Summer student report

1 August 2024

elvis.alexander.aguero.vera@cern.ch

Enabling Julia code to run at scale with artefact caching

Elvis Aguero
Supervisors: Graeme A. Stewart and Pere M. Vila
CERN, Geneva, Switzerland

Keywords: Julia, Distributed Computing, Precompilation, Compilation, LLVM, CVMFS

Summary

The Julia programming language, which has evolved into a mature tool for scientific computing over the past decade, offers high-level capabilities with just-in-time (JIT) compilation and efficient garbage collection. Its performance, comparable to that of C, makes Julia an attractive option for the high-energy physics community. However, Julia's use of intermediate precompiled files to achieve this performance introduces significant startup delays, known as "time to first plot" (TTFP), which would be a severe drawback in distributed systems where each node would have to compile dependencies locally.

This project addresses the TTFP issue by leveraging the shared CernVM-FS (CVMFS) file system to distribute files across nodes and Julia to prepare a ready-to-use folder with all necessary precompiled files for selected applications, thereby reducing startup times. We developed and tested a framework for automating the publication of precompiled Julia files to CVMFS and evaluated its impact on performance using two sample applications: the Julia Jet Reconstruction package and the Geant4 wrapper package. Our findings demonstrate that precompiled files stored in CVMFS significantly reduce startup times, with reductions of up to 97% for the Geant4 package. Furthermore, we examined the effects of cross-compilation for various microarchitectures and found that nodes benefit from shared cache files without notable performance degradation due to microarchitecture differences.

Contents

1 Introduction

Julia is a promising language for high-energy physics [1] as it combines the easy of use and ergonomics of dynamic languages such as Python, with the runtime speed of C or C++.

One of Julia's features is that it uses a JIT (or just-ahead-of-time) compiler to target the specific architecture on which it is being run. This however, comes at the cost of the compilation time, meaning that the first pass through the code is slower.

For Julia to be widely adopted in high-energy physics, especially at a large scale, it is essential to minimize this overhead by precompiling Julia code beforehand, thereby avoiding the need for recompilation on every node. This can be achieved using the `DEPOT_PATH` environment variable.

In this project, we investigate these strategies on cluster systems at CERN, such as lxbatch. We measure startup time and runtime performance as the number of jobs increases. Additionally, we explore the potential of caching Julia code on CVMFS, enabling scalability across the entire grid. Lastly, we assess the feasibility of precompiling artifacts for different microarchitectures, which could harness the full capabilities of modern CPUs residing in large-scale heterogeneous systems alongside other, older CPU servers.

2 Making cache files relocatable

2.1 Understanding how Julia handles cache files

A script in julia is a `.jl` file, which generates `.ji` and `.so` via the LLVM compiler. The `JULIA_DEPOT_PATH` environment variable is used to populate the global Julia `DEPOT_PATH` variable, which controls where the package manager, as well as Julia's code loading mechanisms, look for package registries, installed packages, named environments, repo clones, cached compiled package images, configuration files, and the default location of the REPL's history file.

The `DEPOT_PATH` variable is a list of strings, each pointing to a directory on the current machine. The first entry in this list must be writable, while the subsequent entries are

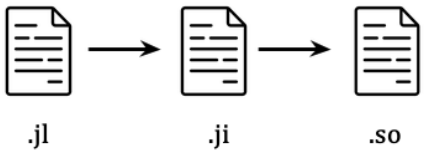

Figure 1: Pipeline of generated files in the compilation stage of Julia's JIT.

treated as read-only. Julia will iterate through these entries and use the first available cache file compatible with the host machine. If none are found, it will generate a new cache file in the writable entry specified in the `DEPOT_PATH` list.

By default, the first entry in the `DEPOT_PATH` list is the directory `$HOME/.julia/`. This directory contains all precompiled cache objects and artifacts unless another path is specified. This setup allows multiple applications and workflows to coexist, sharing files in the same `DEPOT_PATH`.

