MirageOS 4: the dawn of practical build systems for exotic targets

Lucas Pluvinage, Romain Calascibetta, Rudi Grinberg, Anil Madhavapeddy,
and the MirageOS team

1 Introduction

MirageOS [1] is a library operating system that constructs *unikernels* for secure, portable and high-performance applications across a variety of cloud computing and mobile platforms. OCaml code can be developed on a normal OS such as Linux or macOS, and then cross-compiled into a fully-standalone, specialised bootable unikernel that runs under a hypervisor. The intention behind the project is that the vast majority of code that runs in production system should be written in type-safe OCaml, and also that unnecessary dependencies can be stripped out at build time in order to increase portability and reduce attack surface.

Since its origins in 2007, the MirageOS ecosystem has grown in breadth (the number of OCaml libraries available has tremendously increased) as well as in depth (the number of compilation targets). The initial targets for MirageOS were simply normal operating system userspaces and the Xen hypervisor. Today, it features support for compiling to more hypervisors (KVM, FreeBSD's vmm), and more architectures (the ESP32 microcontroller, to RISC-V) and with ever more minimal runtimes (sometimes not even requiring a boot layer).

For the MirageOS project, the biggest growing pain has been taming all this growth by ensuring that the build and packaging systems can effectively manage the new demands placed on it by all this flexibility. For example, cross-compiling to ESP32 chips requires using an OCaml compiler fork, and passing custom options to the underlying C toolchain, and using a homegrown linker to prepare the final firmware. Ideally, this extra complexity for that backend would not be exposed directly to end users.

The MirageOS 3 release in 2015 saw us integrate closely with the opam 1.2 package manager, and have a frontend tool that generates ocamlbuild descriptions that would take care of much of this complexity. Now in 2019, we are iterating towards the next generation of OCaml Platform tools which implement the desired functionality much more natively.

The new MirageOS 4 toolchain now uses the "dune"[2] build tool natively in order to support many advanced cross-compilation and development workflow features. This talk will introduce the dune features that improved build experience either for cross-compilation or general openness, and also explain how the features we contributed to dune will also help the general OCaml library ecosystem become much more portable and flexible to other platforms such as JavaScript and WebAssembly in the future.

2 Challenges

2.1 Variants, virtual libraries and default implementations

The *cmi linking hack* is an age old trick in OCaml for parameterizing libraries without using functors. The idea is quite simple:

- Define interface(s) by writing some .mli files. We dub these interfaces as the *virtual modules* and the library that contains these modules as *virtual.*
- *An implementation for a* virtual library is another library that defines an .ml implementation module for every virtual module in the virtual library.
- Generic library code is written against the virtual library.
- The selection of implementations (for the virtual libraries) is delayed until building executables.

The advantage is clear: we no longer have to commit to particular implementations of interfaces until we need to build a concrete executable. We can continue to write generic code for libraries and are only required to provide implementations when linking executables.

While the idea is clear, there are some non trivial difficulties in making this work in practice, and so we added

[1] `https://mirage.io`

[2] `https://dune.build`

first-class support for this into dune 1.7.0, instead of requiring manual linking rules as has been the case in the past. This cleared the path to having MirageOS libraries that had the same interface, but radically different implementations under the hook for a particular target platform (e.g. one using C code, another bindings to syscalls, another built with JavaScript stubs, another in pure OCaml). In the talk, we will describe the details of how variants work and provide some examples.

However each implementation in a project dependency tree had to be manually selected. Dune 1.9.0 introduces features for automatic selection of implementations. Variants is a tagging mechanism to select implementations on the final linking stage by searching through available implementations in the current scope of the dune dependency tree and supplying them to the OCaml linker.

We first implemented variants in dune 1.9, with a significant revision coming up to make the semantics clearer in dune 1.11. While building the final executable, a set of tags can be given to choose which libraries will implement interfaces. This makes life a lot easier for MirageOS developers who can write platform-independent code without understanding all the details of the various platforms it might run on eventually.

Examples of variants:

- `backend.{unix, xen, freestanding}` allow to choose the correct backend for target-specific libraries, such as mirage-net interface exposing a network base layer.
- `implem.{c, ocaml, js}`: allow to choose between portability and speed for an algorithm such as decompression

We will also describe in the talk who this has improved the MirageOS ecosystem significantly in terms of openness, since we decouple the process of creating a new variant interface and the implementations behind it.

2.2 Cross-compilation

A lot of work has been done recently to have MirageOS running on new platforms, such as RISC-V processors or ESP32 boards. Along with openness issues, cross-compilation is the main blocker for seamless use of Mirage in production. However dune allows to build OCaml and C code with the correct compiler or flags and switches seamlessly between host and target binaries according to their usage. Allowing multiple build contexts enables easy cross-compilation that doesn't require the port of hundreds of packages.

It also opens the door for multiple-target unikernel builds at once.

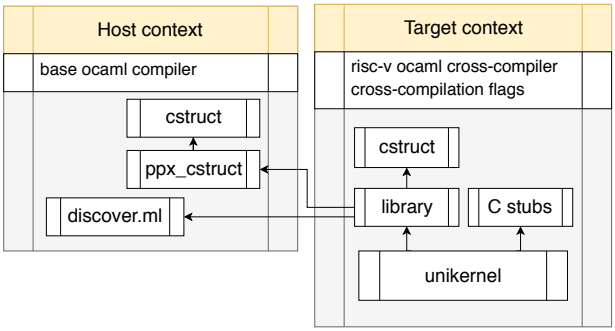

Figure 1: Having multiple build contexts is necessary to cross-compile OCaml projects.

→ All solved by dune build contexts in `dune-workspace` files.

In the talk, we will explain how a single dune project can now systematically compile OCaml projects (including with C stubs) to a variety of exotic hardware targets. Our mechanisms are more generally applicable to other users of OCaml who need to do embedded systems builds as well.