University of Glasgow | School of Computing Science

Safe, Correct, and Fast Low-Level Networking

Robert Clipsham

August 22, 2015

ABSTRACT

In current implementations of low-level networking stacks, performance is favoured over safety, security, and correctness of applications. Despite modern languages and abstractions being available for higher levels of networking stacks, these have so far been dismissed due to the stringent performance requirements. This paper proposes using ownership types with move semantics to introduce the same level of safety and abstraction which is expected at higher levels of the stack, without sacrificing the expected performance.

1. INTRODUCTION

There have been numerous attempts to design and implement safer and more robust ways to interact with network protocol implementations[13, 19, 11, 16], however all major implementations in use today are still written in C and C++. There has been a lack of adoption of these new methodologies for low level networking, for reasons including their ease of use and existing, legacy code bases. Perhaps the most pertinent reason for lack of adoption is performance — as hardware supports increasingly high throughput, every CPU cycle begins to count. Current methods to meet these performance requirements involve partially or completely bypassing the operating system, and, as a result, sacrificing many of the useful and important guarantees the operating system provides.

Modern systems programming languages provide a way to re-introduce many guarantees which have been sacrificed in the name of performance, without introducing unnecessary overhead. In this paper, I exploit ownership types, move semantics, and zero-overhead abstractions to introduce guarantees such as freedom from data-races and memory safety without garbage collection to low level networking. In section 3, I present a compiler plugin for the Rust programming language[7], a language which provides these guarantees, to introduce an embedded domain specific language (eDSL) to specify the layout of packets. Additionally, the extension can automatically generate much of the error-prone bit-manipulation typically required for packet manipulation, which is discussed in section 3.1.

To provide the necessary performance for the implementation, I utilise the netmap library[28], which a provides high performance networking API which may be utilised to provide higher throughput than offered by the operating system. How this is integrated into Rust, with a sensible and safe abstraction, is discussed in section 2. Benchmarks and performance metrics, showing the performance is comparable to using netmap with C are shown in section 4.

This paper provides three key contributions:

- An embedded domain specific language for the Rust programming language enabling simple and efficient packet manipulation, which takes advantage of the features discussed in section 2.
- An algorithm for converting between arbitrary length and offset, unsigned, network-endian values and host-endian values.
- Performance metrics demonstrating comparable performance with equivalent, unsafe, C code.

2. ABSTRACTIONS FOR HIGH PERFORMANCE NETWORKING

There are many libraries for low-level networking which offer superior performance to what the operating system can provide[2, 5]. These typically work in user-space, bypassing the operating system and providing direct access to hardware. This introduces a number of risks either in the form of sharing memory between multiple user-space processes and the kernel, or by allowing network interface cards direct access to arbitrary memory locations.

Whilst it is possible for trusted processes to interact safely with these APIs, it is not enforced, allowing a programming mistake to have unforeseen and unfortunate consequences. Modern programming languages, such as Rust, provide a number of useful features which make it possible to build safe abstractions for these libraries — these will be discussed in the rest of this section.

Memory safety is a guarantee that a particular program will not corrupt memory, nor read or write from invalid memory locations. This guarantee is provided by many popular languages such as Java and Python, but is usually enforced by convention in lower level languages such as C and C++. When programmers inevitably make mistakes, or overlook an important detail, this can lead to process crashes, incorrect behaviour, and even security vulnerabilities. This is particularly problematic if your process potentially has access to arbitrary memory locations via the network interface card, since it any potential issues in the program can lead to multi-process or whole system failures — not just individual processes.

The traditional means to introduce memory safety into a language is garbage collection — this is what Java, Python and Haskell all use. This introduces a (usually unpredictable) runtime overhead, required to find and collect dead objects. Additionally, more advanced algorithms for garbage collection may require precise type information to be available, or the ability to freely move objects in memory. These trade-offs are often considered unacceptable for systems programs, causing a lack of adoption in languages like C or C++, where only conservative collection algorithms are possible.

In C++, techniques such as Resource Acquisition Is Initialisation (RAII) and smart pointers are typically preferred, which have a predictable overhead, but rely on convention for correctness.

Ownership types, combined with lifetimes[30], enable memory safety without using garbage collection, as well as guaranteeing data-race freedom — this makes it a useful concept for creating safe APIs. Under an ownership system, a particular piece of code is said to *own* a particular variable, meaning it has complete, unique control over its value — no other thread or part of the code may read or modify its value. When a function is called with an owned value, the ownership is moved to that function, and access is no longer permitted in the calling function.