2.2 Cache validation in Julia

Julia determines the suitability of a cache file for the current session based on several factors. Below is a non-comprehensive list of these factors, and this analysis is based on the source code of the candidate release 1.11.

1. The relative path of the generated cache file, which is hashed into the metadata of the `.ji` file.
2. Modification times of cache files, which are also stored as metadata.
3. The set of image targets for which a cache file was generated and their compatibility with the current machine.

2.2.1 Absolute Paths

Firstly, it is important to note that prior to Julia 1.11, absolute paths were embedded into the metadata stored in the `.ji` precompiled files. This approach is suboptimal for distributed systems, as changing a single parent's directory name would flag the cache file as invalid, despite the contents of the file remaining unchanged. As of August 2024, a production-ready version of Julia 1.11 has not yet been published, but the release candidates addresses this issue by using only relative paths for packages inside the `DEPOT_PATH`.

2.2.2 Modification time of files

Secondly, cache validation depends on the modification timestamp of the cache files. The modification time (`mtime`) of a file can be checked using the `ls -l directory/` command. Therefore, when transferring files from one directory to another node on the grid, it is crucial to preserve the mtime of all files in the `DEPOT_PATH`. To achieve this, one can use `rsync`, which, by default, preserves the mtime during the transfer.

2.2.3 Cross-compilation

Another key factor that can invalidate a cache file is the set of image targets for which it was generated. Each node in the grid has an Instruction Set Architecture (ISA) that defines the assembly instructions the machine can execute. When Julia starts, it defaults to using the host's ISA to generate all cache files. If a cache file is encountered by another node whose ISA is incompatible, the cache file will be rejected.

This behavior is beneficial because it ensures that Julia always tries to use the most suitable cache files for computation. It also provides stability, as Julia can generate a new cache file at runtime if no compatible files are available in the current `DEPOT_PATH`.

Also, Julia gets cross-compilation capabilities through LLVM which can be leveraged by setting the `JULIA_CPU_TARGET` environment variable. For instance, the following command:

```
export JULIA_CPU_TARGET=generic;sandybridge,clone_all;haswell,-rdrnd
```

instructs Julia to generate cache files for three image targets: a generic target compatible with the current machine architecture (e.g., ARM or x86-64, etc.), the "sandybridge" microarchitecture with all functions cloned for that specific target, and the "haswell" microarchitecture, but without precompiling the "rdrnd" function, thus providing fine-grained control over functions that can be selected for cross-compilation. In this example, ISAs are separated by semicolons, and various flags can be set using comma-separated values.

This setup allows any machine with a generic x86-64 architecture to use the cache file, and machines compatible with either "sandybridge" or "haswell" can also reuse that cache file, taking advantage of more modern CPU instructions. To check the current image targets available on a grid node, the command `Sys.CPU_NAME` can be used to print the machine's ISA.

3 Publication procedure onto CVMFS

The CERN Virtual Machine File System (CVMFS) is a scalable, reliable, and low-maintenance software distribution service used at CERN to deploy software and files across the worker nodes available to all experiments, particularly for offline post-processing. To reduce startup time of Julia applications, they can have access to a ready-to-use `DEPOT_PATH` directory inside CVMFS and treat it as read-only. To achieve this, it is necessary to prepare, precompile, and distribute all relevant packages into a single directory and then publish them to CVMFS. We propose the following workflow for the publication procedure:

1. Select the application workflows to be supported on CVMFS. These may include packages developed by the high-energy physics (HEP) community or any other modules that can be precompiled and would benefit the nodes in the grid.

2. Choose the image targets for which the applications should be precompiled. Ideally, include a generic target for the current architecture.

3. Use a wrapper script to precompile the `Project.toml` file of each application. This `.toml` file, which can be generated by running the program once, lists all of its dependencies. All applications should share the same `DEPOT_PATH` to ensure that cache files are reused, minimizing the number of files generated during this process.

