NetBlocks: Staging Layouts for High-Performance Custom Host Network Stacks

AJAY BRAHMAKSHATRIYA, Massachusetts Institute of Technology, USA
CHRIS RINARD, Massachusetts Institute of Technology, USA
MANYA GHOBADI, Massachusetts Institute of Technology, USA
SAMAN AMARASINGHE, Massachusetts Institute of Technology, USA

Modern network applications and environments, ranging from data centers and IoT devices to AR/VR headsets and underwater robotics, present diverse requirements that cannot be satisfied by the all-or-nothing approach of TCP and UDP protocols. Network researchers and engineers need to create highly tailored protocols targeting individual problem domains. Existing library-based approaches either fall short on the flexibility in features or offer them at a significant performance overhead. To address this challenge, we present NetBlocks, a domain-specific language, and compiler for designing ad-hoc protocols and generating their highly optimized host network stack implementations. NetBlocks DSL input allows users to configure protocols by selecting and customizing features. Unlike other DSL compilers, NetBlocks also allows network researchers to extend the system and add more features easily without any prior compiler knowledge. Our design and implementation employ a high-performance Aspect-Oriented Programming framework written with the staging framework BuildIt. We also introduce a novel Layout Customization Layer that allows "staging packet layouts" alongside the implementation, which is critical for getting the best performance out of the protocol when possible, while allowing the practitioners to maintain compatibility with existing protocol layers where needed. Our evaluations on three applications ranging across deployments in data centers and underwater acoustic networks demonstrate a trade-off between performance (both latency and throughput) and selected features allowing the user to only pay-for-what-they-use.

CCS Concepts: • **Networks** → **Network protocols**; • **Software and its engineering** → **Source code generation**.

Additional Key Words and Phrases: compilers, network-protocols, layouts

ACM Reference Format:
Ajay Brahmakshatriya, Chris Rinard, Manya Ghobadi, and Saman Amarasinghe. 2024. NetBlocks: Staging Layouts for High-Performance Custom Host Network Stacks. *Proc. ACM Program. Lang.* 8, PLDI, Article 166 (June 2024), 25 pages. https://doi.org/10.1145/3656396

1 INTRODUCTION

Since the inception of the World Wide Web, 95% of the network traffic is composed of TCP and UDP packets [39, 45]. These two protocols were developed for the global internet where they had to route packets across multiple independent networks, account for packet drops and corruptions, and deal with out-of-order arrival. These protocols' design and implementation have been heavily hand-optimized with many specialized libraries and algorithmic extensions. However, recent times have seen a massive paradigm shift that has forced network researchers to rethink protocol design.

Authors' addresses: Ajay Brahmakshatriya, Massachusetts Institute of Technology, USA, ajaybr@mit.edu; Chris Rinard, Massachusetts Institute of Technology, USA, crinard@mit.edu; Manya Ghobadi, Massachusetts Institute of Technology, USA, ghobadi@mit.edu; Saman Amarasinghe, Massachusetts Institute of Technology, USA, saman@csail.mit.edu.

ACM 2475-1421/2024/6-ART166
https://doi.org/10.1145/3656396

Network applications have permeated new domains like (*i*) IoT that are constrained by extremely low-power, (*ii*) underwater robotics that are constrained by extremely low bandwidth [25, 37] (*iii*) isolated dedicated networks for Machine Learning training where the overall performance also depends on the best utilization of the network bandwidth, and (*iv*) AR/VR applications that are driven by extremely tight latency guarantees.

These new domains and their environments offer vastly different network properties and constraints that cannot be met by legacy TCP and UDP protocols. For example, an underwater sensor network of robots communicating through acoustic signals with very limited bandwidth cannot afford the 42-byte overhead of IP/UDP network headers. At the same time, these applications don't require all features from the protocols like reliability, in-order delivery, check-summing, and congestion control, but only a subset depending on the application and physical network properties. Recent years have also seen massive innovations in network hardware, which offer microsecond scale latency and 100s of Gigabits of throughput. Naturally, the bottlenecks are shifting from the hardware limitations to the network protocol design and implementation [9]. The end-to-end latency depends not only on the number of bits transmitted but also on the number of cycles spent in host-side processing in implementing the protocol logic on the hosts and other network devices.

The logical solution is to create and deploy custom ad-hoc protocols tailored to the needs of both the applications and the environment. Unlike the all-or-nothing approach of TCP and UDP when it comes to features available, network application developers should be able to pick and choose features and their customization to get the best network performance while meeting the critical requirements of the applications. An example of such a custom protocol is Google's QUIC [34], which combines basic reliability on top of UDP with Transport Layer Security (TLS) to suit web applications. However, writing and optimizing custom protocol implementations is daunting, which has hindered the widespread deployment of such ad-hoc protocols. Even for the example of QUIC, the protocol implementation and maintenance require effort from hundreds of developers at Google. Furthermore, changes in deployment scenarios also require continuous development effort. For example, a custom protocol used in a network with 16 nodes, uses 4 bits to identify source and destination hosts. However, when this deployment is scaled to a network with 32 nodes, 5 bits would be required to represent the hosts which would completely require rearranging the headers to optimally pack the bytes while meeting the alignment and size requirements of the other fields.

This paper demonstrates that the need for ever-changing custom protocols can be met by using compilers to generate highly specialized and optimized protocol implementations. We present NetBlocks, which is a network DSL (domain-specific language) that allows *Protocol Feature Selection and Configuration* through a high-level specification while enabling *Performance Optimized Protocol Execution* through low-level C code generation. Most importantly, NetBlocks is built on top of the C++ staging framework BuildIt, allowing developers to easily *Implement and Extend Protocol Features* without any knowledge of compilers, unlike other DSLs. NetBlocks also extends the BuildIt framework by adding a novel layer for *Precise Customization of Data Layout* that allows optimizing the headers to support the selected features.

NetBlocks enables the custom-generated protocols to run on legacy switches and routers by allowing backward compatibility at the granularity of layers by restricting customization to specific fields and features. This allows for keeping parts of the layout and the implementation strictly compatible when needed and customized and optimized when possible. NetBlocks also reuses the same code generation machinery to create custom WireShark [15] plugins to dissect the ad-hoc protocol packets, improving debuggability and further boosting developer productivity.

This paper makes the following contributions:

- We introduce NetBlocks, a DSL for generating high-performance custom network stacks.

- We show how to use BuildIt [5] framework's staging abilities to support the aspect-oriented programming pattern with high performance.
- We extend BuildIt to add a fine-grained, programmable data layout customization layer to allow "staging layouts" alongside code.
- We show that Netblocks can generate extremely high-performance host network stacks compatible with best-in-class performance but for custom protocols.

Section 2 walks through two examples to motivate the need for custom protocols. Section 3 discusses extending staging from code to layouts with our novel Layout Customization Layer and a high-performance Aspect-Oriented design using BuildIt. Section 4 demonstrates the application of these techniques to our network DSL NetBlocks. Section 5 demonstrates the performance trade-off offered by such a customizable network DSL.

2 MOTIVATING EXAMPLES

In this Section, we present two scenarios that motivate the need for custom-tailored protocols and the performance trade-offs they offer. We compare important metrics such as latency and network header overhead for the custom protocols vs. the common TCP/UDP protocols and motivate the need for a network DSL compiler.

2.1 Video Conferencing Application

The past decade has seen a sharp rise in video conferencing applications and products due to a shift to remote and hybrid work environments. Video and audio conferencing experience depends heavily on the latency of the communication since even a small delay can cause a lag in the voice or the video. The application and the stack they use must be heavily optimized to minimize latency.

At the same time, these applications also have unique characteristics that legacy UDP/TCP protocols are not designed to handle. Firstly, video conferencing does not care about reliability and can tolerate a few packets being dropped. It can also tolerate a few bits in the packet payload being corrupted since these corruptions might lead to only minor distortions in the audio/video. Since the network exchange happens at a fixed rate, the application may not even care about congestion control. These requirements might convince the developer to lean towards a protocol like UDP which is very lightweight. However, video conferencing applications rely critically on some notion of in-order delivery. If some packets arrive out of order, it could lead to voice and video getting completely jumbled. The UDP protocol unfortunately has no notion of in-order delivery or even a connection. Even if the developer decides to use TCP, the notion of in-order delivery in TCP often holds packets back till they arrive in order further compromising latency.

Consequently, the developers must implement a custom in-order delivery mechanism on top of UDP which increases implementation complexity. Furthermore, if we peer at the problem carefully, we realize that even though this application is tolerant to corruption in the payload, it might not be tolerant to corruption of certain control headers in the exchange. To keep the overhead of computing and verifying the checksums, the developer would have to specify only a certain part of the headers and payload to be checksummed. Handling this manually at the application layer would further increase developer effort and implementation complexity.