In many situations, functions do not require ownership of data, only a view of it. In these cases, values may be borrowed, either mutably or immutably by other parts of a program. There may only be a single mutable reference to any given variable at once, or alternatively many immutable references. It is not possible to have both mutable and immutable references to a variable at the same time. Values are said to be borrowed for a particular lifetime. A lifetime is typically delimited by a code block. A value borrowed within a block is said to have the lifetime of that block, and will no longer be valid once the block of code is terminated. This can be used to prevent dangling references, which would lead to memory un-safety.

Combining ownership with lifetimes enables memory safety without the overhead of a garbage collector, as well as enforcing data-race freedom — since only a single mutable binding to a variable can exist at any given time, threads cannot contend for access. Another interesting consequence of this design is that APIs can be designed to enforce correct access patterns — owning a value guarantees unique control over it.

In low-level networking applications, correct management of buffers is essential, and ownership types can enforce this. Once the given memory and data structures are correctly initialised, it is possible to use the type system to statically guarantee that incorrect operations do not occur. For example, an abstraction can be built such that end-user code never holds ownership of a memory block, only borrowed references. Since these references have a bounded lifetime, it is possible to know when it is safe to re-use buffers for new packets.

Rust is a modern systems language which offers ownership types with lifetimes, in addition to other useful features. In Rust, code is memory safe by default, however it is possible to perform unsafe memory manipulation when necessary — providing the exposed interface is logically safe. Since APIs for low-level networking are written in C and C++, it is impossible for them to expose a verifiably safe interface. It is for this reason that Rust provides a means to execute unsafe code. Unsafe code in Rust is delimited using the `unsafe` keyword. There are two key things to note about this approach — the only place unsafe memory operations can occur in a Rust program is within an `unsafe` block. This makes it simple to locate, audit, and verify any potentially dangerous behaviour. Secondly, code within an `unsafe` block *must* expose a safe interface to the code within.

Whilst it would be possible (if unsettling) to use unsafe APIs directly, the typical approach is to create a minimal, safe, wrapper around an unsafe API instead. In this manner, any unsafe operation may be constrained to a small, reusable piece of code.

There are many additional features of Rust which make it a sensible choice for writing abstractions for high performance, low-level networking. Traits (similar to Haskell's type classes[17] or Java's interfaces), enable abstract definitions of functionality, and provide a means to perform both static and dynamic dispatch of method calls. This allows the creation of zero-overhead abstractions, without resorting to duck-typing as C++ does. Rust also provides algebraic data types and pattern matching, which provide a simple and effective way of handling errors, amongst other things.

2.1 Building the Abstraction

In this section, I discuss how these features can be used to build a safe and efficient API for interacting with the netmap library. There are three key stages to ensuring safe interaction with the netmap API: lifetime management; sending packets; and receiving packets. Note that this API has been simplified for the purposes of this paper.

Correct lifetime management for netmap works the same way as any API which provides a file descriptor or other object which must be constructed before use and destructed upon completion. The RAII idiom is used to guarantee this — first, the file descriptor is initialised, and stored in a structure. The structure must have a destructor which can then run the necessary clean-up code once it is no longer in use. All operations on the descriptor must also be defined for the structure which wraps it — the descriptor should not be accessed directly.

An example of this idiom can be seen in listing 1. A structure, `NmDesc`, is defined with a single, private member containing a mutable pointer to the file descriptor. There is a constructor method, `new()`, which takes the name of an interface as a string, and converts it to a C string (appends a null byte). The descriptor is initialised, with an unsafe block denoting a potential lack of memory safety within. If initialisation succeeded, we return a new structure representing the descriptor, otherwise, we return the error which occurred whilst constructing it. Finally, the `Drop` trait is implemented, to define the behaviour to occur once the structure goes out of scope. Note that all unsafe interactions are confined to two small blocks which are easy to find and verify, and the file descriptor will always be closed once the `NmDesc` structure goes out of scope.

Netmap provides access to a number of ring buffers which are directly accessible by the network interface card, and programs using the library. Writing directly to these buffers enables zero-copy construction of packets, providing the ability to quickly send and receive packets. While the process for sending packets is not inherently unsafe, it is difficult to get right, and this functionality can be encapsulated, whilst providing a more correct interface.

An overview of how this is achieved is shown in listing 2. By using a callback, which uses static dispatch, and will likely be inlined, we can provide the user with a way to construct packets in place, with only borrowed access to the buffer — this access will cease once the callback has completed. This also allows high performance, whilst abstracting the more complex logic of how exactly packets are sent into library code. The function is designed to enable a large number of packets to be sent with the minimal number of

```
struct NmDesc {
  desc: *mut nm_desc
}
impl NmDesc {
  fn new(iface_name: &str)
    -> io::Result<NmDesc> {
    let c_name = CString::new(iface_name);

    let fd = unsafe {
      nm_open(c_name.unwrap().as_ptr(),
              ptr::null(),
              0,
              ptr::null())
    };

    if fd.is_null() {
        Err(io::Error::last_os_error())
    } else {
        Ok(NmDesc { desc: fd })
    }
  }
}
impl Drop for NmDesc {
  fn drop(&mut self) {
    unsafe {
      nm_close(self.desc);
    }
  }
}
```