4. If the previous steps were performed on a node without publication credentials for CVMFS, log in to a node with the necessary credentials and `rsync` the directory containing all cache files to the publishing node.

5. Finally, trigger the publication procedure on the publishing node using the following commands:

```
cvmfs_server transaction sft-test.cern.ch
rsync -r --ignore-existing src/ /cvmfs/sft-test.cern.ch/cvmfs_environment/
cvmfs_server publish sft-test.cern.ch
```

3.1 Automation of the procedure

The DepotDelivery.jl package is a public package available in the Julia registry that facilitates the downloading and preparation of dependencies for air-gapped environments. However, for the specific use case of this project, the package lacked certain features. To address these needs, we submitted a pull request (PR) to the repository, extending the package's capabilities by implementing the following features:

- Support for an user-defined `DEPOT_PATH` destination.
- Support for `Project.toml` files generated by the project dependencies of an application, even if the application itself is not a package. This support includes creating multiple subdirectories within the target depot path and copying only the relevant `.toml` files to the final destination.
- Support for caching all the artefacts needed to support multiple workflows.

As of the publication of this document, the PR is under review, and the author has expressed their intention to merge our contribution.

4 Results

A number of experiments have been devised to test the proposed framework of the last sections. First, the Julia 'JetReconstruction.jl' [2] and the 'Geant4.jl' wrapper package [3] as examples of applications that can be used at scale on a grid. Two simple examples, one for each application were chosen to represent a typical run that can be made at a node of the grid. We used the lxbatch service provided by CERN to distribute computing jobs accross nodes.

4.1 Saving precompilation time at CVMFS

Probably the most relevant and striking contribution of this project is the computational time saved from applications that have to run for the first time. To measure this, 20 jobs were submitted for each of the two applications without connecting to CVMFS, forcing

precompilation of all dependencies. Another 20 jobs were then submitted, this time allowing the nodes to access CVMFS, where precompiled files were already available.

Table 1 shows the time saved from having precompiled files available in CVMFS. It can be observed that the Julia Jet reconstruction package has saved around 90% of its runtine, and Geant4 had reduced its runtime by 97%, as it has a heavier set of external dependencies.

	JET	Geant4
Without cache	90 ± 7	270 ± 80
With CVMFS Cache	6 ± 2	6 ± 1

Table 1: Time in seconds it takes for Julia to start running the workflow.

This example also shows that all nodes were able to benefit from the same cache file, regardless of the host's microarchitecture. As it can be seen in figure 2, the distribution of times with precompilation is less skewed, as there are multiple factors that influence precompilation time.