The key requirement here is that the developer should be able to quickly experiment with different features and their performance to decide what fits their specific application needs. This is where our compiler NetBlocks steps in. Figure 2 shows three different protocol inputs to NetBlocks that progressively remove features from a complete TCP-like implementation. Protocol 1 has full reliability and in-order delivery and is almost overkill for the job. Protocol 2 disables reliability, uses a simpler version of inorder delivery, and restricts the checksumming only to the network

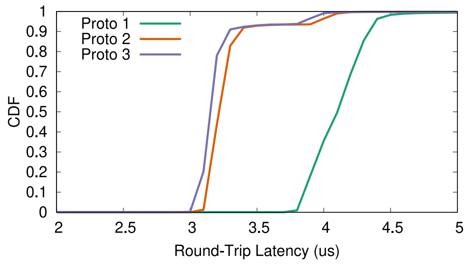


Fig. 1. Round trip latency for the three protocols described in Figure 2 for 256 byte packets.

```
// a) Schedule for checksumming the whole packet,
// enable reliable and inorder delivery
inorder_module.configInorder(HOLD_AND_DELIVER);
reliable_module.configReliable(ENABLE);
checksum_module.configChecksum(FULL_PACKET);
// b) Schedule for checksumming just the header
// disable reliability and drop out of order packets
inorder_module.configInorder(DROP_OUT_OF_ORDER);
reliable_module.configReliable(DISABLE);
checksum_module.configChecksum(HEADER_ONLY);
// c) Schedule for disabling checksumming, reliability
// and in order delivery
inorder_module.configInorder(NO_INORDER);
reliable_module.configReliable(DISABLE);
checksum_module.configChecksum(NO_CHECKSUM);
```

Fig. 2. NetBlocks DSL input for the three protocols with reduced reliability and checksumming

```
inorder_module.configInorder(NO_INORDER);
reliable_module.configReliable(DISABLE);
routing_module.configRouting(DISABLE);
identifier_module.setAppIdRange(0, 1);
```

(a)

```
identifier_module.setHostIdentifierRange(
  "aa:aa:aa:aa:aa:00", "aa:aa:aa:aa:aa:10");
payload_module.setLengthRange(0, 256);
checksum_module.configChecksum(DISABLE);
```

(b)

Fig. 3. NetBlocks DSL input to progressively shrink the number of bits used in the headers. a) removes the unnecessary and redundant fields from UDP + IP + Ethernet and b) Shrinks the ranges on the fields to use fewer bits. Figure 4 shows the packet layout for these protocols.

headers. Protocol 3 removes all notions of reliability, inorder delivery, or checksumming. Figure 1 shows the round-trip latency when these protocols are deployed. We can see that the Protocol 2 which is almost as performant as the bare-bones Protocol 3, has about 25% median lower latency than Protocol 1 which adds unnecessary features. With a carefully selected protocol, the developer only pays for what they need in terms of performance.

2.2 Underwater Robotics Sensing

As another motivating example, we look at a remote-sensing robot deployed underwater that gathers and sends sensor data to the base station. The key constraint of this environment is that the communication is done through audio waves traveling through water, unlike typical EM waves in the air. This unique environment characteristic means that the robot and the base station operate at very low bandwidth, typically tens to hundreds of bits per second. The network stack needs to minimize the number of bits transferred in every way possible not only to save network bandwidth but also to minimize the device's power utilization.

For an application where the robot gathers and sends sensor data, the payload size is typically as small as 16 bits. As the actual payload size is reduced, the overhead from packet headers starts to dominate. Even the simplest UDP protocol running on top of IP running on top of an Ethernet-like protocol requires a 42-byte header with many useless or redundant fields. Figure 4 a) shows the headers for the three protocols (Ethernet, IP, and UDP) stacked on top of each other. The fields in red correspond to bits that have no utility in this scenario. The fields in orange like length and checksum are redundant since multiple layers implement them. This is an example of overheads due to the independent development of protocol layers. A simple customization could strip these fields from the headers. Figure 4 b) shows such a minified protocol that requires only *16* bytes.

We can compress this further. In this header, 6 bytes are used to identify the source and destination MAC addresses of the robots. However, if our deployment has only 16 robots, 4 bits would be enough to identify all the hosts. Similarly, if there is a cap on the size of the payload, the number of bits for the size can also be shrunk, giving us a protocol shown in Figure 4 c) that only uses *2* bytes for the headers. Figure 3 shows the NetBlocks inputs to generate the two minified protocols.

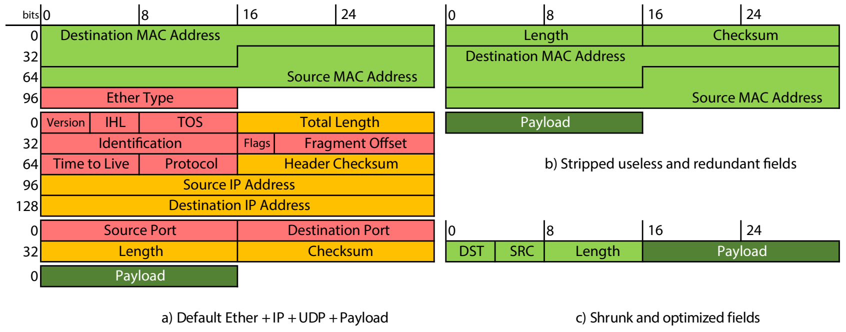


Fig. 4. a) Default Ethernet + IP + UDP headers with 50 bytes b) Protocol with useless and redundant fields stripped down c) Custom protocol with fields shrunk to fit the required deployment. 2 bytes of payload

Suppose our deployment changes and our fleet has 32 robots instead of 16. Consequently, the headers and protocols need to be adjusted to use 5 bits instead of 4. These could also be bumped to 8 bits instead of 5 for alignment reasons. Without a framework like NetBlocks, the developer would have to change these protocols with every new deployment manually.

2.3 Why a DSL Compiler for Customizing High-Performance Host Network Stacks?

The two motivating examples have shown that modern applications require custom host network stacks to meet their tight latency and bandwidth requirements. However, the design and development of these custom protocols remain challenging. The most straightforward approach to this problem is creating a modular and configurable network library that allows the user to compose different features. This approach uses the classic Aspect-Oriented Programming pattern that allows locally defined logic to control the global behavior of the whole application. However, such an approach suffers from several overheads when implemented in C or C++ without language-level source rewriting support. Firstly, since the configuration parameters like required features and ranges of values of individual fields are available as runtime values, these options need to be checked on the performance critical path. These checks and extra logic based on the parameters (like computing the bit-masks and shifts) add significant overhead for latency-critical applications. Similarly, features implemented as modules need to communicate data between each other and need queues and buffers to maintain modularity. These queues add a significant amount of performance overhead, and as a result, existing libraries like PicoTCP [19] and uIP [14] provide modularity and configurability only at the granularity of entire protocol layers as opposed to individual features.

A DSL compiler solves both these problems by generating code tailored for the configured options, eliminating expensive operations that can be done statically instead of at runtime. Furthermore, modules can be broken down into fine-grained features combined during code generation, eliminating the need for queues and expensive modular dispatching. However, conventional DSL compilers has a major drawback – they are not extensible. Network domain experts not only require creating custom protocols to combine features but also need to implement new features and algorithms. With a traditional DSL compiler, a network researcher would have to add a new compiler pass to implement a new feature or a variation of an existing feature like congestion control instead of writing the logic for the feature in a library.

Finally, a network DSL has a unique requirement uncommon for other domains – specializing packet layouts. Packet layout significantly changes with different features enabled/configured differently. The DSL would have to generate specialized code to pack and unpack binary data from packets that the modules can access. Doing this efficiently and in a configurable way requires

```
dyn<int> power(dyn<int> base, static<int> exp) {
    dyn<int> res = 1, x = base;
    while (exp > 0) {
        if (exp % 2 == 1)
            res = res * x;
        x = x * x;
        exp = exp / 2;
    }
    return res;
}
...
context.extract_function(power, "power_15", 15);
```

Fig. 5. Power function written in BuildIt

```
int power_15 (int arg0) {
  int var0 = arg0;
  int var1 = 1;
  int var2 = var0;
  var1 = var1 * var2;
  var2 = var2 * var2;
  var1 = var1 * var2;
  var2 = var2 * var2;
  var1 = var1 * var2;
  var2 = var2 * var2;
  return var1 * var2;
}
```

Fig. 6. Code generated for the Figure 5, exp = 15

carefully generating bit-masks and shift operations without having to rewrite a lot of implementation for a configuration change. We refer to this problem as staging data layouts. In summary, our network DSL compiler has the following capabilities:

- Rapid **Protocol Feature Selection and Configuration** for easily exploring protocol variations.
- **Performance Optimized Protocol Execution** via efficient C code generation.
- **Extending and Adding Protocol Features** without the knowledge of compiler internals.
- **Precise Control of Data Layout** to exactly match a protocol packet layout when compatibility is needed and optimize the layout when possible.

