Networking in MirageOS

Fabian Bonk, Paul Emmerich*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: fabian.bonk@tum.de, emmericp@net.in.tum.de

***Abstract*—MirageOS is a modern library operating system written in the functional, memory-safe OCaml programming language. Users of MirageOS write application code in OCaml and link against various libraries provided by MirageOS. These include a complete network stack (Ethernet, IP, TCP, UDP, TLS) written in pure OCaml as well as a number of backends for receiving and transmitting packets. We introduce some of MirageOS' techniques for handling raw memory. We detail two of the various networking backends offered by MirageOS as well as a library used for safe abstraction over raw memory. We additionally suggest possible performance improvements in MirageOS. Finally we compare MirageOS with *ixy.ml*, a small userspace driver for *ixgbe*-compatible NICs written entirely in OCaml.**

***Index Terms*—library operating system, unikernel, OCaml, networking**

1. Introduction

MirageOS introduces the concept of unikernels: Application-specific, standalone, bootable virtual machine images designed to run on top of a hypervisor (initially Xen, nowadays also KVM) [1]. The hypervisor provides hardware abstractions and isolation between unikernels. Unikernels are configured at compile-time to target a specific hypervisor and only include code to support their exact required features. Unlike the typical VM deployment that runs few services on top of an entire operating system such as Linux (including Linux's filesystems, drivers, network stack, userspace, etc.), a unikernel only includes the code it requires, e.g. a static webserver includes only a network stack and its hardcoded webpages. Anything that isn't strictly required is not included in the final unikernel, which leads to typical image sizes of a few MiB.

Figure 1 compares a typical VM deployment and a unikernel deployment.

MirageOS provides these libraries for many backends including a standard UNIX backend that runs the unikernel as a normal process, a Xen backend, and a KVM backend (via Solo5). These backends all provide specific implementations for MirageOS' interfaces, including MirageOS' `Mirage_net.S` network interface. In the following we analyse the KVM and Xen implementations of the `Mirage_net.S` interface.

MirageOS is written in OCaml[1], a multi-paradigm programming language that supports functional, imperative and object-oriented programming styles. OCaml features

1. https://ocaml.org/

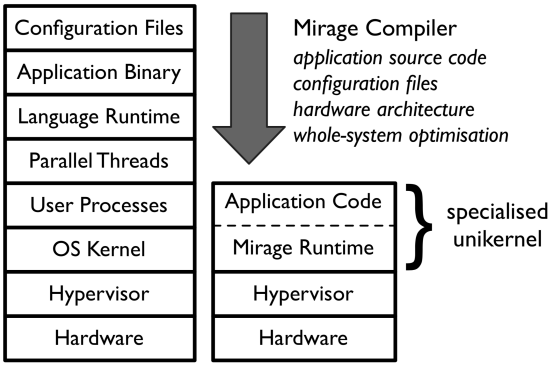

Figure 1: Typical VM deployment (left) vs. Unikernel deployment (right) [1]

```
module Main
  (N : Mirage_net_lwt.S)
  (C : Mirage_clock_lwt.PCLOCK) = struct
  let start n c =
    N.listen n (fun _ ->
        let now, _ = C.now_d_ps c in
        Logs.info
          (fun f ->
              f "got a packet at %d s!" now);
        Lwt.return_unit)
end
```

Figure 2: Example unikernel

memory safety, static type checking with type inference, garbage collection, and an optimizing native code compiler with support for multiple architectures. *Real World OCaml* [2] provides a good introduction to OCaml.

MirageOS users write their applications as OCaml functors. A functor is an OCaml module that is parameterized over other OCaml modules. In MirageOS' case the parameters are modules that implement functionality that is commonly provided by operating systems (network stack, file sytems, clocks, etc.).

Figure 2 shows a unikernel that simply prints a log message whenever it receives a packet. It is parameterized over a network interface (`N`) and a POSIX clock (`C`). At compile-time specific implementations for these modules must be chosen. The available choices depend on the runtime environment, i.e. which hypervisor or host OS will be used.

Section 2 introduces MirageOS' memory handling libraries. Section 3 explains MirageOS' network interface abstractions as well as two specific implementations of

doi: 10.2313/NET-2019-06-1_10

MirageOS network interfaces. Section 4 suggests some candidate performance improvements for MirageOS' network interface implementations. Finally section 5 compares MirageOS' network interface with ixy.ml, a network driver written in OCaml.

2. MirageOS Memory Handling

MirageOS needs to communicate with hypervisors using shared memory. Since OCaml natively only has limited support for accessing raw memory, MirageOS provides two libraries to handle page-aligned allocation and access in a safe way.

