leveraged on a variety of client systems, including HPC clusters, personal workstations or laptops, and cloud instances, all across different CPU families and microarchitectures, regardless of the client's operating system.

The EESSI project consists of three layers:

1) The *filesystem layer*, where the CernVM-FS [11] provides a shared POSIX filesystem that can be mounted via FUSE from anywhere in the world;
2) The *compatibility layer*, where Gentoo Prefix [12] is used to install a minimal set of tools and libraries in a CernVM-FS repository, to make the scientific software stack installed on top of it independent of the operating system of clients; and
3) The *software layer*, in which scientific software applications and all their required dependencies, including compilers and peripheral libraries for MPI, BLAS, LAPACK, FFT, etc. are installed using EasyBuild [3] and Lmod [13], on top of what is provided in the compatibility layer.

The packages in the software layer are compiled for multiple specific CPU microarchitectures, which is critical to ensure good performance (as shown in Figure 1).

`archspec` is leveraged in this context in two main ways. First, the software layer is split up into disjoint subsets, each of which targeting a specific CPU microarchitecture, and located in a specific subdirectory in the CernVM-FS repository. For example, the software installations that target systems with an Intel Haswell microprocessor are located in the `x86_64/intel/haswell` subdirectory; likewise, installations for systems with AMD microprocessor of the Zen2 generation are located in the `x86_64/amd/zen2` subdirectory, etc. The labels for CPU microarchitectures defined by `archspec` are employed here to determine in which subdirectory software should be installed on a particular build host, and to ensure that the naming scheme is easy to interpret by both humans and scripts that in turns also leverage `archspec`.

The other way in which `archspec` is used in the EESSI project is to determine at runtime which of the subsets that are available in the software layer should be selected for a given client system on which the provided software will be used. Note that this goes well beyond finding an exact match between the CPU microarchitecture of the client and the available microarchitecture-specific subsets: if there is no exact match `archspec` can be used to automatically determine which of the available options is the best (compatible) match, thanks to the partial ordering discussed in section 2.3.

For example: assume that software installations for Intel Haswell, Intel Cascade Lake, and AMD Zen 2 systems are provided through EESSI, and that a client system powered by Intel Broadwell microprocessors wants to leverage the provided software. Through `archspec`, a simple initialization script (which is included in the EESSI software layer) can automatically determine that the installations in the `x86_64/intel/haswell` subdirectory are the only compatible option for this client system. On an Intel Icelake client system however, `archspec` would indicate that the best choice performance-wise out of the available options would be the installations in the `x86_64/intel/cascadelake` subdirectory, since those binaries employ AVX-512 instructions that are supported by Intel Icelake microprocessors.

```
feature.node.kubernetes.io/cpu-cpuid.AESNI=true
feature.node.kubernetes.io/cpu-cpuid.AVX=true
feature.node.kubernetes.io/cpu-cpuid.AVX2=true
feature.node.kubernetes.io/cpu-cpuid.FMA3=true
feature.node.kubernetes.io/cpu-cpuid.IBPB=true
feature.node.kubernetes.io/cpu-cpuid.STIBP=true
feature.node.kubernetes.io/cpu-hardware_multithreading=true
feature.node.kubernetes.io/kernel-version.full=4.18.0-211.el8.x86_64
feature.node.kubernetes.io/pci-1af4.present=true
```

Figure 6. Kubernetes node labels generated by the Node Feature Discovery

3.3. Multi-Architecture container builds

Using generally available packages (in the form of container images) from an official source or a certified provider (presented as container registries), comes with a big caveat in relation to performance-sensitive workloads. These packages may provide ABI compatibility at some levels, but they are not optimized for every specialized hardware (e.g GPUs or high-performance NICs), nor every different CPU microarchitecture. One way to address this problem is to compile the application packages (build the container images) for every required architecture.

Orchestrating resource specific or performance sensitive applications requires real time knowledge of the available resource inventory. Kubernetes [14], the open source system for orchestrating container workloads, presents a feature named "*Labels and Selectors*" [15]. Labels are key/value pairs that can be attached to any object, such as nodes. By annotating resources, we are able to deploy applications on environments that best fit its needs or enhance its behavior, which is becoming increasingly important. Nevertheless deploying the same application over different compute nodes can come at a performance disadvantage when it comes to microarchitectural differences on the same CPU chip instruction set [16]. Using the Node Feature Discovery Operator (NFD) [17], we can automate the detection of hardware features and configuration in a Kubernetes cluster by labeling the nodes with hardware-specific information. NFD will label the host with node-specific attributes. In Figure 6 we present some of the labels generated by NFD.

