A Solution to Static vs. Dynamic Linking

Ted Blackman ~rovnys-ricfer
Philip C. Monk ~wicdev-wisryt
Urbit Foundation
Tlon Corporation

Abstract

Computing systems that utilize library code must either link to a self-provided version (static linking) or a system-supplied version (dynamic linking). This can lead to memory duplication or dependency hell. The Urbit `++ford` build system elegantly solves for the linking problem by promoting structural sharing of objects (nouns) in memory and by utilizing a referentially transparent build cache.

Contents

Manuscript submitted for review.

Address author correspondence to ~rovnys-ricfer.

1 Introduction

A compiled computer program is conventionally built by parsing and compiling the source code into an object file and then linking that object file to library code, yielding an executable file. That linking can be accomplished in two ways: either directly including all of the library code in the program file, or supplying the library code in the operating system as a service. Programmers have to balance flexibility, portability, and dependency management when deciding how to link a program, but both approaches can still lead to practical difficulties.

2 Static vs. Dynamic Linking

"Static linking" includes all library code in the program executable file. Static linking is the naïve way to combine source files into a program: compound the files together into one object and then compile that into a program. The semantics tend to be clean and simple, and many programs are linked this way. Statically-linked programs can also run faster since they do not need to resolve library references during execution.

The problem with static linking is that if the same library is used by more than one application, there is more than one copy of it in memory as a result. This memory duplication can be demanding on RAM utilization and reduce cache locality, degrading overall system performance.

Dynamic linking was invented to address this problem. Instead of linking a library into a program at build time, one links it at runtime. The OS keeps a single copy of the shared library in RAM that multiple programs can use. This reduces memory usage and improves performance, but can lead to version mismatches and dependency issues. Locklin explains,

> The shared object concept itself is a towering failure. This is little appreciated but undeniably true. The idea of the shared object is simple enough: if you have a computer running lots of code, some of the code used will be the same. Why not just

> load it to memory once and share that memory at runtime? … When people invented shared objects back in the 1960s, the computer was a giant, rare thing ministered to by a priesthood: there was no such thing as multiple versions of a shared object. You used what the mainframe vendor sold you.
>
> It's now such an enormous problem there are multiple billion-dollar startups for technologies for dealing with this complexity by adding further complexity. Docker, Kubernetes, various Amazon atrocities for dealing with Docker and Kubernetes and their competitors, Flatpak, Conda, AppImages, MacPorts, `brew`, RPM, NixOS, `dpkg/apt`, VirtualBox, `pacman`, Yum, SnapCraft…. As the number of packages grows, this breaks down, and even the OS maintainers are giving up and turning to flatpak, AppImages and Snap files. These are extremely complicated and incredibly wasteful *(of memory)* ways of literally packaging up a bunch of needed shared libraries with your application and presenting it to you as a crappy simulacrum of a statically compiled binary. (Locklin, 2022)

Locklin's picturesque exposition highlights the "dependency hell" or "DLL hell" that mires modern software development (cf. Grimes (2003)). From Urbit's perspective as a solid-state computer, another problem with dynamic linking is that it is not deterministic. Dynamic linking was something of a pact with the devil, permitting efficient memory usage at the price of legibility.

Programmers have to balance flexibility, portability, and dependency management when deciding how to link a program.

3 Building Code Deterministically

Determinism has long been a desired characteristic of any given build system. The source code should be a function of build

environment and build instructions in a straightforward way. Even accessing the linked libraries in a different order can alter the resulting binary, however, meaning that true reproducibility is elusive. Declarative package managers like Nix (NixOS, 2024) and Guix (GNU Guix, 2024) use a functional package management approach to achieve reproducibility, marking packages using cryptographic hashes to track dependencies uniquely and repeatably.

In Urbit, compilation means converting Hoon source code (as text) into Nock code (as a binary tree). This process is handled by `++rash`, `++mint:ut`, and other components of the Urbit build system.

Linking, in the Urbit sense, derives from supplying nouns to nouns at compile time.[1] In the Urbit build arm `++ford`, a pair of builds becomes the build of a pair. The subject (environment) used to build a file is the tuple `[import_n … import_2 import_1 stdlib]`. Since Hoon symbol lookup is left-to-right, this nests scopes seamlessly and predictably. The linking technique is essentially trivial.

Ford compiles a Hoon source file into a data structure called a "vase", a pair `[type noun]` where `noun` is a member of the set of nouns described by `type`. Linking is thence just calling the `++slop` funciton, a one-liner from a pair of vases to a vase of the pair.

For example, the (trivial) Hoon source file with text `'3'` compiles to the vase `[[%atom %ud ~] 3]`. `3` is the value. Its type is an atom (number), tagged as an unsigned decimal (`%ud`) for printing.

As another example, the Hoon source file with text `'[3 0x4]'` compiles to

```
[[%cell [%atom %ud ~] [%atom %ux ~]] [3 0x4]]
```

This is a vase of a `%cell` (pair) of unsigned decimal number (`%ud`) and unsigned hexadecimal number (`%ux`). If `/foo/hoon` is `'3'`, `/bar/hoon` is `'0x4'`, then importing both of those files changes the build subject to (a vase of) `[foo=3 bar=0x4 <stdlib>]`.

[1] This is perhaps the biggest distinction from the conventional scenario for linking, which refers to RAM words. *Ceteris paribus*, this article's discussion can inform the traditional dialogue.