Listing 1: An example of using RAII in Rust to enforce correct the correct lifetime for a file descriptor.

```
impl NmDesc {
  pub fn build_and_send<F>(&mut self,
                           num_packets: usize,
                           packet_size: usize,
                           func: &mut F)
    -> io::Result<()>
      where F : FnMut(MutEthernetPacket)
  {
    let pollfd = self.get_poll_fd();
    let mut idx = 0usize;
    while idx < num_packets {
      if unsafe { poll(&mut pollfd, 1, -1) } {
        return Err(io:Error::last_os_error());
      }
      for i in (*self.desc).first_tx_ring ..
               (*self.desc).last_tx_ring + 1 {
        if idx >= num_packets {
          break;
        }
        let buf = unsafe {
                    self.get_buf_for_ring(i)
                  };
        let mep = MutEthernetPacket::new(buf);
        func(mep);
        unsafe {
          self.advance_pointers();
        }
      }
    }
  }
}
```

Listing 2: An overview of how the abstraction for sending works.

system calls. By specifying the `num_packets` parameter to be a larger number, as many packets will be sent as there are slots available for them, before resorting to a system call to send the packets and free up the buffers. Many implementation details are omitted in this example, and the implementation has been restricted to Ethernet packets — a trait could be used to support other packet types.

Receiving packets works in a similar manner to sending packets, however it provides an external iterator as an interface, rather than using a statically dispatched callback. A `next()` method is provided to get the next packet. The most interesting thing to note is that the lifetime of packets returned by the `next()` method of the iterator is only valid until its next call — this enables the buffers to be released to netmap to receive more packets immediately once they are no longer needed, and there are guaranteed to be no references to the packets remaining.

3. AN EMBEDDED DOMAIN SPECIFIC LANGUAGE FOR PACKET PROCESSING

Domain specific languages (DSLs) are languages which provide simpler or more expressive ways to describe functionality for specific purposes than typical general purpose languages. There have been many attempts at designing domain specific languages for packet processing, which have a number of desirable features, as well as a number of flaws[10, 21, 25, 23]. Each language introduces a different set of functionality, ranging from introducing type safety or simplifying protocol composition, however there are a range of issues including poor performance or being unable to express certain packet types. Some of these languages are discussed in more detail in section 5.2.

Poor performance is not a ubiquitous issue amongst DSLs, however there are many implementations of them which do not perform as well as equivalent hand-written C or C++. This causes the languages to have little appeal, particularly in scenarios where there are already existing C or C++ implementations — regardless of additional safety guarantees.

Many attempts to write DSLs involve a separate preprocessing step to generate code for the target language, or create an entirely new language to tackle the problem. This leads to poor integration with the host language, making their usage more intrusive to development. It is desirable to use a general purpose language, without reliance on an external tool.

Many DSLs define the structure of packets using a new and unusual syntax, which adds additional cognitive overhead for the programmer. Some DSLs have poor support for certain types of packet. If the DSL is unable to define arbitrary or common types of packets, then it can only be used some of the time and is of limited use. Additionally, some DSLs provide poor support for custom field types — it should be possible to define how to read and write custom values to and from the network, enabling the DSL to be extended.

To combat these issues, I have implemented a new, embedded DSL with the following features:

- Integrated into the host Language.

  The DSL uses standard Rust structure definitions to define the layout of packets. There is no need to learn a new syntax or language in order to specify the layout of packets.

- High performance.

  The DSL generates a lazy, zero-copy packet parser which is able to avoid many bounds checks which would usually be required for safety, by performing a single check upon initialisation. This enables packets to be parsed incredibly quickly, with no overhead for fields which are not read from or written to.

- Support for a large range of packet types.

  There are a number of features which enable many different packet types to be defined and parsed. In the event of an unusual field type which cannot be handled automatically, it is possible to easily extend the functionality using standard Rust code.

The DSL has been designed so that normal Rust structure definitions may be used, with some annotations. An example of this can be seen in listing 3. There are several things of note here, for example, the `#[packet]` attribute is specified before the structure definition, which indicates that the structure should be treated as a packet definition, and differentiates it from regular Rust structures. Additionally, the fields use custom integer types such as `u16be` and `u4`. These specify that the fields as sixteen bit unsigned integers, which is in big endian format when on-the-wire; and an unsigned four bit integer, respectively. There are additional attributes which are discussed in more detail in the following paragraphs.

