University of Basel

DRAFT VERSION

01.01.1980

Gachix: A content-addressable binary cache for Nix over Git

Bachelor Thesis

Department of Computer Science
Department of Mathematics and Computer Science
Computer Networks Group
cn.dmi.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Dr. Erick Lavoie

Ephraim Siegfried
e.siegfried@unibas.ch

January 01, 1980

Abstract

This project develops a binary cache for Nix packages using Git's content-addressable filesystem. This approach improves upon traditional input-addressable packages by reducing memory usage and enhancing trust. Leveraging Git provides a key advantage: simple peer-to-peer replication of the binary cache across multiple nodes. The core work involves modeling package dependency graphs and user profiles within Git and creating an interface for Nix to interact with this system. The project will be evaluated through performance benchmarks measuring memory usage and package retrieval speed, alongside functional tests of the peer-to-peer replication and Nix interface.

Table of Contents

1
Introduction

2
Background

2.1. Nix

Nix is a declarative and purely functional package manager. The main purpose of Nix is to solve reproducibility shortcomings of other package managers. When packaging an application with RPM for example, the developer is supposed to declare all the dependencies, but there is no guarantee that the declaration is complete. There might be a shared library provided by the local environment which is a dependency the developer does not know of. The issue is that the app will build and work correctly on the machine of the developer but might fail on the end user's machine. Nix provides a functional language with which the developer can declaratively define all software components a package needs. Nix ensures that these dependency specifications are complete and that Nix expressions are deterministic, i.e. building a Nix expressions twice yields the same result. [1]

2.1.1. The Nix Store

The Nix store is a read-only directory (usually located at /nix/store) where Nix store objects are stored, which are (most importantly) all binaries of a system and components for building the binaries. Store objects can refer to other store objects. Notably Nix ensures that references in binaries are unique paths to objects in the Nix store. This prevents undeclared dependencies and interference with the system environment. There are two major types of Nix stores, which depend on how these unique paths are stored. The prevalent way is to compute the path by hashing the package build dependency graph (see Section). There is also an experimental feature where the paths are computed by hashing the resulting binary (see Section).

TODO: Add figure and example

Since the hash is computed recursively, any change in the dependencies of an application is reflected in the hash. Thus the hash is a unique identifier for a configuration.

Nix builds the software components in a sandbox. It ensures that in the build process of a component only exclusively declared build and runtime dependencies can be accessed (e.g. /bin gets pruned from the environment variable PATH) and network access is limited. This makes component builders pure; when the deployer fails to specify a dependency explicitly, the component will fail deterministically. [2]

2.1.2. Derivations

A derivation (a file with the .drv extension) is a build plan that specifies how to create one or more output objects in the Nix store. It is an intermediate artifact generated when a Nix package expression is evaluated, analogous to an object file (*.o) in a C compilation process. A derivation consists of:

- A name
- A set of paths to other drv files needed in order to build the object.
- An outputs specification, specifying which outputs should be produced, and various metadata about them.
- The system on which the executable will be run.
- The specification of the builder which will produce the object. This is usually a script which defines how to build the package (e.g. builder.sh), the shell which executes the script (e.g. bash) and all environment variables passed to the shell. [3]

To build a derivation, Nix first ensures all its input derivations are built. It then runs the builder in an isolated sandbox, providing only the resources declared in the derivation. The final result is a immutable output in the Nix store.

We call a derivation *pure* if it produces the same output regardless of when, where, or by whom it is built, given the same inputs. This implies that in pure derivations there is no network access, since the URL content might change or the server might go down. A derivation is called *fixed* if the output hash is known and declared before the build process and else it is called *floating*. [4]

There are different types of derivations, depending on how the derivation output is addressing. In the following the most important types of derivations are explained.

Input-Addressing Derivations

In input-addressing derivations, the output path of the resulting Nix object is computed by hashing all its inputs, more specifically:

- The sources of all input components.
- The script that performes the build.
- Any arguments or environment variables passed to the build script.
- All build time dependencies. [2]

Because the hash is computed before the build, input-addressing derivations are fixed. Any change to an input, even a dependency's dependency, alters the final hash, ensuring deterministic builds.