Furthermore, the DSL compiler should also be able to maintain backward compatibility with specific protocol layers like Ethernet and IP to allow the use of existing legacy hardware.

We build our network DSL compiler NetBlocks on top of the BuildIt [5, 6] multi-stage programming framework that allows us to generate efficient code while just writing library-like implementations automatically. Section 3 explains how we apply BuildIt's staging to aspect-oriented programming and introduce a novel layout customization layer for staging data layouts with code.

3 NETBLOCKS COMPILER FRAMEWORK

In this Section, we explain our compiler implementation on top of the BuildIt framework [5, 6] that meets all the four requirements we set out for NetBlocks in Section 2. We start by explaining the BuildIt framework and its staging abilities. We then explain the design and implementation of our novel Layout Customization Layer and finally explain our implementation of a high-performance Aspect-Oriented Programming framework.

3.1 The BuildIt Staging Framework

BuildIt is a type-based multi-stage programming library in C++. This choice of language is suitable for the NetBlocks domain since most network code is written in low-level imperative languages with pointers and explicit memory management. BuildIt introduces two type templates - `static<T>` and dyn<T> which respectively are used to declare variables for the first and the second stages. BuildIt completely evaluates all operations on `static<T>` in the first stage allowing specialization of the code generated with dyn<T>.

Figure 5 shows a simple power function implemented with repeated squaring specialized for a specific value of an exponent, 15, and the code it generates. The code in Figure 6 is faster due to it lacking any branches or loops. BuildIt's explicit code generation ability without writing a parser or a manual code generator is an instrumental tool in making compiler implementation accessible to network experts allowing them to **Extend and Add Protocol Features** just like they would for a library. At the same time, the specialization in BuildIt provides the much-needed **Performance Optimized Protocol Execution**.

3.2 Layout Customization Layer Language Definition

This section describes the Layout Customization Layer, one of the key contributions of this paper and the language that the module developers can use to define and optimize layouts, for exercising **Precise Control on Data Layouts**.

The Layout Customization Layer offers a flexible and easy-to-use for controlling layouts, embedded in C++, allowing a seamless interface with the rest of the system unlike alternatives like ProtoBuf which require generating or writing a separate DSL input. The API uses string values to identify and control fields in the layout allowing a lot of dynamism to the developer. However since the language uses BuildIt to stage the code, the string values and the look-up of fields and their properties is evaluated in the first stage generating precise and optimized code to be executed in the generated network stack.

Before describing the language API, we will describe the overall workflow for defining and optimizing a layout and using it to generate efficient code. We use the term *Layout* to refer to a specific collection of fields with fixed sizes, ordering, and alignment and describes how the data would be laid out in a buffer and can be used to generate code to read or write the fields. A *Layout* is similar to defining a `struct` in C or C++ code with the added benefit that the fields and their properties can be programmatically controlled in the compiler while still generating very efficient code. A specific *Layout* is held in an object of type `dynamic_layout`. Following are the steps to define and customize a layout.

- The developer starts by declaring an object of type `dynamic_layout`. The object starts as empty and the fields can be added along with their properties identified by a string name.
- To allow for modularity, the developer allows different modules to add the fields based on their configuration parameters.
- After all fields have been added, the developer finalizes the layout by choosing an optimization strategy. At this point, the order, sizes, and exact offset of each field have been decided.
- Finally during the actual protocol implementation, the modules can use this object to access the fields, again identifying them by the string names. At this point, the modules will also supply a dyn<T> buffer to enforce the layout on.

In the final step, the compiler generates specific C code to read or write the field based on the information stored inside the `dynamic_layout` object eliminating the overhead for looking up the fields based on the string names. This elimination is guaranteed to happen in the compiler since BuildIt's staging is precisely controlled with declared types instead of relying on opportunistic optimizations in a compiler.

Table 1 shows the two main types, the `dynamic_layout` and the `dynamic_member` which describe an entire *Layout* and a single field inside the *Layout* respectively. The table also shows all the member functions that the modules can call with their parameters. The type `dynamic_member` is an abstract type and needs to be specialized to describe different types of fields. We also show two such specializations namely the `generic_int_member<T>` which can be used to define integer-like fields of different widths and byte buffers respectively in Table 2.

The `dynamic_layout` type provides the `add_member` function which accepts a new member to be registered with the layout as well as a group ID to add it to. Each group can be optimized with a different policy allowing manual control over a set of fields when required. This feature is used in NetBlocks to maintain compatibility with existing protocols like Ethernet when running the generated protocol on legacy hardware like switches while optimizing the rest of the headers. The second key function is the `apply_policy` function that accepts a set of groups and optimizes them together. Groups that do not have any policies applied retain the order in which they are inserted. As mentioned above, `finalize_layout` computes the offsets and sizes of each field based on the size and alignment returned by each field. Finally, the `operator[]` is used to query a field from the layout for generating code. Since this API uses strings to identify fields, the function performs basic type-checking if the field has been inserted. However, this check is performed in the first stage and does not add any overhead to the generated code.

The `dynamic_member` abstract type provides three key virtual functions - `get_addr`, `get_integer` and `set_integer` that accept the buffer to apply the layout to and access the values. Notice that these functions use BuildIt's `dyn<T>` types for arguments and return types thus inling the logic into the generated stack. However, the virtual functions and computation of the bitmasks and offsets are performed in the first stage providing true zero-cost abstractions.

The `generic_int_member` is the most commonly used type for fields in NetBlocks since most fields like length, host identifiers, application identifiers, checksums can be viewed as integers. This type is templated on integer types like `short`, `int`, `long` for the logical size of the field. However, the precise number of bits required is determined by the specified range of values using the `set_range` function. This type also provides implementations for the three key accessor functions using BuildIt's `dyn<T>` types. The implementation is discussed in the next section. The other commonly used specialization for the `dynamic_member` is the `byte_array_member` type, which is used to represent long sequences of bytes, ideal for fields like payload. This type only provides the `get_addr()` accessor and a first-stage check ensures that the integer accessors are not called.

3.2.1 Optimizing Layouts. Once all the fields have been inserted, the developer can ask the Layout Customization Layer to rearrange the fields to minimize the amount of bits required while satisfying the alignment requirements of every field. Currently, the framework only supports optimizing the number of bits (`OptimizeWidth` policy), but similar optimizations can be implemented to optimize other metrics like the number of bitshifts or bitmask operations. Our implementation uses a brute-force approach to figure out the optimal packing order. We further improve the algorithm by reducing the permutations by identifying fields with the same size alignment and sizes. For typical network stacks, this is a common occurrence for fields like source and destination IP addresses and ports. For all our explored protocols, the algorithm terminates in less than a second. Even though this brute force search is exponential, it runs inside the first stage and does not hamper the runtime.

However, currently, there are two main limitations to our algorithm. Firstly, since optimization policies are applied on sets of groups, two groups cannot be co-optimized i.e. their fields cannot be interleaved if they are under different policies. For example, a group optimized for minimizing bit shifts and bit masking cannot have smaller fields from other groups packed into the padding. However, this can be addressed by manually changing the alignment of such fields to be byte-aligned and then optimizing all groups for size. Secondly, since the optimization algorithm uses a brute-force approach, the compile time can blow up for a very large number of fields. This can be addressed by grouping fields and optimizing separately, or by manually ordering some fields.

3.2.2 Nested Layouts, Unions and Optional Fields. The Layout Customization Layer allows generating code for complex nested layouts by specializing `dynamic_member` to hold other `dynamic_layout` objects. This allows accessing members inside members all optimized with different policies. Just like the scalar members, the exact offset of each field is computed at compile time avoiding any runtime overheads. Unions can be supported in the same way, where multiple `dynamic_members` can be mapped at the same position. Finally, our language also supports optional fields using BuildIt's `dyn<T>` types. For optional fields, the accessor can branch on arbitrary dynamic values to compute the size of the field and BuildIt will defer the branches to the generated code with the most optimized code on both sides of the branches. Variants can be realized by combining dynamic branches and Unions. Section 3.3 discusses the implementation of these specializations.