Given a packet definition, such as the one in listing 3 the compiler plugin will generate a number of things. First, it will generate mutable and immutable structures which may be constructed using the underlying buffers which the operating system uses for sending and receiving packets. These provide a simple method for representing *on the wire* packets, without introducing unnecessary overhead. These structures contain a number of generated accessors and mutators which provide the necessary bit shifting for accessing each field in the packet. It is necessary to introduce both mutable and immutable structures, since Rust does not currently provide a way to abstract over the mutability of field types based on the mutability of the structure itself. Providing both allows simple and flexible interaction with the packets.

Additionally, a number of traits will be implemented for each of the structures. These provide a standard interface for packets and mutable packets; support for debug printing of packets; a way to convert on the wire packets to their `#[packet]` equivalents; and a way to determine the size of the packet. Finally, an iterator is created, to allow iterating through fields containing vectors of the packet type.

The eDSL provides a number of integer types whose names match the regular expression `u[0-9]+(be|le)?`. These are unsigned integers of the specified size, in either big endian (be) or little endian (le). These are simply aliases for the next largest integer type supported by Rust, that is `u8`, `u16`, `u32`, or `u64`. They serve as hints to the eDSL about how large the field is on the wire, and how to generate the necessary bit manipulations to interact with the value. Any values to be stored in these fields are host endian, removing the cognitive overhead of bit manipulation from the programmer. If a value greater than the maximum for that type is stored, the higher bits will be dropped when converting the value to wire format (for example, attempting to store 255 in a `u7` would result in 128 on the wire). This is an unfortunate trade-off, discussed in detail in section 3.2.

[1] `https://github.com/libpnet/libpnet`

```rust
#[packet]
pub struct Ipv4 {
  version: u4,
  header_length: u4,
  dscp: u6,
  ecn: u2,
  total_length: u16be,
  identification: u16be,
  flags: u3,
  fragment_offset: u13be,
  ttl: u8,
  #[construct_with(u8)]
  next_level_protocol: IpNextHeaderProtocol,
  checksum: u16be,
  #[construct_with(u8, u8, u8, u8)]
  source: Ipv4Addr,
  #[construct_with(u8, u8, u8, u8)]
  destination: Ipv4Addr,
  #[length_fn = "ipv4_options_length"]
  options: Vec<Ipv4Option>,
  #[payload]
  payload: Vec<u8>,
}

fn ipv4_options_length<'a>(
     ipv4: &Ipv4Packet<'a>
   ) -> usize {
  ipv4.get_header_length() as usize - 4
}
```

Listing 3: A possible implementation of an IPv4 packet definition using the DSL. The definition of `Ipv4Option` and `IpNextHeaderProtocol` have been omitted — these can be found in the source code for libpnet[1]. The definition for Ipv4Addr can be found in the Rust distribution.

There are a number of attributes for fields within packets. A field which is marked with the `#[payload]` attribute indicates that the following field can be treated as the packet's payload, and will be used for chaining packets together. The field must always have a type which is a vector of unsigned, eight bit bytes — this matches the buffers which are provided by the operating system.

The `#[length_fn]` attribute enables support for arbitrary variable length fields. A function name is given as an argument for the attribute, and when calculating field offsets this method will be called, along with an immutable reference to the packet. The function should return the number of bytes which the field will use.

The `#[construct_with]` attribute enables arbitrary, user-defined field types to be used, and specifies a list of primitive values which may be used to construct it. The specified type should provide a method named new which takes these arguments, and also an implementation of the

`PrimitiveValues` trait, which, given a value of that field type, will return a tuple of the same list of primitive values, representing the value. This enables the value to be read from, and written to wire format, without manually writing bit shifts. Elaboration on why this attribute is required is discussed in section 3.2.

3.1 Generating Bit Manipulations

Algorithm 1 Get a list of bit shifting operations to retrieve a number of bits at a given offset

Require: $offset \leq 7$ and $size > 0$
  function GET_OPERATIONS(offset, size)
    calculate the number of output bytes
    operations := []
    for $i = 0$ to *number of output bytes* **do**
      mask ← *calculate mask*
      lshift ← *calculate left shift*
      rshift ← *calculate right shift*
      append (mask, lshift, rshift) to operations
    end for
    return operations
  end function

As part of the DSL, I designed an algorithm to generate the necessary bit shifts to extract an arbitrary number of bits from an array of bytes, at a given offset. This is outlined in algorithm 1. Considering the case where we want to get a value, there are four key stages to calculate the correct bit manipulations. First, it is necessary to calculate the number of bytes in the output word. For a 32-bit integer this would be four, and for a 33-bit integer this would be five. We may then calculate the operations required to populate each of those bytes.

First we generate a mask for the byte. This can be done by masking the most significant $\min(\textit{total size of bits}, 8)$ bits of the byte, starting at the current offset.