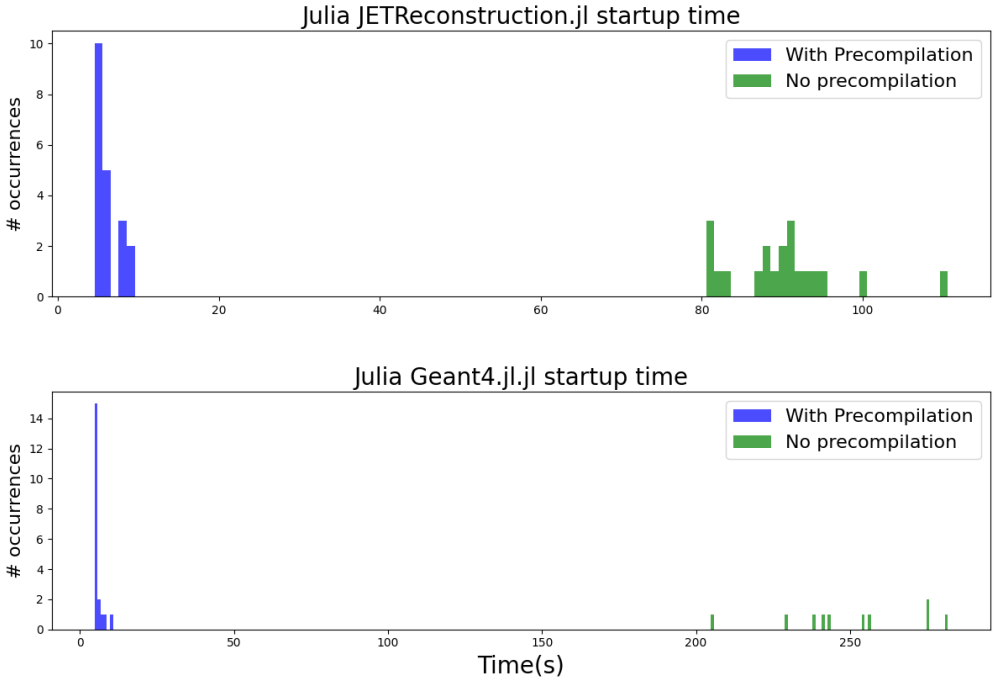

Figure 2: Time in seconds for 20 jobs with and without precompiled files available from CVMFS for both packages used.

4.2 Performance impact of CVMFS cache

CernVM-FS utilizes HTTP connections and caches files locally when available. When a node requests a cache file for the first time, it may not have the file stored locally and must retrieve it from the server. The system makes an HTTP request for the file, which cascades up through various proxy cache levels to fetch the file. The file will then be cached locally for future use, and cleaned periodically using an LRU policy.

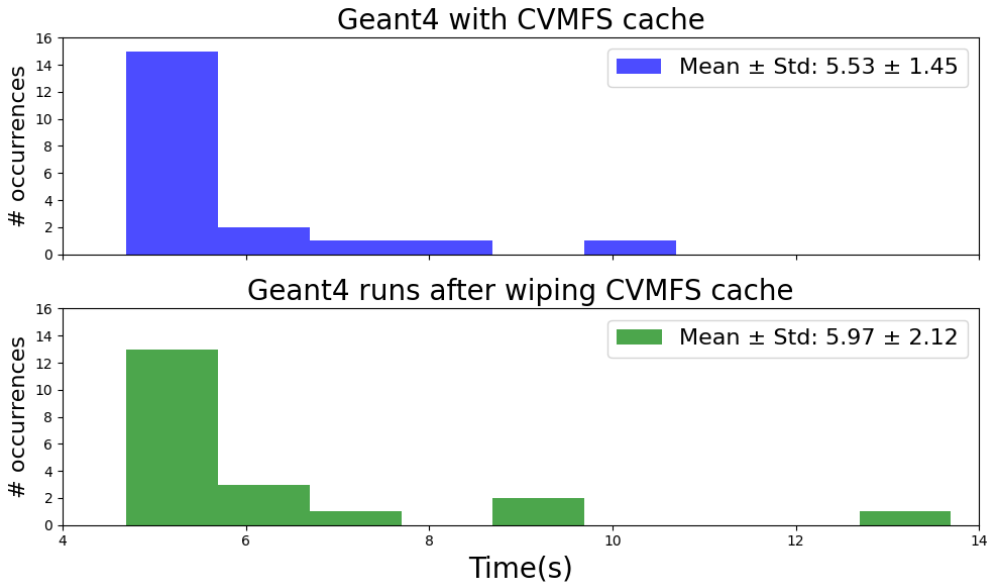

Figure 3: Time in seconds for 20 jobs with and without readily available cache files from CVMFS for the Geant4 package.

This scenario leads to additional retrieval time, and it's called "cold cache". In contrast, a "hot cache" refers to the state where the file is readily available locally, reducing retrieval time.

To evaluate CVMFS's performance in providing these cache files, we conducted experiments on the lxbatch service, comparing the cold and hot cache states.