Table 1. Two core types of the Layout Customization Layer Language and their member functions.

Type Name	Function Name	Description
`dynamic_layout`	Constructor	Creates a new instance of a *Layout*.
	`void add_member(std::string name, dynamic_member*, int group)`	Inserts a new field described by the `dynamic_member` in the specified group.
	`void apply_policy(set<int> groups, Policy p)`	Apply an optimization policy to a set of fields together.
	`void finalize_layout(void)`	Finalize and compute offsets and sizes for each field.
	`void print_layout(std::ostream)`	Pretty print a layout with all fields, their sizes and offsets
	`dynamic_member* operator[] (std::string name)`	Retrieve a member using the name for access.
`dynamic_member`	Constructor	Abstract type, constructor is defined private.
	Destructor	Virtual to enable dynamic inheritance
	`dyn<char*> get_addr(dyn<char*> buffer)`	Return the address of this field given the base pointer
	`dyn<long> get_integer(dyn<char*> buffer)`	Read the field as an integer given a base pointer
	`void set_integer(dyn<char*> buffer, dyn<long> value)`	Write the field as an integer given a based pointer of a buffer and a value

Table 2. Two specializations of the `dynamic_member` type. The `byte_array_member` type does not support accessing the fields as integers and does not override the functions.

Type Name	Function Name	Description
`generic_int_member<T>`	`generic_int_member(int flags)`	`aligned` flag can be used to enforce alignment
	`void set_range(T min, T max)`	Accept a custom range of values for the field and shrink the number of bits required based on it.
	`dyn<char*> get_addr(dyn<char*> buf)`	Override for the virtual function
	`dyn<long> get_integer(dyn<char*> buf)`	Override for the virtual function
	`set_integer(dyn<char*> b, dyn<long> v)`	Override for the virtual function
`byte_array_member<N>`	`byte_array_member()`	Arrays are always aligned
	`dyn<char*> get_addr(dyn<char*> buffer)`	Override for the virtual function

3.3 Layout Customization Layer Implementation

In this Section, we explain the implementation of our novel Layout Customization Layer and how it extends BuildIt's staging capabilities to layouts. Our implementation allows the creation of network headers that are as small as 2 bytes. The main design goal behind the Layout Customization Layer is to allow modules to control various aspects of the fields including ranges, sizes, and alignment independently without having to change the implementation based on configuration in other modules while generating the most efficient code. We call this problem *staging layouts*.

Most network libraries solve the problem of defining the packet layout by declaring a C struct that has all the fields corresponding to the headers and reading and writing to them by casting the packet buffer to the struct type. However, such an approach offers little to no flexibility and is not programmable. Another approach to solving this problem is to use a specialized DSL like Protobuf [47] that allows users to define a layout in a separate configuration file and generate code for serializing to and reading data back from the layout. Since Protobuf is an entire DSL with its own syntax, modules would have to generate the DSL input collaboratively during the configuration phase. Not only is this approach cumbersome and requires module writers to learn a new language, but the options for configurability in protobuf are limited and do not offer fine-grained control over the fields as we provide.

To understand our implementation with the layout customization layer in NetBlocks, consider a simple example of a binary layout for inode metadata in a toy filesystem where we want to store metadata for files containing permission bits, size of the file, and the offset into the block where it is stored. Let us look at a code example of how the example would be written with NetBlocks's Layout Specialization Layer. Figure 7 shows the initial definition of `file_metadata` *Layout* in the first stage. The type `dynamic_layout` simply holds a map from the names of the fields (`std::string`) to the type `struct dynamic_member*`. To generate code, each field needs to know two properties - the size of the field and the offset of the field in the layout. The size of the field is defined nominally by the type of the field. For example, since the `is_writable` is a boolean, the size of the field would

```
nb::dynamic_layout file_metadata;
auto is_readable = new generic_int_member<bool>(aligned);
file_metadata.add_member("is_readable", is_readable);
auto is_writable = new generic_int_member<bool>(aligned);
file_metadata.add_member("is_writable", is_writable);
auto is_executable = new generic_int_member<bool>(aligned);
file_metadata.add_member("is_executable", is_executable);
auto size = new generic_int_member<unsigned>(aligned);
file_metadata.add_member("size", size);
auto offset = new generic_int_member<unsigned>(aligned);
file_metadata.add_member("offset", offset);
```

Fig. 7. Definition of the file_metadata layout in the layout specialization layer. The various fields are created and inserted.

```
bool* var0 = base + 2; // is_executable's offset is 2
var0[0] = true;
uint32_t* var1 = base + 8; // offset's offset is 8
var1[0] = file_offset();
```

Fig. 8. Generated code when the fields in file_metadata are accessed.

```
template <typename T>
struct generic_int_member: public dynamic_member {
  size_t get_size() override {
     return sizeof(T) * byte_size;
  }
  size_t get_offset() override {
    size_t off = prev->get_offset() + prev->get_size();
    return align_to(off, alignof(T) * byte_size);
  }
  dyn<long> get_value(dyn<bytes> base) {
    dyn<T*> addr = base + get_offset();
    return addr[0];
  }
  void set_value(dyn<bytes> base, dyn<long> val) {
    dyn<T*> addr = base + get_offset();
    addr[0] = val;
  }
};
..
file_metadata["is_executable"]->set_value(base, true);
file_metadata["offset"]->set_value(base, file_offset());
```

Fig. 9. Definition of the get_offset and get_size functions in the generic_int_member<T> type.

```
struct generic_int_member: public dynamic_member {
   size_t align = alignof(T) * byte_size;
   size_t get_offset() {
     if (flags & aligned) return align_to(off, align);
     else return off;
   }
};
auto size = new generic_int_member<unsigned>(aligned);
size->align = byte_size;
file_metadata.add_member("size", size);
```

Fig. 10. Definition of the get_value and set_value functions in the generic_int_member<T> type.

```
nb::dynamic_layout file_metadata;
auto is_readable = new generic_int_member<bool>();
is_readable->set_range(0, 1);
file_metadata.add_member("is_readable", is_readable);
auto size = new generic_int_member<unsigned>();
size->set_range(0, 4096);
file_metadata.add_member("size", size);
auto offset = new generic_int_member<unsigned>();
offset->set_range(100, 600);
file_metadata.add_member("offset", offset);
```

Fig. 11. Definition of the fields in `file_metadata` with specified ranges to reduce the number of bits.

be 1 byte. The offset of each field can be computed by taking the offset of the previous field adding the previous field's size and aligning it to the alignment of the current field. To facilitate this, the type `dynamic_member` declares two internal virtual functions `size_t get_size();` and `size_t get_offset();`. These functions return values in bits rather than bytes for more fine-grained control. Each field also has a pointer to the previous field initialized by the `finalize_layout` function. The implementation of `generic_int_member<T>` is shown in Figure 9.

Figure 8 shows part of the generated code when the fields are written to. As we can see, the generated code contains the simplest implementation with no virtual calls from the `dynamic_layout` or the `set_value`, or `get_offset` functions. This is because these are executed in the first stage.

In our example, the size field has an alignment of 4 bytes. While alignment is important for faster accesses on CPUs, it wastes bytes in binary protocols. We will start by allowing fields to have custom alignment instead of alignment tied to the type. Figure 10 shows the new parameter `align` in the `generic_int_member<T>` type initialized to the alignment of the type but can be set by the caller. The `get_offset` function now uses this and also checks the aligned flag. For this example, we can now set the alignment for the integers to be just byte-aligned. With this change, the layout is better packed and has no padding.