2.1. io-page

MirageOS provides *io-page* [4], a library for allocating page-aligned memory. io-page supports both UNIX and Xen backends (as well as Windows, though MirageOS itself doesn't support Windows). On Xen it uses Mini-OS' [5] `_xmalloc()` memory allocator. Mini-OS is a small kernel developed by the Xen project. MirageOS uses parts of it for CPU initialization, console output and memory allocation.[2] On other platforms (besides Windows) `posix_memalign()` is used for allocation.

2.2. cstruct

MirageOS uses a small wrapper library around C-like structures called *cstruct* [3] to facilitate safe and easy access to raw memory blocks. This library is split into the core cstruct library that manages the raw memory as well as a preprocessor called *ppx_cstruct* and the UNIX-specific library *cstruct-unix*.

2.2.1. cstruct. The main cstruct library defines an OCaml type `Cstruct.t` (referred to as cstruct from now on) that stores a reference to an OCaml Bigarray (which in turn references a raw memory region) as well as the array's length and an optional offset into the array.

The library includes functions for reading from and writing to these arrays in both little and big endian modes. Additionally cstructs can be converted to various other OCaml types such as `string`, `bytes` and S-expressions. Reads and writes are bounds-checked at runtime to ensure safety.

2.2.2. ppx_cstruct. ppx_cstruct is an OCaml ppx preprocessor that automatically generates accessor functions from C-like struct definitions (akin to LuaJIT's `ffi.cdef`).

Programmers simply declare the fields and types of `struct` and `ppx_cstruct` generates a number of functions for reading and writing each field as well as functions to hexdump an instance of the `struct`. Figures 3 and 4 show a UDP header definition and the values generated by ppx_cstruct respectively.

Additionally C-like `enums` can also be declared.

2. https://mirage.io/blog/introducing-xen-minios-arm

```
[%%cstruct
  type udp_header = {
    sport : uint16;
    dport : uint16;
    length : uint16;
    checksum : uint16
  } [@@big_endian]
]
```

Figure 3: cstruct UDP header declaration

```
val sizeof_udp_header : int
val get_udp_header_sport :
  Cstruct.t -> int
val set_udp_header_sport :
  Cstruct.t -> int -> unit
val get_udp_header_dport :
  Cstruct.t -> int
val set_udp_header_dport :
  Cstruct.t -> int -> unit
val get_udp_header_length :
  Cstruct.t -> int
val set_udp_header_length :
  Cstruct.t -> int -> unit
val get_udp_header_checksum :
  Cstruct.t -> int
val set_udp_header_checksum :
  Cstruct.t -> int -> unit
val hexdump_udp_header_to_buffer :
  Buffer.t -> Cstruct.t -> unit
val hexdump_udp_header :
  Cstruct.t -> unit
```

Figure 4: Values generated by `ppx_cstruct` from declaration in Figure 3

2.2.3. unix-cstruct. unix-cstruct wraps the `mmap(2)` system call and creates a cstruct by memory-mapping a file descriptor. The file descriptor is not mapped as shared (`MAP_SHARED`) but as private (`MAP_PRIVATE`), therefore writes to the cstruct are not reflected in the underlying file.

3. MirageOS Network Interfaces

The `Mirage_net` [6] module defines the module signature (interface) MirageOS programs use to send and receive packets. At compile-time a specific implementation that fulfills this signature must be chosen and linked into the unikernel.

While `Mirage_net.S` is an abstraction over network devices it itself leaves some implementation details abstract. All MirageOS backends actually implement `Mirage_net_lwt.S` which uses the Lwt library [7] for concurrency.

There are a number of different backends that implement this signature:

- mirage-net-unix [8]
- mirage-net-xen [9]
- mirage-net-macosx [10]
- mirage-net-flow [11]
- mirage-net-fd [12]
- mirage-net-solo5 [13]

doi: 10.2313/NET-2019-06-1_10

Sections 3.2 and 3.3 detail the hypervisor backends *mirage-net-xen* and *mirage-net-solo5* respectively.

MirageOS' network stack (layer 2 and up) simply calls into the network backend to communicate; the same network stack can be run on any backend.

3.1. `Mirage_net.S`

A MirageOS network interface must support these core functions:

- `write` transmits a single packet
- `writev` transmits a list of buffers concatenated into a single packet; this is generally implemented by concatenating the buffers into a freshly allocated, larger buffer and then transmitting this buffer
- `listen` calls a specified handler function for every received packet

The underlying implementation of the module signature may specify the type of packet buffers, asynchronous I/O operations, device state, MAC addresses and allocation operations for new buffers.

In the case of `Mirage_net_lwt.S` packet buffers are cstructs, I/O operations are Lwt promises and allocation is done using MirageOS' *io-page* library (see Section 2).

The `Mirage_net.S` signature also requires a number of other functions such as disconnecting from a network interface (interestingly connecting to an interface is not required), retrieving the interface's MAC address as well as reading the interface's receive and transmit statistics (bytes/packets sent/received).