Next, we calculate the left shift for the byte. For the first byte of a 32-bit integer with an offset of zero, this would be twenty-four — we have a single input byte, but want to produce a 32-bit unsigned integer, so the value must be shifted into position.

Finally, we calculate the right shift for the byte. This will always be zero, except for the final output byte. If we have a twelve-bit integer with an offset of two, first byte must be shifted left ten places, and the second right shifted two. This is because the least significant bit of the input bytes will be two bits from the end of the byte, but must be the least significant bit in the output byte.

A structure is generated to contain each of these values, leading to a list of operations to perform to get a value. In order to set the value, three things occur: the left and right shift are reversed; the inverse of the calculated mask is used to save bits in the array; and a new mask is calculated to select bits from the input integer. This new mask is calculated by masking the n least significant bits of a byte, then left shifting that mask by the original left shift value, where n is the number of bits set in the original mask.

3.2 Challenges of DSL Implementation in Rust

Rust enables extensions to the language in two main forms — macro rules, and syntax extensions (also known as compiler plug-ins). The former uses a declarative language which matches tokens, and produces Rust code. They are useful for reducing boiler-plate code, or for producing eDSLs which have a direct correspondence with the source language. Syntax extensions provide a framework to directly manipulate the compiler's abstract syntax tree, and provide a far more powerful means of extending the language. Since it is necessary to keep track of state to calculate offsets, the eDSL is implemented as a syntax extension. Specifically, it is a decorator, meaning that it provides an attribute (`#[packet]`), which will expand to additional code.

There are a number of limitations to this approach.

First, syntax extensions which act as decorators are evaluated after parsing, but before type checking. Whilst this simplifies the compilation process, it introduces a number of problems for eDSLs. For primitive values, it is necessary to assume that types with the correct name are also the correct type — if the user defines their own types with the same names as those which are special-cased in the plugin, it will lead to type checking errors in generated code, and there is no way to prevent this. Additionally, it leads to (by necessity) making potentially incorrect assumptions about the source code. For example, the plugin assumes that user-defined field types will have a method named new, which takes a number of parameters — but this cannot be checked in advance of code generation, leading to confusing error messages. It also introduces a number of unfortunate attributes to the eDSL. For example, there is a `#[construct_with()]` attribute, which allows the user to specify what parameter types are required to construct a user defined field type. This is necessary, since the type cannot be inspected to deduce this information. This is also the reason for using type aliases for primitive values, rather than a structure which enforces values to be within the bounds of the wire-format integer type. This limitation could be partially overcome by implementing a lint pass which utilises value-range propagation[27] to catch many invalid values at compile time.

Another problem with this approach is that syntax extensions are not part of the stable subset of the Rust language. While they are widely used to implement a large variety of functionality, including Quickcheck[12] style property based testing[2], structure serialisation and de-serialisation[3], and compile-time compiled regular expressions[4], they are subject to unannounced breaking changes to the API, and will not work on the stable channel of the Rust language. This means that users of the stable language will not currently be able to use the eDSL, and the eDSL may sporadically stop working for users of the nightly versions. Since the API is not stable, there are many parts which are overly verbose, or not implemented in the most coherent or consistent manner.

There are other approaches which could be taken to this problem. For example, the syntex project[5] enables syntax extensions to operate as a pre-processing step during compilation. It provides a stable, versioned API to prevent breakage, and provides the same interface as the compiler. It does, unfortunately, introduce an extra step into compilation, and still does not provide type information. To properly solve this problem, it would be necessary to modify the

[2] `https://github.com/BurntSushi/quickcheck`
[3] `https://github.com/erickt/rust-serde`
[4] `https://github.com/rust-lang/regex`
[5] `https://github.com/erickt/rust-syntex`

Rust compiler to allow syntax extensions to have access to type information.

4. EVALUATION

There are two parts to this evaluation — the improvements made by introducing an embedded domain specific language, and the performance of the whole system, from the eDSL down to netmap. These are discussed in the following two sections.

4.1 Domain Specific Language

Creating an embedded domain specific language for describing packets has proven incredibly valuable, enabling better expression of packet types, and finding bugs in existing, manually written packet implementations. In libpnet's definition of IPv4 packets alone, transitioning to the eDSL uncovered a bug in the manually written bit manipulations for setting the DSCP field; and enabled a more accurate representation of the packet, allowing the IPv4 options field to be simply expressed, where it was ignored before due to implementation difficulty.

Additionally, introducing an eDSL dramatically reduced the amount of code required to implement different packet types. Table 1 shows the code size reduction in lines of code once the DSL was introduced. This includes supporting code, such as that required for calculating checksums, as well as documentation comments. Packet types defined using the eDSL used on average 2.66 times less code for the existing packet types defined in libpnet. There was a 2.91 reduction in the code required for IPv4 packets, despite additional functionality being introduced in the conversion.