However, there are still more inefficiencies in our layout. The permission fields are all booleans but require a byte to store. These can easily be compressed into individual bits. Furthermore, if we know that the size of a file will never be larger than 4096 bytes, the size can be packed into 12 bits. The `set_range()` function computes the maximum required bits for representing the range. Furthermore, the `get_value` and `set_value` functions offset the values based on the range and use bit masking and shifts to store just the required bits. This way the calling code stays exactly the same and reads and writes the same values. The generated code takes care of mapping the

```
struct nested_member: public dynamic_field {
    dynamic_layout* nested_layout;
    nested_member(dynamic_layout* d): nested_layout(d) {}
    size_t get_size() override {
        assert(nested_layout->is_finalized);
        return nested_layout->get_total_size();
    }
    size_t get_offset() override {
        return prev->get_offset() + prev->get_size();
    }
    dyn<char*> get_addr(dyn<bytes> base) override {
        return base + get_offset();
    }
};
dynamic_layout child, parent;
child.add_member("mem1", new byte_array_member<16>());
child.finalize_layout();
parent.add_member("mem_nested", new nested_member(&child));
...
child["mem1"]->get_addr(parent["mem_nested"]->get_addr(p));
```

Fig. 12. Implementation of the `nested_member` type for enclosing layouts inside other layouts.

```
struct my_optional_field: public generic_int_member<int> {
    size_t get_size() override {
        if (packet_has_field)
            return generic_int_member<int>::get_size();
        return 0;
    }
    ...
};
layout.add_member("optional", new my_optional_field());
layout.add_member("next_member", new other_field());
layout.finalize_layout();
...
packet_has_field = true;
x = layout["next_member"]->get_integer(p);
// Generated code
if (packet_has_field)
    x = *(int*)(p + 4);
else
    x = *(int*)(p);
```

Fig. 13. Implementation of an optional field with a dynamic branch in the `get_size()` function along with the generated code.

bits to a value in the range. Figure 11 shows the creation of the fields with the updated ranges. Notice, we also remove the `aligned` flag from the constructor to pack the fields at the bit level. This optimization is why the sizes and offsets are at the granularity of bits to allow for better packing.

With this optimization, the entire layout can now be stored in a total of 24 bits as opposed to the initial 96 bits. NetBlocks uses these optimizations to shrink the number of bits required to represent IP addresses, port numbers, and sizes of packets among others to tailor the protocol for a deployment and application. While this layout specialization layer is developed for NetBlocks, the implementation is generic enough that it can be used for other applications as shown in the example above. These techniques have applications in file systems, databases, and compression.

Nested, Unions and Optional Fields. Figure 12 shows the implementation of `nested_member` which specializes `dynamic_member` to allow nesting of other layouts as members. The constructor for this type accepts a pointer to the nested `dynamic_layout` and uses the nested layout to compute the size of the field by overriding the `get_size()` function. Calls to the `operator[]` from both the layouts can be chained as shown to access the fields inside the child layout. For type checking, we assert in the first stage that the inner layout has been finalized before the outer layout. Figure 12 also shows how the nested layout is created and accessed. Figure 13 shows the implementation of an optional field. The optional integer field is exactly like the `generic_int_member<int>` it inherits from, except it has a branch of a dynamic value in the `get_size()` function. As a result, when any other field after this field is accessed, a branch appears in the generated code. This mechanism can be used to implement optional "Options" fields in the IP protocol headers. Variants can be implemented by combining these two techniques. Thus the Layout Customization Layer is flexible and extensible enough to support a variety of different layouts.

3.4 High-Performance Aspect-Oriented Programming

Aspect-oriented programming is a powerful programming pattern for creating modular and configurable abstractions. However, simple but modular implementation of aspect-oriented programming in C++ for fine-grained features has a lot of performance overheads. In this Section, we explain our implementation of an aspect-oriented programming framework on top of BuildIt to allow efficient code generation while being modular and configurable.

Aspect-oriented programming allows breaking down the logic into so-called distinct *concerns*. These concerns are allowed to augment the behavior of existing code without modifying the code itself. The simplest and most modular way of implementing aspect-oriented programming in a language like C++ is done through the use of virtual hooks that can be inserted into various paths.

```cpp
// Abstract class for all concerns
class Concern {
    virtual void hook_message (std::string request);
};
// Implementation of the logger concern
class LoggerConcern: public Concern {
    void hook_message (std::string request) override {
        log_stream << "Request␣=␣" << request << std::endl;
    }
};
// Implementation of the validation concern
class ValidateConcern: public Concern {
    void hook_message (std::string request) override {
        if (!parse_request(request)) {
            std::err << "Invalid␣request" << std::endl;
            abort();
        }
    }
};
// Register concerns
std::vector<Concern*> registered_concerns;
registered_concerns.push_back(new LoggerConcern());
registered_concerns.push_back(new ValidateConcern());
// Implement code path
void process_message(std::string message) {
    for (auto c: registered_concerns)
        c->hook_message(message);
}
```

(a) Apect-Oriented Programming in C++ for message processing system using virtual dispatches

```cpp
// Abstract class for all concerns
class Concern {
    virtual void hook_message (dyn<std::string> request);
};
// Implementation of the logger concern
class LoggerConcern: public Concern {
    void hook_message (dyn<std::string> request) override {
        dyn_log_stream << "Request␣=␣" << request << "\n";
    }
};
// Implementation of the validation concern
class ValidateConcern: public Concern {
    void hook_message (dyn<std::string> request) override {
        if (!dyn_parse_request(request)) {
            std::dyn_err << "Invalid␣request\n";
            dyn_abort();
        }
    }
};
// Register concerns
std::vector<Concern*> registered_concerns;
registered_concerns.push_back(new LoggerConcern());
registered_concerns.push_back(new ValidateConcern());
// Implement code path
void process_message(dyn<std::string> message) {
    for (auto c: registered_concerns)
        c->hook_message(message);
}
```

(b) Example in Figure 14a implemented with BuildIt dyn<T> types to generate efficient code.

Fig. 14. Two ways to implement Aspect-Oriented Programming - one that uses a library approach and virtual dispatches, another that uses BuildIt's staging for code-generation

Figure 14a shows a simple implementation of a message processing system with `Logger` and `Validate` concerns. An abstract class `Concern` is shown that declares a virtual method `hook_message`. The two concerns, `LoggerConcern` and the `ValidateConcern` that inherit from this class implement the hook functions to implement their logic. Concerns are registered based on the configuration. For instance, the `LoggerConcern` would be registered only if logging is enabled. Finally, in the actual code path for processing the messages, the hooks for the registered concerns are invoked. This pattern allows us to isolate all logic related to logging from validation while also conditionally enabling concerns without having to change the core implementation.

We observe that such a pattern maps very well to the modular network stack. All the features in NetBlocks like reliability, in-order delivery, and check-summing can be implemented as concerns that implement hooks. The implementation of the network stack also has several code paths like the logic to be run when a new flow is established or the logic to be executed when a packet is to be sent. The features implemented as concerns can define hooks for different paths to alter the behavior of these paths. Features can then be registered based on what features are required. However, such an implementation using virtual dispatches in C++ has huge overheads. The virtual dispatches often compile to indirect function calls that are harder for the hardware to execute speculatively and affect latency. Furthermore, configurations within features are implemented as conditions that are expensive to evaluate especially when run on performance-critical paths that run in tight loops.

```cpp
void process_message(std::string message) {
    log_stream << "Request␣=␣" << message << "\n";
    if (!parse_request(message)) {
        std::err << "Invalid␣request\n";
        abort();
    }
}
```

Fig. 15. Code generated from the implementation shown in Figure 14b without any virtual dispatches or conditions.

The common approach to mitigate this overhead using C++ templates greatly hampers the usability and productivity of network domain experts. We alleviate both issues by using BuildIt's multi-stage execution.

With BuildIt, we evaluate the virtual dispatches in the first stage to generate simplified overhead-free code for the second stage. Figure 14b shows the same example above with the messages turned into `dyn<T>` variables (changes highlighted in red). When this code is executed, we generate an implementation for `process_message` shown in Figure 15. The generated code does not have any virtual dispatches or even function calls but the logic for the hooks from the registered concerns is inlined in the generated code. In the same way, conditions based on first-stage configuration options can also be completely evaluated before generating optimized code.

4 NETBLOCKS IMPLEMENTATION

In Section 3, we explained the two key compiler contributions of the paper - a high-performance aspect-oriented programming pattern in C++ and a novel Layout Customization layer. In this Section, we explain the application of these techniques to a modular and extensible network DSL - NetBlocks. We will also discuss the programming API of the generated NetBlocks stack.

4.1 NetBlocks Architecture

NetBlocks takes a compiler approach to implement a high-performance network stack. Figure 18 shows the overall architecture of the system. The compiler which is built on top of BuildIt is divided into two major components - the Framework and the Modules which together apply the aspect-oriented programming pattern to generate code. The user-specified NetBlocks DSL input or schedule is fed into the compiler which configures different modules including picking required features and their variations and constraints of deployment like number of hosts, maximum payload length, among others. The compiler then generates low-level C customized to the DSL input. This generated code implements both the logic of all the selected features and the specialized packet layout required by the features. The generated code is compiled and linked with the network application along with a NetBlocks runtime library that provides some common utility functions that do not depend on the specialization. The runtime library also implements a POSIX compatibility layer that allows unmodified network applications written targeting the POSIX API to be linked against NetBlocks generated code. Finally, we describe the implementation of a WireShark plugin generator in the compiler for improved debuggability and boosting developer productivity.