The NFD Kubernetes operator does not provide the required information that package managers like Spack require to optimize builds via compilation flags. But NFD can be extended with sidecar containers and hooks. We have developed a sidecar container for NFD, that will label nodes with information provided by `archspec`. Named *archspec-feature-discovery* [18], this sidecar container will expose all the CPU microarchitecture information of the host. For example:

```
archspec.io/cpu.vendor=GenuineIntel
archspec.io/cpu.model=85
archspec.io/cpu.family=6
archspec.io/cpu.target=skylake_avx512
```

The Openshift community Distribution of Kubernetes (OKD) [19] provides a way to orchestrate image builds based on defined events called *builds*. A build is the process of transforming input parameters into a resulting object.

```
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
    archspec.io/cpu.target: skylake_avx512
```

Figure 7. Kubernetes buildConfig manifest.

```
{
  "OCIv1": {
    "config": {
      "Labels": {
        "io.archspec.cpu.vendor": "GenuineIntel",
        "io.archspec.cpu.model": "85",
        "io.archspec.cpu.family": "6",
        "io.archspec.cpu.target": "skylake_avx512"
      }
    }
  }
}
```

Figure 8. archspec labels in an OCI image descriptor.

Most often, the process is used to transform input parameters or source code into a runnable image. A BuildConfig object is the definition of the entire build process.

The missing part for building hardware-specific images is to orchestrate the build process over the different available resources. Kubernetes labels are a grouping primitive, that can be used to group or processes by label equality. By adding a nodeSelector stanza in the buildConfig manifest the client/user can define a set of labels/nodes where to run the image build (Figure 7).

By adding the above snippet to a Kubernetes job manifest, we constrain any pod created by that job to be allocated on nodes labeled with the desired label.

3.4. Custom image labels

Another approach is to annotate the microarchitecture in the image itself. The open containers initiative (OCI), version 1 image specification [20] comes with a property OCIv1.config.Labels that allows users to add metadata to an image. An image label is a key-value pair and must follow the OCI annotations rules [21]. Canon and Younge 2019 [16] proposed to store the architectural details of packages inside the container, such as processor features, whether the container application requires a specific glibc, MPICH support, or CUDA driver on the host in order to run. Tools like buildah [22] allow users to add labels to images once they are built. buildah can then be used in conjuction with archspec to create labels than will be stored as image metadata. Following the example from the previous section, the labels stored in an image config will look as shown in Figure 8.

Finally, it should be noted that in principle archspec could also be used to have an OCI image index [23] reference multiple image manifests, each optimized for a different microarchitecture. Investigating this approach in more detail is under consideration.

4. Related work

archspec's key differentiators are its ability to reason about compatibility using comparators, its database of compiler flags, and its language-agnostic JSON model. These features allow it to fit easily into toolchains for packaging or container management, and to be used as an aid for distributing, selecting, and using binary packages.

Existing tools aim to *detect* the underlying architecture or to enable different types of microarchitecture-specific code to be used at runtime. For example, the CPUID [24] tool queries the CPUID bits on Intel, AMD, and other vendors' processors, and returns a feature set and a processor name. There is no structure beyond this, and the library does not know the compatibility relationships among chips. Google's cpu_features library [25] similarly provides logic for detecting feature information from various OS data sources, but it does not map this information to specific microarchitectures. While it does allow client code to detect whether a compiler has been *passed* a microarchitecture-specific optimization option, it does not provide archspec's database or any way to *look up* options for a compiler. The user is still responsible for build configuration. Both of these are informational runtime libraries; they do not provide archspec's rich DAG-based compatibility model.

gcc and the glibc dynamic loader, ld.so have some archspec-like features built in. In particular, ld.so has a *multilibs* [26] feature that allows users to build multiple versions of the same library and deploy them in different, microarchitecture-specific subdirectories under any normal library search path. This allows vendors to build libraries for multiple architectures and deploy them together, similar to a fat binary but with dynamic ld.so support and separate files. gcc provides a mechanism that is similar in spirit – it allows different microarchitecture-specific versions of the same function to be labeled, compiled into the same binary, and dispatched dynamically at runtime based on the host [27]. This approach is likely the most robust. It allows the most important functions in a library to be optimized separately for specific platforms, and it allows a single binary to work well across environments. However, this approach is also the most invasive, as the user must instrument their library with compiler-specific features and code variants. We use archspec to enable solutions that work for *all* compilers and unmodified binaries.

Optimized binary distribution is also being investigated in other ecosystems. The Julia community has modified their BinaryBuilder tool to support microarchitecture-specific optimizations and their own labeling scheme for CPU features [28]. The glibc community is investigating a *coarser* set of architecture *levels* [29], which they hope will reduce the number of microarchitectures that need to be considered for optimized builds. These are essentially "synthetic" microarchitectures – chosen because they balance compatibility and performance well. We may consider adding these to archspec as additional nodes in the DAG, which would allow for direct comparison between gcc's coarse levels and detected host architectures.

5. Future work

archspec currently models CPU compatibility, but we also aim to model other hardware features. In particular, GPU compatibility and networking interfaces cause many headaches for HPC binary distribution. Shipping binary artifacts that are portable across GPUs and networking devices is non-trivial [16], and we hope to enable tools to reason about these aspects just as we have done for the CPU.

In addition to this, we are looking into adding more "virtual" microarchitectures to our database. We currently model real CPUs, but ARM provides versioned ISA specifications, like armv8-a or armv8.1-a, and glibc appears to be adding similar concepts with their "levels" of x86_64 support. We will consider adding these generic targets and their compiler information to our database, as they provide a good balance between optimized, microarchitecture-specific builds and more widely distributable binaries.