This is a form of static linking. The linking is performed at build time, not at runtime, and the resulting program contains its imports. "Relocation pointers" in C correspond to adjusting tree slots in Hoon, which the Hoon compiler does for the developer. Because Urbit uses static linking, it long had the same problem static linking has always had: memory duplication. If two different apps imported the same library, that library would be built twice and two copies of it would exist in memory.

In Urbit, everything is a "noun" (a binary tree with arbitrarily-sized integers at leaves). If you "copy" a noun `foo`, like `[foo foo]`, the runtime just copies a pointer to it. They are immutable, so everything shares structure. Nouns are "persistent" data structures, like Clojure's collections. If one copies a library, the copy is merely a pointer to the library—the library is a *noun*. If an imported library can be looked up from a build cache, the builder can copy it into a new app's build subject without duplicating it in memory.

But how does one know whether the cached library is valid? To achieve global (cross-application) deduplication, one needs a referentially transparent build cache, i.e. a build factored as a full description of an input to the build system. Since the build system is deterministic, if one sees the same input, one knows that it will produce the same output.

With one referentially transparent cache for all builds in the whole system, no invalidation is necessary. Cache eviction can take place efficiently due to reference counting, since which revisions of which apps refer to which builds is known.

As of #5745, Urbit supplies such a referentially transparent build cache. What this means in practice is that Urbit can have the memory deduplication benefits of dynamic linking while still using static linking. Since every filesystem snapshot lives at a unique, immutable, authenticated path within Urbit's scry namespace, reproducible builds are possible on every node, a crucial feature for software supply chain security and reliable app distribution.

4 The Modern `++ford` Build Cache

The former Ford cache was per-desk,[2] keyed on the name of the build (e.g. a file at a certain path). It was impossible to share such a cache between desks because the name may refer to different things on different desks or at different revisions.

Since the caches weren't shared, they commonly held exactly the same data but generated independently. For instance, the same library used on different desks would be built repeatedly and the memory was not shared unless the user manually ran the `|meld` command to manually deduplicate nouns. For users with many apps installed, this added significant memory pressure.

A new cache was designed which is keyed on the name of the build plus its dependencies. This is all the input to a build except the standard library. Thus when one matches a key it does not matter which desk the value was on; the cache can be shared across all desks. When the standard library changes, all caches are cleared automatically.

Reclaiming space from this cache becomes important. Since this is a "true" cache, it's never incorrect to keep data in it. One could adopt a heuristic such as to clear hourly or so you could use a heuristic such as "clear the whole thing every hour" or a least-recently-used (LRU) policy. However, `++ford` has a long history of trying to use such heuristics and still using exorbitant amounts of memory. The primary innovation of the former `++ford` cache was that its size is deterministic, and it stored no builds unless they could be generated from the head of its desk (`%home` then `%base`).

These properties are extended to the global cache by counting references and maintaining a per-desk set of references to builds which are still relevant to the head of that desk. On every commit, the per-desk set of references is inspected to determine which have been invalidated. Invalidated references are freed in the usual manner—their refcount is decremented, and if it's now zero, then it is deleted from the cache and all of

[2] Clay desks in Urbit are like Git branches, describing particular filesystem continuities.

its immediate dependencies are freed.

An alternative would be to garbage collect, which could be done on every commit to maintain determinism. However, this scales with the size of the cache (thus, with the number of desks installed), whereas refcounting scales only with the desk in question. Additionally, the cache is acyclic, and no manual refcounting is required—there is precisely one place where references are gained and one where references are lost.

As a result, the current cache has a "least upper bound" property: first, it minimizes the number of rebuilds required; given that, it minimizes the amount of memory required. In other words, cache entries are thrown away only when they become irrelevant. An alternate approach would be a "greatest lower bound"—throw away any cache entries that the system is not certain to need. This uses a bit less memory but results in more rebuilds. (It is also a little more complex to implement, since it requires clients to "register" the builds they want to keep warm in the cache, even if those builds didn't become invalid.)

Generating a cache key for this cache can be slow—hundreds of milliseconds is not uncommon, and it scales with the number of transitive dependencies. To mitigate this, a per-desk cache is included isomorphic to the former cache system. This has sub-millisecond lookup speed, and remains well-understood.

The current procedure to perform a build is thus:

1. Check if the build is in the per-desk cache; if so, return it generate its dependencies.

2. Check if the build, with these dependencies, is in the global cache; if so, return it.

3. Else, build it.

In each case, the new build is added to either cache if it was not already present; and if it was not in the per-desk set of references, it is added there as well.

5 Conclusion

Developers have long sought to balance the flexibility and portability of static linking with the better system demands of dynamic linking. Despite care, the balancing act has led from well-managed mainframe code into the current linker spaghetti situation. Urbit's `+ford` build system elegantly solves for the linking problem by promoting structural sharing of objects (nouns) in memory and by utilizing a referentially transparent build cache across desks. This balances efficiency in code compilation and building with the reliability of solid-state computing. ⚛

References

GNU Guix (2024) "GNU Guix transactional package manager". URL: `https://guix.gnu.org/` (visited on ~2024.1.23).

Grimes, Richard (2003). *.NET and DLL Hell*. Dr Dobb's Blog.

Locklin, Scott (2022). *Managerial Failings: Complification*. Locklin on Science Blog.

NixOS (2024) "Nix and NixOS reproducible builds and deployments". URL: `https://nixos.org/` (visited on ~2024.1.23).

`~wicdev-wisryt`, Philip C. Monk (2022) "`ford`: rewrite cache to share more builds #5745". URL: `https://github.com/urbit/urbit/pull/5745` (visited on ~2022.5.3).