3.2. Xen

The Xen implementation of the `Mirage_net_lwt.S` signature is written entirely in OCaml. It communicates with Xen via the netfront/netback protocol[3].

mirage-net-xen's `listen` loop sleeps until an event is fired on the event channel associated with the specified network interface. Once an event is fired a new cstruct is allocated for each received packet and the packet fragments delivered by Xen are assembled into the cstruct. MirageOS' handler function is called for every packet.

When transmitting, packet data is copied into a shared memory page and a reference to the page is stored in the transmit ring.

Both receiving and transmitting packets requires a full copy of the packet data from/to a shared page.

3.3. KVM

MirageOS' KVM implementation is provided by Solo5. Solo5 is an execution environment for unikernels. It can be used to run MirageOS unikernels on Linux's KVM hypervisor. On Linux it can interface with both *virtio* and TAP network interfaces, though virtio support is no longer maintained.

3. https://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/include/public/io/netif.h

3.3.1. hvt. Solo5 provides a small hypervisor manager called *hvt* (hardware virtualized tender). hvt sets up a KVM virtual machine and runs a unikernel inside this machine. hvt connects to a TAP interface on the host operating system and forwards packets between unikernel and host.

3.3.2. mirage-solo5 and mirage-net-solo5. A unikernel interfaces with hvt via *mirage-net-solo5*, a small OCaml wrapper around *mirage-solo5* which in turn wraps hvt's hypercalls (hypervisor equivalent of a system call) and makes them callable from OCaml. OCaml cannot directly call C functions due to differing value representations. mirage-solo5 converts OCaml values to their C representation and vice versa.

mirage-net-solo5 implements MirageOS' `Mirage_net_lwt.S` signature.

The `listen` function repeatedly calls `solo5_net_read` function. If a packet has been received, it is written into a freshly allocated cstruct before MirageOS' handler function is applied to the buffer. If nothing has been received, the thread blocks until an I/O event is signaled by hvt. Note that the I/O event need not be a received packet; Solo5 currently offers no mechanism for waiting for specific I/O events.

mirage-net-solo5's `write` function calls `solo5_net_write`.

Both `solo5_net_read` and `solo5_net_write` simply call hvt's `hypercall_netread` and `hypercall_netwrite` which read from/write to hvt's TAP device.

4. Possible performance improvements

We identified some possible peroformance improvements for MirageOS' network interfaces.

4.1. Avoiding memory copies

MirageOS' Xen networking backend copies every single received and sent packet buffer to and from cstructs. Solo5 requires full copies to and from the host's kernelspace when sending and receiving packets respectively. Copying every packet's payload incurs performance penalties proportional to each packet's size, though it allows users to transmit any cstruct and keep any received cstruct forever. Additionally cstructs can be collected by OCaml's garbage collector; there is never any need for manual memory management.

Implementing "zero-copy" packet buffers will likely require modifications to MirageOS' APIs and its network stack.

4.2. Batching

MirageOS handles packets individually. Handling batches of packets could reduce per-packet overhead. Batching is a common pattern in high-performance networking toolkits such as the DPDK [14], Snabb [15] or ixy and its derivatives [16] (see Section 5). Implementing packet batching requires a modifications to MirageOS' `Mirage_net.S` signature and all of MirageOS' networking backends.

doi: 10.2313/NET-2019-06-1_10

4.3. Parallelism

MirageOS unikernels can only be run on a single CPU core at once. This limitation is imposed by OCaml's runtime. Once the *ocaml-multicore* [17] project reaches maturity, it may be possible for the Lwt library to upgrade its scheduler. Once Lwt supports parallelism it should require little effort to run MirageOS unikernels on multiple CPU cores, given that MirageOS' interfaces already support concurrency.

5. Comparison with ixy.ml

ixy.ml [18] is a userspace driver for *ixgbe*-compatible NICs (Intel 82599) written entirely in OCaml and targeting Linux machines. ixy.ml also makes use of cstruct.

5.1. Memory

ixy.ml does not use OCaml `lists` but rather stores all data elements in arrays. Arrays in OCaml are fast to traverse, and are mutable (i.e. can be modified in-place). Mutable state requires careful programming to prevent race conditions. Functional programming languages generally favor immutable data structures.

5.2. Packet buffers

ixy.ml uses Linux's *hugetlbfs* for memory allocation. It provides the `Ixy.Memory` module that allows users to create fixed size memory pools from which packet buffers can be allocated. These packet buffers contain cstructs that wrap part of a huge page (2 MiB page). A user accesses packet data directly in the hugepage through the cstruct library. This memory is ready for DMA (Direct Memory Access) and can immediately be read and written by the NIC.

ixy.ml requires explicit allocation and deallocation of packet buffers by the user due to the fact that packet data must be written to DMA memory. Packet data is never copied between buffers. Use-after-free cannot be detected, though should be avoided.

mirage-net-xen requires copying of packet data for both receive and transmit; users never directly write to the buffer read by Xen's backend driver and vice versa. mirage-net-solo5 triggers a copy of packet data between userspace and kernelspace when calling `read`/`write` on the TAP device's file descriptor.