Figure 4.2 presents the results for 20 jobs with cold cache and 20 jobs with hot cache for the Geant4 wrapper package. The data suggests that CVMFS overhead on cache files does not significantly impact performance of jobs running at CERN (which are thus close to the CVMFS upstream servers). This conclusion is supported by similar results for the Julia Jet Reconstruction application, as shown in Figure 4.2.

5 Conclusion

The Julia programming language has demonstrated its potential for large-scale use in high-energy physics applications. A framework for automating and publishing precompiled files on CVMFS has been proposed, and initial performance metrics have been evaluated using two sample applications relevant to CERN.

The project successfully established methods for generating precompiled files that are compatible with multi-architecture systems. This achievement is crucial for optimizing performance across diverse computational environments.

For future work, several areas need further exploration. Firstly, the impact of using a generic CPU implementation on performance should be assessed, as generating excessively large cache files for native compilation for all grid nodes might be impractical, calling for generation of multi-architecture builds for the parts of the workflows that benefit more from this, without unnecesarily increasing the size of the artefact files. Therefore, obtaining a sample

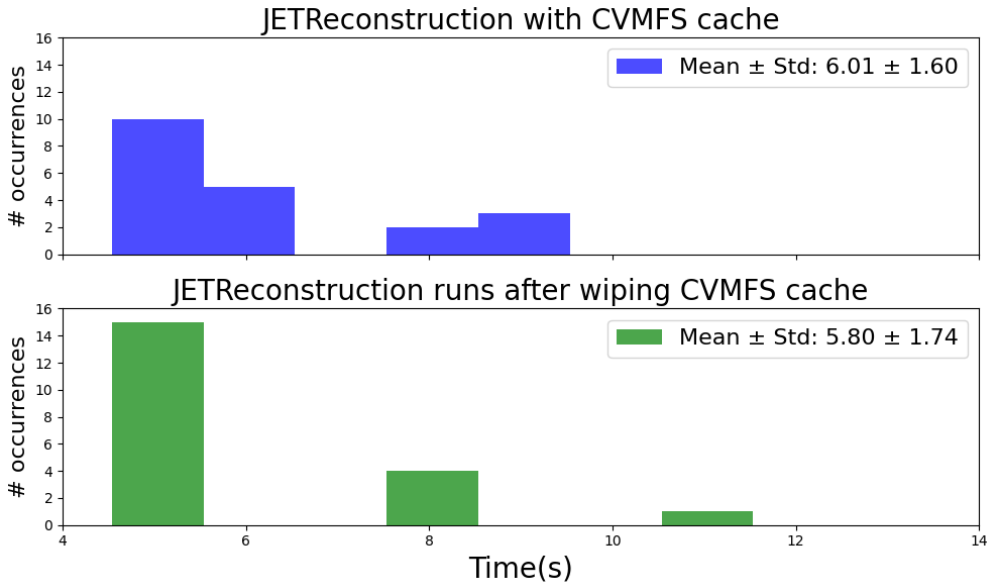

Figure 4: Time in seconds for 20 jobs with and without readily available cache files from CVMFS for the Julia Jet Reconstruction package.

of the nodes in the grid to identify and focus on the most prevalent microarchitectures could be beneficial. Precompiling for these specific instruction sets, in conjunction with a generic target, may enhance efficiency. Lastly, developing a script to fully automate the publishing process to CVMFS is essential, as the current framework separates the precompilation and publication stages.

References

[1] Jonas Eschle, Tamás Gál, Mosè Giordano, Philippe Gras, Benedikt Hegner, Lukas Heinrich, Uwe Hernandez Acosta, Stefan Kluth, Jerry Ling, Pere Mato, et al. Potential of the julia programming language for high energy physics computing. *Computing and Software for Big Science*, 7(1):10, 2023.

[2] Graeme Andrew Stewart, Philippe Gras, Benedikt Hegner, and Atell Krasnopolski. Polyglot jet finding, 2023.

[3] Pere M Vila. Geant4.jl. https://github.com/JuliaHEP/Geant4.jl, 2024. Accessed Aug 1st, 2024.