	Original	DSL	Reduction
Ethernet	223	87	2.56×
IPv4	393	135	2.91×
IPv6	323	92	3.51×
UDP	394	240	1.64×

Table 1: Code size (in lines of code) for implementing different packet types before and after introducing the DSL, along side their reductions.

There are, unfortunately some limitations with the current implementation of the DSL. For example, it is unable to express Real-time Transport Protocol (RTP)[29] packets, since they have a field pinned to the end of the packet, regardless of the length of previous fields. The DSL could be extended to support this by introducing a new attribute to pin fields to the end of the packet, rather than relying on an already calculated offset.

4.2 Performance

To evaluate the performance, a range of tests were conducted. The tests took place using two machines running FreeBSD 10.1 — one with 64x AMD Opteron 6274 cores and 500GB of RAM, the other with 8x Intel Xeon E5–2609 cores and 64GB RAM. Both machines were equipped with Intel X540–T2 10 gigabit Ethernet cards, with flow control disabled. Each port on the machine was connected directly to another 10 gigabit Ethernet port on the other machine. To get a baseline level of performance, the netmap tool *pkt-gen* was used to send and receive packets, ranging from minimally sized packets of 60 bytes, to 250 bytes. The raw performance in packets per second was recorded at each packet size. These tests were then repeated using the equivalent Rust code, utilising the eDSL and the netmap backend.

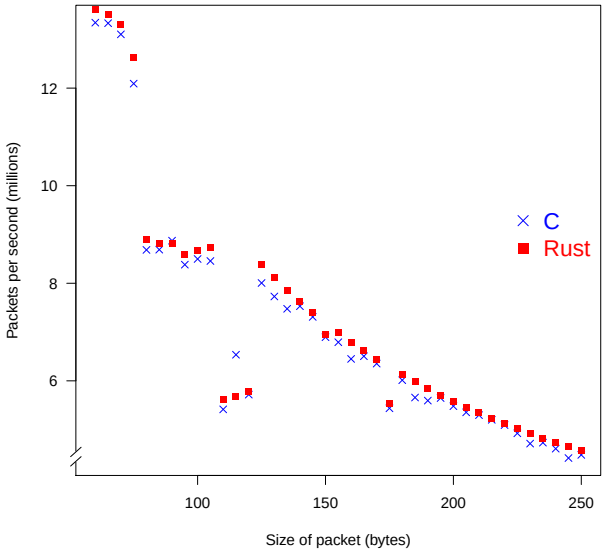

Figure 1: A graph showing the performance in packets per second when receiving differently sized packets.

Figure 1 shows the results of these tests. Both *pkt-gen* and the Rust equivalent show similar performance, exhibiting the expected $1/size$ behaviour, as well as an expected drop in performance early on, which is caused by the network interface cards being unable to efficiently handle packets which are not multiples of 64 bytes for small packets. This shows that the Rust code, which provides guarantees about memory and thread safety, as well as removing much of the need to do manual bit manipulation, is sufficiently performant for high-performance, low level networking. Whilst the graph shows that the Rust code is slightly faster than the C code, this is unlikely to be significant, and could be due to different optimisation passes being used by the compiler. Both applications were compiled using the same LLVM backend, with *pkt-gen* being compiled using `clang -O3`, and the Rust code using `cargo build -release`. It is worth noting that neither implementation manages to reach line rate — this is a limitation of the test setup, since Netmap is able to provide such speeds. The results show that Rust should be able to perform at line rate, given the correct set up.

5. RELATED WORK

There has been significant work completed in the area of network protocol implementation and high performance networking, some of the more relevant work is discussed here.

5.1 Systems Languages

There are many languages which could be suitable for implementing safe abstractions over networking APIs, each with different advantages and disadvantages. Due to the need for high predictable performance, the language must compile to native code and provide good control over the hardware. There are many potential candidates for this,

including D[9], Nim[3], Swift[6], and Rust.

The D programming language provides powerful compile-time meta-programming support, with access to full type information, making it a strong candidate for eDSL implementation. Additionally, it supports user defined attributes, without extending the compiler. It relies on garbage collection for memory safety, however it provides a `@nogc` attribute which would allow an API to guarantee that collection does not happen during critical parts of the code. Like C++, it provides smart pointers which may be used to emulate ownership semantics, but these cannot be statically verified and incur a runtime overhead.

Swift is currently only supported on OS X, where high performance APIs for low level networking only work on Linux and FreeBSD. It uses automatic reference counting, and require the programmer to explicitly mark weak references in order to break cycles, and prevent memory leaks. Since there is no explicit collection, it makes it viable to create abstractions over APIs which require high performance, however it does not offer memory safety, nor thread safety.