Since packet data is always copied in MirageOS, the user may hold on to previously sent or received buffers. Therefore MirageOS provides more memory-safety guarantees than ixy.ml.

5.3. Receive/Transmit

All `Mirage_net.S` implementations only transmit and receive packets one at a time. ixy.ml implements batching; multiple packets are sent/received at once.

5.4. API

MirageOS generally implements I/O asynchronously. Most function calls that may block can be run in the background.

ixy.ml's receive and transmit functions do block though they never wait (unless explicitly told to and the NIC cannot keep up with the user program's transmit speed). Given that the NIC operates asynchronously there is never any need to wait. If there is not enough room in ixy.ml's transmit queue(s), any unsent packets are simply returned to the user program.

MirageOS' network functions don't require any manual memory management by the user. Any cstruct can be sent as a packet (assuming its size is within the MTU).

Figure 5 shows an implementation of a bidirectional layer 2 forwarder using ixy.ml's API. Initialization code has been omitted. See `app/fwd.ml`[4] in ixy.ml's repository for a full implementation.

Figure 6 shows an implementation of a bidirectional layer 2 forwarder using MirageOS' API. Initialization code has been omitted and errors will be ignored.

```
let forward rx_dev tx_dev =
  (* receive a batch of packets *)
  let rx = Ixy.rx_batch rx_dev 0 in
  (* transmit all packets *)
  Ixy.tx_batch_busy_wait tx_dev 0 rx

let () =
  let a, b = init_devs () in
  while true do
    forward a b;
    forward b a
  done
```

Figure 5: ixy.ml layer 2 forwarder

```
open Lwt.Infix

module Main
  (N_a : Mirage_net_lwt.S)
  (N_b : Mirage_net_lwt.S) = struct
  let start net_a net_b =
    Lwt.join
      [ N_a.listen
          net_a
          (fun frame ->
            N_b.write net_b frame >|= ignore)
        >|= ignore;
        N_b.listen
          net_b
          (fun frame ->
            N_a.write net_a frame >|= ignore)
        >|= ignore;
      ]
end
```

Figure 6: MirageOS layer 2 forwarder

4. https://github.com/ixy-languages/ixy.ml/blob/master/app/fwd.ml

doi: 10.2313/NET-2019-06-1_10

References

[1] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, J. Crowcroft, “Unikernels: Library Operating Systems for the Cloud,” SIGPLAN Notices, vol. 48, pp. 461-472, March 2013

[2] Y. Minsky, A. Madhavapeddy, J. Hickey, “Real World OCaml,”, https://v1.realworldocaml.org/, 2013

[3] MirageOS project, “ocaml-cstruct,” https://github.com/mirage/ocaml-cstruct, 2019

[4] MirageOS project, “io-page,” https://github.com/mirage/io-page, 2019

[5] Xen Project, “Mini-OS,” https://wiki.xen.org/wiki/Mini-OS, 2019

[6] MirageOS project, “mirage-net,”, https://github.com/mirage/mirage-net, 2019

[7] Ocsigen Project, “Lwt,” https://ocsigen.org/lwt/4.1.0/manual/manual, 2019

[8] MirageOS project, “mirage-net-unix,”, https://github.com/mirage/mirage-net-unix, 2019

[9] MirageOS project, “mirage-net-xen,”, https://github.com/mirage/mirage-net-xen, 2019

[10] MirageOS project, “mirage-net-macosx,”, https://github.com/mirage/mirage-net-macosx, 2019

[11] MirageOS project, “mirage-net-flow,”, https://github.com/mirage/mirage-net-flow, 2019

[12] MirageOS project, “mirage-net-fd,”, https://github.com/mirage/mirage-net-fd, 2019

[13] MirageOS project, “mirage-net-solo5,”, https://github.com/mirage/mirage-net-solo5, 2019

[14] Linux Foundation, “Data Plane Development Kit,” https://dpdk.org/, 2013

[15] Luke Gorrie et al., “Snabb: Simple and fast packet networking,” https://github.com/snabbco/snabb, 2012

[16] Paul Emmerich et al., “ixy-languages”, https://github.com/ixy-languages/ixy-languages, 2018

[17] OCaml Labs, “Multicore OCaml,” http://ocamllabs.io/doc/multicore.html, 2017

[18] Fabian Bonk, “ixy.ml,” https://github.com/ixy-languages/ixy.ml, 2018

doi: 10.2313/NET-2019-06-1_10