4.1.1 Framework. The main component of the NetBlocks DSL compiler is the Framework. The Framework implements the logic to register, and invoke modules that implement individual features. The Framework also implements the network packet customization logic using the Layout Customization Layer and calls BuildIt for code generation. In the following, we describe the primary functions of the Framework.

Registering and Scheduling Modules: The Framework acts as a baseboard where all the modules are plugged in. The Framework allows modules to be registered by calling the `register_module` function. This function also accepts the dependencies for each module and schedules their execution. For example, the Inorder Module that implements various flavors of inorder delivery can register itself to depend on the Identifier Module. The Identifier Module identifies which connection a particular packet belongs to and this information is required by the Inorder Module to compare the sequence numbers. Modules specify different dependencies for each control path. The Framework performs a topological sort on the modules on each path to ensure all dependencies are satisfied before a module is invoked.

Implementing Control Paths: The Framework implements logical paths that correspond to the various control flow paths executed in the generated code at runtime. The registered modules can then insert their hooks into these paths to augment their behavior in accordance with the aspect-oriented programming pattern. The NetBlocks Framework implements the following paths.

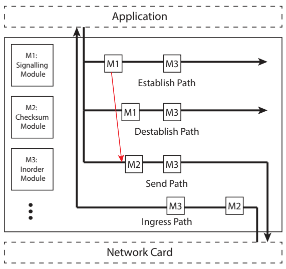
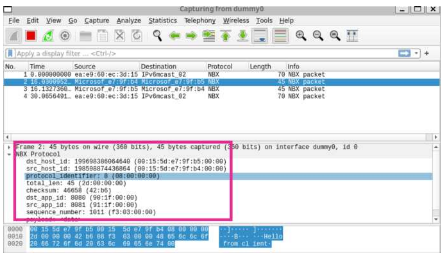


Fig. 16 The Framework with different paths and registered Modules in the NetBlocks compiler. Modules can invoke other paths if required.


Capturing from dummy0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time          Source              Destination        Protocol  Length  Info
   1 0.000000000   ea:e9:60:ec:3d:15   IPv6mcast_02       NBX            70 NBX packet
   2 16.0300952…   Microsof_e7:9f:b4   Microsof_e7:9f:b5  NBX            45 NBX packet
   3 16.1327360…   Microsof_e7:9f:b5   Microsof_e7:9f:b4  NBX            45 NBX packet
   4 30.0656491…   ea:e9:60:ec:3d:15   IPv6mcast_02       NBX            70 NBX packet

Frame 2: 45 bytes on wire (360 bits), 45 bytes captured (360 bits) on interface dummy0, id 0
NBX Protocol
    dst_host_id: 199698386064640 (00:15:5d:e7:9f:b5:00:00)
    src_host_id: 198598874436864 (00:15:5d:e7:9f:b4:00:00)
    protocol_identifier: 8 (08:00:00:00)
    total_len: 45 (2d:00:00:00)
    checksum: 46658 (42:b6)
    dst_app_id: 8080 (90:1f:00:00)
    src_app_id: 8081 (91:1f:00:00)
    sequence_number: 1011 (f3:03:00:00)

0000  00 15 5d e7 9f b5 00 15  5d e7 9f b4 08 00 00 00   ··]····· ]·······
0010  2d 00 00 00 42 b6 08 f3  03 00 00 48 65 6c 6c 6f   -···B··· ···Hello
0020  20 66 72 6f 6d 20 63 6c  69 65 6e 74 00             from cl ient·
```

Fig. 17 Screenshot of WireShark dissecting packets from a custom generated NetBlocks protocol. Bit fields are expanded to full values.

- **Init Path**: This path implements the one-time initialization logic for the whole generated stack. This allocates global data structures owned by the modules like the timer heap, or the table to hold active connections.
- **Establish Path**: This path implements the steps involved in creating a new connection when the application calls `nb_establish`. This includes allocating and initializing data structures and sending signaling packets to inform the remote host if the feature is enabled.
- **Destablish Path**: This path does the opposite of the Establish Path and implements logic to tear down a connection and free up resources when the application calls `nb_destablish`.
- **Send Path**: The Send Path is one of the key paths that includes logic to send a packet to the remote host. This includes allocating the packet, setting the headers, filling the payload, and handing over the packet to the NIC. This is invoked when the application calls `nb_send`. The Send Path can also be invoked by the modules from other paths for example to send acknowledgements when a packet is received on the Ingress Path.
- **Ingress Path**: The Ingress Path implements all the logic to process an incoming packet. Unlike other paths that are invoked by the application, this path is invoked by the NIC when it receives a new packet.

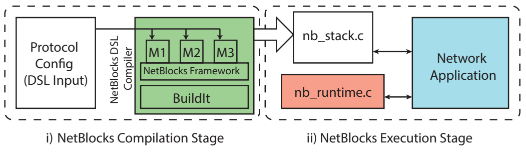


Fig. 18 The overall architecture of the NetBlocks DSL compiler with compilation phase where the NetBlocks DSL input is used to generate a specialized stack that is linked with the application in the execution phase.

The Framework generates code for each path by iterating through all the hooks on that path and invoking them. The hooks return a status code `HOOK_CONTINUE` to continue the execution or `HOOK_DROP` to drop the packet and terminate the execution of the path. Iterating through and invoking the hooks and checking the status code is done in the first stage with BuildIt's `static<T>` types while the actual logic to process the packets is implemented using BuildIt's `dyn<T>` type. This results in generating simplified code that doesn't have the control overhead. Figure 16 shows a block diagram of the Framework with the modules inserting hooks into the various paths.

Creating Packet Layout: The Framework also performs the critical function of creating and maintaining the packet layout. We discussed in Section 3 how the Layout Customization Layer allows creating and configuring binary fields. The Framework maintains a single `net_packet` object where the fields are inserted by the modules. The Framework allows inserting fields in different groups for independent scheduling. For each group, the user either specifies a manual layout or asks the Framework to pick the best packing ordering based on the size and alignment specified for the field by the modules. This idea of different policies for different groups allows for maintaining compatibility with existing protocols. For example, if the generated code is running on a network that has legacy switches that understand only the Ethernet protocol, the user can choose to manually fix the layout for the Ethernet group and let the framework optimize the rest. As explained in Section 3, besides conditionally inserting fields, the modules can also fine-tune the exact ranges of the values for each field allowing the framework to shrink the number of bits required to store the header.

Debugging and Generating WireShark Plugins: Debugging NetBlocks generated protocols and packets can be tricky since NetBlocks uses a minimal number of bits to pack the headers which might change positions even with a slight change in the DSL input. To facilitate debugging, NetBlocks automatically generates a WireShark [15] protocol dissector plugin that displays all packets with their headers values shown in a readable form. The framework simply iterates through all the headers added in the `net_packet` and calls the `get_integer()` function. The value returned is converted to a string and passed to the WireShark plugin API functions to be displayed as a field. This requires no extra effort from the user or the module developer. Figure 17 shows the screenshot of WireShark with the NetBlocks generated packets and their fields. Notice that even though the fields are shrunk to a few bits, the complete value is displayed by the plugin. Since NetBlocks modules are built on top of the BuildIt framework, it automatically benefits from the accompanying D2X [7] debugger. D2X allows attaching a standard debugger like `gdb` to the generated code and viewing the first stage code and state which in our case is the implementation of the module. This completely alleviates the need to look at the generated code and further streamlines debugging.

4.1.2 Modules. The Modules are the part of the NetBlocks compiler that implements the actual features. The modules are designed in a way that each module implements a different feature that can be composed with other modules like - reliability, checksumming, and in-order delivery among others. Besides generating implementation for the logic, modules also create and manage headers that are required to implement the feature. For example, the checksumming module controls the checksum field, while the in-order delivery module controls the sequence number field. Each module also has a series of configuration parameters that allow the protocol designer to choose different flavors of the feature in the NetBlocks DSL input. For example, for checksumming the developer can choose whether they want to checksum the entire packet or just the headers. For In-order delivery, the developer can choose whether the out-of-order packets should be simply dropped or held in a reorder buffer. We describe the implemented modules below.

Modules are similar in use to compiler passes in a traditional compiler like LLVM with the key difference that instead of containing logic to analyze, transform, and generate code, the modules are simply written as a library for the feature they implement allowing NetBlocks to be incredibly easy to extend for network developers. The implementation of the modules uses classic C++ abstraction techniques like inheritance and virtual dispatch. All the modules derive from an abstract class called `Module`. The `Module` class declares virtual methods for the hooks on each of the above-described paths which the modules can choose to override. Figure 19 shows the definition of the `Module` class and all the hooks. Notice that the hooks accept arguments of type `dyn<T>` since these values like the buffer and the length of the packet will only be known at runtime. The modules are the primary means of extensibility in the NetBlocks compiler. Networks researchers and developers can easily implement new features like compression, and encryption by adding a new module which as we have explained looks exactly like a library.

We now describe each of the modules currently in NetBlocks, the feature they implement, and the configuration options they have -

Payload Module: The payload module is the simplest module in NetBlocks that copies the payload in and out of the packet and sets the length field. This module does not offer any configuration