Furthermore, input-addressing derivations must be pure. To illustrate, consider a derivation that fetches a repository's latest commit. Its output is not guaranteed to be consistent over time, as new commits could alter the result. Input-addressing prevents such impurities by requiring all inputs to be fully defined and hashed at the time of derivation creation, thereby proving the build's reproducibility.

Content-Addressing Derivations

The disadvantage of input-addressing derivations is that changes in the inputs that don't alter the resulting Nix object (e.g. a comment in the source code) will also alter the hash and force a rebuild of the derivation. This is prevented in content-addressing derivations which only hash the content of the resulting Nix object. It enables early cutoff, i.e. stopping a rebuild if it can be proven that the end result will be the same as a known object in the Nix store. [5] In content-addressing derivations the purity condition is lifted.

TODO: Explain why

2.1.3. The Deployment Pipeline

2.2. Git

3 Design

3.1. Mapping Nix packages to Git objects

This chapter introduces how Nix concepts are mapped to the Git object model. Just like the Git object database, the Nix Store is an immutable collection of data. Apart from packages the Nix store also stores source files, links and other types of objects. Since a binary cache only serves binary packages, we can focus on only those type of objects.

Every top-level entry in the Nix store is uniquely addressable by some hash. There exist also files which are not uniquely addressable, which are inside those directories (e.g. a directory called 'bin'). When mapping Nix entries to Git objects we have to make sure that the corresponding objects in Git are also uniquely addressable. Fortunately, this is already the case in Git, because Git uses the hash of the data content to address it.

A Nix package is either a single file executable or a directory containing an executable. In Git, files can be mapped to blobs. This mapping will lose file metadata information: When constructing a blob from a file, and then reconstructing the file, it cannot be known whether the file is executable. There are many ways to solve this issue. One way to solve this, is to always assume that top-level files are executable, because the binary cache only serves executables. When sending the Nar of a top-level file to a client, we can always mark it as executable. Another way to solve this, is to construct a Git tree with the blob of the file as the single entry. In this tree, we can mark the blob as executable. In order to distinguish this tree from other trees, the blob in this special tree can be named with a unique magic value which marks the tree and the blob as a single file executable. The latter approach is favourable, because in Section 3.2. we will create commits for each package, and commits can only point to trees and not blobs.

Package directories can be mapped to Git trees. These directories can contain symbolic links, files and other directories. Directories can be mapped to trees, symbolic links and files can be mapped to blobs, while metainfo such as the name of the objects can be stored in the trees.

In order to track which packages have which runtime dependencies, Nix manages a Sqlite database. This database tracks which Nix paths have references to other paths in the Nix store. This information helps Nix to copy package closures to other stores and it prevents deleting packages which have references to them. In Gachix this dependency management is achieved using commit objects. For each package, a commit object is created where the commit tree is the tree containing the package contents as described above. The parents of the commit are the dependencies of the package which are also represented as commits. To find out what the dependency closure of a package called 'foo' is, we can follow recursively follow the parent pointers which is equivalent to running `git log <foo-commit-hash>`.

Notice that after mapping Nix store entries to Git objects, the size of the database will have decreased. One reason is that Git compresses its objects. Nix does not compress entries because the Nix store is not just a passive database as the system accesses and runs files inside it. Although Git reduces the size of the database, it's not a drop in replacement of the Nix store, because it does not allow to easily access files. Another reason why the size of the data will decrease, is because all objects in Git are content addressed. If two package directories in the Nix store contain a file with the exact same content, Git will store this file as a blob only once. The package directories will be mapped to trees which point to the same blob.

3.2. Dependency Management

Bibliography

[1] The NixOs Foundation, "How Nix Works." Accessed: Sep. 19, 2025. [Online]. Available: https://nixos.org/guides/how-nix-works

[2] E. Dolstra, "The Purely Functional Software Deployment Model," 2006.

[3] Nix Dev, "Nix Derivations." Accessed: Sep. 22, 2025. [Online]. Available: https://nix.dev/manual/nix/2.31/store/derivation/index.html

[4] Nix Dev, "Derivation Outputs and Types of Derivations." Accessed: Sep. 22, 2025. [Online]. Available: https://nix.dev/manual/nix/2.31/store/derivation/outputs/index.html

[5] NixOS Wiki, "Ca-derivations." Accessed: Sep. 22, 2025. [Online]. Available: https://nixos.wiki/wiki/Ca-derivations