6. Conclusion

The diversity of modern microarchitectures is growing rapidly, and with the death of Moore's law it will become increasingly important to exploit the ISA extensions on these new chips. However, without better tooling that can *reason* about compatibility, the average user will never see these benefits. Packaging and container tools are needed that can reason about microarchitecture compatibility and select the *fastest* available binary for a given host. We have presented the archspec library, which provides a DAG-based compatibility model for CPU microarchitectures. It enables simpler querying and simpler building of binaries, as it encapsulates feature compatibility information as well as the compiler flags needed to exploit these features. It provides essential comparison features that container runtimes, package managers, and orchestrators can use to compare builds – something no existing tool provides. It is our hope that archspec will be picked up by the packaging and container communities and used to enable the use of optimized binaries for a much broader range of users. archspec is available online at https://github.com/archspec/archspec.

Acknowledgments

This work was performed under the auspices of the Colombia "Sistema General de Regalías", Contract BPIN-2013000100007, by COLCIENCIAS, Gobernación del Valle del Cauca. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-815724).

References

[1] T. Callaway, "Fedora Packaging Guidelines," October 10 2020. [Online]. Available: https://docs.fedoraproject.org/en-US/packaging-guidelines/

[2] "PRACE Unified European Applications Benchmark Suite (UEABS)." [Online]. Available: https://repository.prace-ri.eu/git/UEABS/ueabs

[3] K. Hoste, J. Timmerman, A. Georges, and S. Weirdt, "EasyBuild: Building software with ease," 11 2012, pp. 572–582.

[4] "GROMACS website." [Online]. Available: http://www.gromacs.org

[5] H. J. C. Berendsen, D. V. D. Spoel, and R. V. Drunen, "Gromacs: A message-passing parallel molecular dynamics implementation," *Comp. Phys. Comm*, vol. 91, pp. 43–56, 1995.

[6] "GCC documentation." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html

[7] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral, "The Spack Package Manager: Bringing order to HPC software chaos," in *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015.

[8] T. Gamblin, G. Becker, and S. Smith, "Spack Tutorial on AWS," Virtual event, Jul. 2020, virtual event. [Online]. Available: https://spack-tutorial.readthedocs.io/en/aws20/

[9] "The Extreme-scale Scientific Software Stack." [Online]. Available: https://e4s-project.github.io

[10] "European European Environment for Scientific Software Installations (EESSI)." [Online]. Available: https://eessi.github.io/docs

[11] J. Blomer, G. Ganis, N. Hardi, and R. Popescu, "Delivering LHC Software to HPC Compute Elements with CernVM-FS," 10 2017, pp. 724–730.

[12] "Gentoo Prefix." [Online]. Available: https://wiki.gentoo.org/wiki/Project:Prefix

[13] "Lmod: A new environment module system." [Online]. Available: https://lmod.readthedocs.io

[14] "Kubernetes homepage." [Online]. Available: https://kubernetes.io

[15] "Kubernetes labels and selectors." [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels

[16] R. S. Canon and A. Younge, "A case for portability and reproducibility of hpc containers," in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2019, pp. 49–54.

[17] "Node feature discovery." [Online]. Available: https://github.com/kubernetes-sigs/node-feature-discovery-operator

[18] "Archspec feature discovery." [Online]. Available: https://github.com/archspec/archspec-feature-discovery

[19] "Openshift community distribution of kubernetes." [Online]. Available: https://www.okd.io

[20] "Oci image configuration." [Online]. Available: https://github.com/opencontainers/image-spec/blob/v1.0.1/config.md

[21] "Oci annotations rules." [Online]. Available: https://github.com/opencontainers/image-spec/blob/v1.0.1/annotations.md

[22] "Buildah." [Online]. Available: https://buildah.io

[23] "OCI Image Index." [Online]. Available: https://github.com/opencontainers/image-spec/blob/master/image-index.md

[24] "Todd Allen's Tools: cpuid," http://www.etallen.com/cpuid.html, (Accessed on 09/11/2020).

[25] "google/cpu_features: A cross platform C99 library to get cpu features at runtime." https://github.com/google/cpu_features, (Accessed on 09/11/2020).

[26] D. Wiki, "Multiarch paths and toolchain implications," https://wiki.debian.org/Multiarch/LibraryPathOverview, October 20 2015, (Accessed on 09/11/2020).

[27] "Function multi-versioning in GCC 6 [LWN.net]," https://lwn.net/Articles/691932/.

[28] E. Saba and M. Giordano, "BinaryBuilder.jl - The Subtle Art of Binaries That Just Work," in *JuliaCon*, July 25 2020. [Online]. Available: https://www.youtube.com/watch?v=3IyXsBwqll8

[29] F. Weimer, "New x86-64 micro-architecture levels," https://gcc.gnu.org/pipermail/gcc/2020-July/233088.html, July 10 2020.