```
struct Module {
  virtual void hook_init(void) {return HOOK_CONTINUE;}
  virtual void hook_establish(dyn<conn_t*> c, dyn<host_t> dst, dyn<app_t> dst_app, dyn<app_t> src_app) {return HOOK_CONTINUE;}
  virtual void hook_destablish(dyn<conn_t*> c) {return HOOK_CONTINUE;}
  virtual void hook_send(dyn<conn_t*> c, dyn<char*> buff, dyn<size_t> len, dyn<size_t*> ret_len) {return HOOK_CONTINUE;}
  virtual void hook_ingress(dyn<packet_t>) {return HOOK_CONTINUE;}
};
```

Fig. 19 Definition of the `Module` abstract class and the default definitions of all hook functions

options besides controlling the range of the length field. If the user specifies a fixed-sized packet, the length field is dropped.

Network Module: The network module is the other basic module in NetBlocks that does not implement any feature but performs the job of interfacing with the NIC. It is scheduled at the end of the send path and the beginning of the ingress path. It calls functions from the runtime library to enqueue outgoing packets into the NIC and pick up received packets. This module does not offer any configuration parameters.

Identifier Module: The identifier module is one of the key modules in NetBlocks and implements identifying which connection a particular packet belongs to. It manages fields like source and destination host and application identifiers and sets these fields on outgoing packets. These roughly correspond to the IP addresses and port numbers in standard UDP and TCP. The Identifier Module also maintains a table of all active connections and their identifiers. This module runs at the beginning of the receive path so the connection is identified before other modules like in-order and reliability can use this information. This module allows configuring whether a 2-tuple or a 4-tuple should be used to identify a connection. 2-tuples are useful when the hosts run a single application and all packets should go to the same application. This module also allows setting the ranges of the host identifiers and the app identifiers to shrink the number of bits required.

Checksumming Module: The checksumming module computes and checks the checksum of all packets sent on the network. This module inserts logic in the send path to stamp a checksum and checks it in the ingress path. This module is scheduled pretty early on the ingress path to drop packets where the checksum doesn't match. This module offers a configuration option to specify whether the whole packet should be checksummed or just specific header fields. Currently, this module implements a basic checksumming algorithm similar to the one in IP but more algorithms can be easily added just like a library. Checksumming can also be completely disabled if not required.

Inorder Module: The inorder module ensures that incoming packets arrive in the same order they were sent. This module inserts and manages the sequence number field which is set and incremented on each packet sent. On receiving a packet, the sequence number is compared against the last received sequence number. This module offers a configuration option to decide whether to drop the out-of-order packets or hold them indefinitely in a reorder buffer to be delivered later. In-order delivery can also be completely disabled if not required.

Reliable Module: The reliable module ensures that each packet that is sent is received by the remote host. This module shares the sequence number field with the in-order module but also adds the acknowledgment sequence number field. The reliable module hooks the ingress path and sends an acknowledgment for each packet received. It also hooks the send path to keep unacknowledged packets in a redelivery buffer. This module inserts a timer with a configurable timeout to resend a packet in case an acknowledgment is not received in time. Currently, our implementation doesn't support features like dup-ack but can be easily added by a network developer. This module supports a configuration option to piggy-back acknowledgments on outgoing packets. Reliability can also be completely disabled if required.

Signalling Module: The signaling module performs the job of informing the remote host when a connection is established. This module invokes the send path with an empty packet in its implementation of `hook_establish` and fires the ESTABLISHED callback only when a packet is received from the other side. This ensures that both sides are aware of the connection before any real data is exchanged. We have described how this module combined with other modules realizes the three-way handshake commonly seen in TCP in the supplementary material. Signaling can be disabled if not required.

Routing Module: This routing module implements routing of packets over multiple hops based on global identifiers. This performs functions similar to the IP protocol. Routing can be configured to be enabled or disabled.

Compatibility Module: The compatibility module simply inserts and sets fields required for compatibility with existing protocols like the `EtherType` field in the ethernet protocol or the set of flags and identification fields in IP. This module can be configured to be compatible with existing protocols like ethernet, IP, or UDP or no compatibility. Full byte-level TCP compatibility is currently not supported but most essential features from TCP are implemented.

4.1.3 Runtime. The NetBlocks DSL compiler also ships with a runtime library that is linked against the application. This hand-written runtime library implements utility functions called by the generated code like a timer heap, a data queue to hold data delivered to the application, and some routing table functionality among others. The runtime library also provides support for interfacing with the Network Interface Cards (NIC) which we call the Transport Runtime. This code is fixed and does not change with the generated code. We have implemented 4 different transport runtimes - *i*) an IPC transport for testing applications running on the same host as different processes and communicating via IPC channels *ii*) a Linux transport runtime that uses POSIX raw sockets to send and receive packets over the NIC (works with any NIC that is supported by Linux) *iii*) A low-latency kernel bypass MLX5 transport that works with Mellanox ConnectX-5 NICs providing single digit latency *iv*) A NS3+DESERT transport runtime to interface NetBlocks generated code with the DESERT [37] underwater robotics simulator.

Table 5 shows the lines of code required to implement the components of the NetBlocks DSL compiler. We see that each of the modules implementing a separate feature is only a few hundred lines of code. This is because even though NetBlocks is a compiler, the modules need to implement just the core logic of the feature and not write compiler transformations, analyses, or code generation. This also shows that the amount of effort required to extend NetBlocks to add more features is very low making NetBlocks accessible to non-compiler experts. The lines of code for the Runtime include all 4 transport runtimes and contain logic to interface with the NIC.

4.2 Network API

In this Section, we briefly explain the programming APIs of the generated network DSL code that the applications can use. The most critical part of the design is that this API is the same for all the generated stacks. This means the application developers can quickly swap out the stacks with different features without having to rewrite or modify the applications.

NetBlocks stack basic API: The NetBlocks generated code implements an API that is different from the standard POSIX API. Instead of sockets with blocking operations and `select/poll/epoll`, we implement a callback-based API where the application can register a callback function when creating a connection. This callback is invoked when any event occurs on the connection like when new data is ready to be read. Such an inverted control flow allows the network stack to better schedule the operations and is more suitable for a low-latency interrupt-free environment. Table 3 shows the API functions that the application can call and their descriptions.

Table 3 API functions implemented by the NetBlocks generated code to be called by the application.

Function Name	Description
`void nb_net_init(void)`	Initialize the NetBlocks generated stack
`nb_connection_t* nb_establish(unsigned long dst_host, unsigned int dst_app, unsigned int src_app, callback_t c)`	Establish a new connection with the specified 4-pair. The `dst_host` and the `dst_app` can be specified as wildcards to accept connections. The registered callback function is registered with the newly created connection object
`void nb_destablish(nb_connection_t* c)`	Destablish a connection and free the resources associated with the object.
`int nb_send(nb_connection_t* conn, char* buffer, int len)`	Send the buffer of the length `len` on the specified connection and return the number of bytes sent. `-1` returned on failure
`int nb_read(nb_connection_t* conn, char* buffer, int len)`	Read upto `len` amount of data in the buffer from the specified connection and return the actual amount of bytes read. This call is non-blocking and returns `0` if no data is available to be read. Returns `-1` on error
`void nb_main_loop_step(void)`	Drive the protocol implementation main loop and process packets

NetBlocks POSIX compatibility layer The NetBlocks runtime library also implements a POSIX compatibility layer for running the NetBlocks generated code with existing applications that use the POSIX API. The POSIX compatibility layer wraps around the above NetBlocks API by creating virtual file descriptors and implementing the functions - `socket`, `connect`, `accept`, `bind`, `listen`, `recv`, `send`, `setsockopt`, `ioctl`, `close`, `select`, `write`, `read`, `writev`. The functions that block internally call `nb_main_loop_step` till the required event is met. Thanks to the POSIX compatibility layer, we are able to run applications like NGINX that were originally written for TCP and run with lightweight protocols like UDP without a single line of code modification.