Nim, like D, offers powerful compile-time meta-programming. It uses a non-tracing garbage collector, which allows it to support soft real-time systems, however memory safety is not enforced. It provides strong support for implementation of DSLs, by providing the ability to directly manipulate the abstract syntax tree.

Rust was chosen for this paper, since it enforces memory and thread safety through ownership types, does not require garbage collection, and allows extensions to the language through compiler plug-ins. There are a number of limitations to this approach which have already been discussed — perhaps indicating that introducing ownership types to another language, or introducing more powerful meta-programming to Rust could be beneficial.

5.2 Domain Specific Languages

There have been many different attempts to implement domain specific languages for network protocol implementations. Despite the large variety and strong guarantees often provided, none of these have gained mainstream adoption, and all major network protocol implementations still utilise C and C++.

An early example of an eDSL for networking is FoxNet, which extends Standard ML with support for primitive types and continuations. It uses a similar design to the x-kernel[20], which uses a single interface for all protocols, allowing simple protocol composition. This technique is powerful, and is partially emulated in the design of libpnet, which this paper builds upon. FoxNet's main weakness is that it performs poorly (four to ten times slower) in comparison to the x-kernel, though this is not necessarily an inherent property of the design.

The Prolac protocol language takes the approach of designing a complete, object-orientated language from scratch to enable protocol implementation. It uses an unfamiliar syntax, and includes many mis-features, which can code to silently break as it is extended. For example, it contains implicit methods, which can cause code to silently break as it extended. It has a focus on zero-overhead abstractions, and manages to give similar performance to the Linux TCP stack of the time.

PacketTypes is a packet description language which compiles to C, enabling type safe packet definitions for C. It has a syntax similar to BNF, and can offer similar performance to hand-written C. Unfortunately it has limitations which mean it cannot express many types of packet — in particular those which require custom de-serialisation support. The DSL in this paper avoids this issue by allowing custom field types to be defined, extending the capabilities as necessary for arbitrary protocols.

The Meta Packet Language (MPL), is an DSL for packet definitions in OCaml, Java, and Erlang. The authors claim that they are able to produce better performing code for equivalent implementations of the SSH and DNS protocols, whilst using less code and providing type safety. MPL is possibly the most similar DSL to the one presented in this paper, and offers useful functionality such as being able to define variations of a packet in the definition. This is in contrast to the DSL in this paper where variations of a packet would be handled using additional packet definitions. There are advantages and disadvantages to both methods, and it is not clear which is more useful.

One notable paper is *Linear types for packet processing*[14], which exploits linear types as a mechanism for programming high-performance networking hardware. It introduces the PacLang language, and shows that linear typing is a viable method of implementing network protocols, as well as making it easier to reason about what the code is doing.

The PADS/ML data description language[15, 24] is designed for the processing of ad-hoc data, not just network protocols. It offers a syntax similar to Haskell's Parsec library[22], and supports design-by-contract, enabling predicates for error handling to be specified. It is not particularly well suited to parsing packets, since it lacks the ability to parse fields which are smaller than a byte in size, requiring the user to manually shift bits to get the desired values from a packet. There are a number of modern solutions such as Apache Thrift[6] or Google's Protocol Buffers[7] which have similar issues.

5.3 High Performance Networking

Typical operating system APIs for low-level networking such as Linux's AF_PACKET[4], FreeBSD and OS X's Berkeley Packet Filter (BPF)[26, 1], and WinPcap[8], offer sufficient performance for gigabit networking, however due to the overheads of data copying and per-packet system calls perform poorly on faster networks. There have been a number of attempts to introduce faster APIs without these limitations, typically by moving the network stack into user-space.

Intel's Data Plane Development Kit (DPDK) offers performance of around 160 million minimum-sized packets per second. It achieves this by enabling direct access to the network interface card (NIC) from user-space (avoiding context switches, system calls, and unnecessary copies); and by requiring dedicated polling drivers to avoid system interrupts. Platform specific details are abstracted using an environment abstraction layer, and it supports both Linux and FreeBSD.

PF_RING Zero Copy (ZC) from ntop is a proprietary, closed-source solution for saturating multiple ten-gigabit links on Linux. It requires custom drivers to achieve this performance, though offers a degraded performance mode involving a copy for unsupported drivers.

[6] `https://thrift.apache.org/`

[7] `https://developers.google.com/protocol-buffers/`

Both DPDK and PF_RING ZC allow the NIC to write directly to arbitrary memory locations, which means care must be taken by the user to ensure that only valid memory locations for the current process are read from, and written to. This is incredibly unsafe, and abstracting these details as discussed in section 2 would be of great benefit for these APIs.

The PacketShader I/O Engine[18] is interesting, in that it is designed to offload packet processing to the GPU, and works directly with the IXGBE driver, rather than supporting multiple drivers. By exploiting the architecture of GPUs it enables massively parallel packet processing, achieving speeds of up to thirty-nine gigabits per second on commodity hardware.

The Netmap Library offers similar performance to DPDK and PF_RING ZC, though offers superior safety guarantees by leaving the operating system in charge of memory access, and using system calls to synchronise buffers. The additional cost of the system calls can be amortized over multiple reads and writes.

6. CONCLUSIONS

Utilising a unique type system, which offers a sensible way to introduce memory and thread safety to an unsafe API, along side RAII and lifetimes has shown to be a sensible and cheap way to exploit high performance networking libraries. Introducing a domain specific language to describe packets has significantly reduced the amount of code required to parse packets, whilst also abstracting details which are difficult to get right manually such as bit manipulation.

By presenting an embedded domain specific language, and building safe abstractions atop of unsafe APIs, I have shown that it is possible to provide high performance in implementing network protocols, without the need to sacrifice safety guarantees.

6.1 Future Work

There are several useful directions in which to extend this work. Introducing support for alternative high performance back-ends with different safety trade-offs such as DPDK or PF_RING ZC would be an interesting task, and would show if the guarantees given by uniqueness, memory safety, and thread safety extend to APIs which provide even fewer safety guarantees, or whether a different approach is needed.

It would also be useful to address some of the limitations of the DSL — either by utilising another language, introducing a pre-processor, or improving Rust's support for compiler plug-ins. Another task would be to implement additional packet types using the DSL to verify its flexibility.

Acknowledgments. I would like to thank Dr Colin Perkins for his invaluable feedback throughout the course of this project, without which this paper would not have been possible. I would also like to thank the members of the Rust community for their continued support and useful insights, and additionally the language developers for responding swiftly to issues found in the compiler in the duration of the project.

7. REFERENCES

[1] bpf(4). `http://www.freebsd.org/cgi/man.cgi?bpf(4)`.

[2] Intel Data Plane Development Kit. `http://dpdk.org/`.

[3] Nimrod Programming Language. `http://nimrod-lang.org/`.

[4] packet(7) - Linux manual page. `http://man7.org/linux/man-pages/man7/packet.7.html`.

[5] PF_RING ZC (Zero Copy). `http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/`.

[6] Swift - Overview - Apple Developer. `https://developer.apple.com/swift/`.

[7] The Rust Programming Language. `http://www.rust-lang.org/`.

[8] WinPcap - Home. `http://www.winpcap.org/`.

[9] A. Alexandrescu. *The D programming language*. Addison-Wesley Professional, 2010.

[10] E. Biagioni. A structured TCP in standard ML. In *ACM SIGCOMM Computer Communication Review*, volume 24, pages 36–45. ACM, 1994.

[11] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in standard ML. *Higher-Order and Symbolic Computation*, 14(4):309–356, 2001.

[12] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[13] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. *USITS*, 1:15–15, 2001.

[14] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *Programming Languages and Systems*, pages 204–218. Springer, 2004.

[15] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. In *ACM Sigplan Notices*, volume 40, pages 295–304. ACM, 2005.

[16] M. E. Fiuczynski and B. N. Bershad. An extensible protocol architecture for application-specific networking. In *USENIX Annual Technical Conference*, pages 55–64, 1996.

[17] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[18] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.

[19] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. Rekindling network protocol innovation with user-level stacks. *ACM SIGCOMM Computer Communication Review*, 44(2):52–58, 2014.

[20] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *Software Engineering, IEEE Transactions on*, 17(1):64–76, 1991.

[21] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the prolac protocol language. *ACM SIGCOMM Computer Communication Review*, 29(4):3–13, 1999.

[22] D. Leijen. Parsec, a fast combinator parser, 2001.

[23] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: creating a functional internet. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 101–114. ACM, 2007.

[24] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez,

and A. Gleyzer. PADS/ML: A functional data description language. In *ACM SIGPLAN Notices*, volume 42, pages 77–83. ACM, 2007.

[25] P. J. McCann and S. Chandra. Packet types: abstract specification of network protocol messages. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 321–333. ACM, 2000.

[26] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2. USENIX Association, 1993.

[27] J. R. Patterson. Accurate static branch prediction by value range propagation. In *ACM SIGPLAN Notices*, volume 30, pages 67–78. ACM, 1995.

[28] L. Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[29] H. Schulzrinne. Rtp: A transport protocol for real-time applications. 1996.

[30] D. Walker and K. Watkins. On regions and linear types. In *ACM SIGPLAN Notices*, volume 36, pages 181–192. ACM, 2001.