5 EVALUATIONS

In this Section, we evaluate the performance of the various protocols generated and demonstrate the tradeoff the NetBlocks compiler offers in terms of performance and features. We evaluate the latency of communication for a simple Echo application and a real-world unmodified NGINX web server. We also demonstrate the performance tradeoff when the network protocol is deployed underwater. Finally, we compare the packet header overheads for the various protocols and the lines of code they generate to get a sense of the resources they require at deployment.

5.1 Evaluation Methodology

Testbed. For latency-critical applications typically found inside data centers, we run our experiments on two servers with 4-core Intel Xeon Gold 5122 CPUs running at 3.6 GHz with 64 GB of main memory and 16.5MB L3 cache. Both servers are running Ubuntu 22.04. Each node is equipped and connected to each other with a 100 Gbps Mellanox MT27800 family ConnectX-5 NIC that offers microsecond round trip times.

For the underwater robotics evaluation, we run our generated protocols with the DESERT [37] underwater simulator that is built on top of the NS2 network simulator [21]. DESERT simulates the low-bandwidth, high-latency, and other network conditions in the underwater acoustic medium. We used a 4800-bit/second channel, along with the default MAC protocol to prevent collisions.

Comparison Configurations The primary aim of our evaluation is to demonstrate the feature-vs-performance tradeoff offered by the NetBlocks generated code. We compare the performance of various protocols generated with NetBlocks with increasing degrees of features. We list the set of protocol configurations below -

- **UDP-like**: Closely resembles the UDP protocol running on top of IP+Ethernet and does not have any features like reliability, in-order delivery, or signaling.
- **UDP-over-Ethernet**: Has the same features as UDP-like, but removes the IP layer and thus does not support routing. Routing is not needed for many deployments like in an all-to-all topology or a deployment like underwater robots.
- **Inorder**: Builds on top of UDP-over-ethernet and adds a basic inorder delivery.

Table 4 Header sizes in bytes and the code size in lines of C code. All generated protocols are linked against a runtime library of 493 LoC C code.

Protocol Configuration	Header size	Generated code size
UDP-Like	42	252
UDP-over-Ethernet	20	241
Inorder	24	296
Reliable	28	364
Signalling	26	331
FullChecksumming	28	358
ShrunkFields	4	253
Linux (UDP)	42	–
Linux (TCP)	> 56	–

Table 5 Implementation complexity of different components of NetBlocks. Each module is only a few hundred lines of C code.

NetBlocks Component	Lines of Code
Identifier Module	343
Inorder Module	177
Reliable Module	164
Routing Module	124
Signalling Module	131
Checksumming Module	123
Payload Module	85
Network Module	61
Framework	1,113
Runtime	1,826

- **Reliable**: Builds on top of Inorder and adds reliable packet delivery with acks.
- **Signalling**: Builds on top of Inorder and adds signaling packets at connection establishment.
- **FullChecksumming**: Builds on top of Inorder and adds full packet checksumming.
- **ShrunkFields**: Similar to UDP-over-Ethernet but uses restricted fields.
- **Linux (UDP/TCP)**: We also compare our implementations against the default Linux implementation to show that our implementation is competitive with existing implementations. We use UDP for the echo application and TCP for the NGINX application.

5.2 Protocol Header and Code Size Overhead

Before we evaluate the performance of protocols, we first present the header sizes for the implementations and the size of the generated code. Minimizing protocol header sizes is critical for bandwidth-constrained applications while reducing code size and memory footprint is important for memory-constrained deployments like IoT. Table 4 shows all the configurations and the header sizes along with their generated code size. We notice that the UDP-over-Ethernet protocol eliminates redundant headers and reduces the header size by over 50%. Other protocols that build over UDP-over-Ethernet, add small overhead for the specific feature-related fields they add. For example, Inorder delivery adds a 32-bit field to store the sequence numbers, while Reliable delivery requires another 32-bit field for storing the acknowledgment sequence numbers. ShrunkFields shows the smallest header possible with 4 bits each for source and destination host identifiers (MAC addresses) 4 bits for source and destination app identifiers (port numbers) and 16 bits for the length field. For each of the configurations, similar to the header size, the lines of code required to implement are marginally more than the base case. The lines of code shown here are just the generated lines of C code and do not include the linked runtime library.

5.3 Applications

In this Section, we compare the latency of the protocol configurations for a simple Echo application, an NGINX web server [12], and an underwater robotics simulation.

Echo Application We implement a simple echo application that ping-pongs messages back and forth between server and client and measures round-trip latency for each message. Since the server and the client do not perform work other than networking, this application allows us to isolate the overheads of each feature. To test the effects of the features like signaling, each message is sent over a newly established connection. The performance of protocols that don't use signaling is unaffected other than the small local setup cost since they don't send any messages. We evaluate this application on two hosts connected with a 100Gbps connection. We link NetBlocks generated code against a kernel bypass runtime to avoid syscall overheads. The default Linux (UDP) protocol is evaluated using the kernel implementation and suffers from the syscall overhead. For this evaluation,

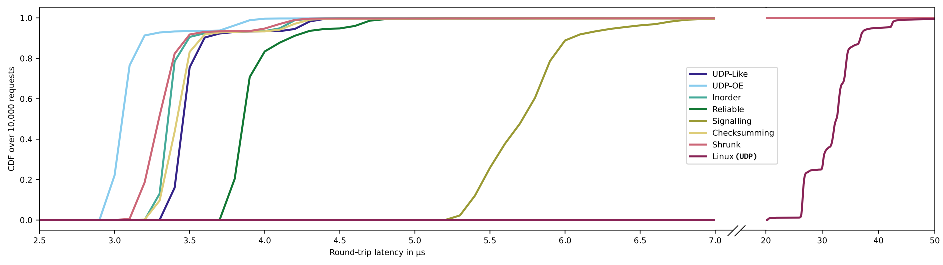


Fig. 20 CDF of the round-trip latency over 10,000 messages for the Echo application with the 8 protocol configurations. The message size used for the ping-pong messages is 256 bytes.

we send messages of 256 bytes back and forth to thoroughly test the overhead from the schemes that involve checksumming.

Figure 20 shows the Cumulative Distribution Function (CDF) plot of the latency over 10,000 packets. We notice that the minimum latency is obtained with the UDP-over-Ethernet protocol configuration with a median latency of 3.25µs since it does not have any features like routing, checksumming, signaling, in order or reliable delivery. This is close to the single-digit microsecond latency possible with RDMA for packets of this size [27]. As we enable these features the latency increases with signaling having the highest median latency of 6.0µs. This latency is almost twice the latency because for every connection signaling packets have to be exchanged before the actual messages are sent. For the protocol that implements reliability, an acknowledgment needs to be sent for every packet, but the sender doesn't have to wait for the acknowledgment before sending the next message and hence this scheme increases the latency slightly. Finally, we see despite having the smallest header size, the ShrunkFields configuration has more latency overheads than the UDP-over-ethernet protocol. Despite having the same features, this configuration needs to generate bit-packing code with masks and shifts adding latency in the host processing. With this observation, we conclude that the different schemes provide a real tradeoff in different metrics. The Linux configuration is an order of magnitude slower than our slowest scheme because it doesn't use a kernel bypass stack and suffers from overheads of syscalls and interrupts. .

This evaluation demonstrates that, unlike the all-or-nothing approach of UDP and TCP, our approach of generating protocols with select features provides a spectrum of performance. This allows the users to have a pay-for-what-you-use policy when it comes to features. The DSL inputs for each of these protocols are less than 50 lines of C++ code, allowing the user to switch between vastly different protocols with minimal effort.

NGINX For the next evaluation we use a real-world web server NGINX [12] to demonstrate that NetBlocks generated protocols are all supported by applications written for the POSIX API. We run an unmodified NGINX web server written with TCP sockets with the 7 protocol configurations generated from NetBlocks. We send GET requests that download static files off the server. The files are stored in an in-memory file system to avoid variance from disk reads. The NGINX web-server runs on top of the NetBlocks POSIX compatibility layer explained in Section 4. We measure the round-trip latency over 10,000 GET requests downloading a file of 850 bytes. The total payload also includes the HTTP response headers along with the file.

Figure 21 shows the CDF plot for the 10,000 requests. We observe a similar trend as the Echo application, even though the absolute latency numbers are higher. This is due to the fact that the server has to perform system calls to read the files. However, we demonstrate that the benefits of specializing the protocol are translated to real-world applications. In this evaluation, the overhead of Checksumming is significantly higher than the Echo Application because the payload size is much larger. Similarly, the overhead of Reliability is higher since the NGINX server sends the

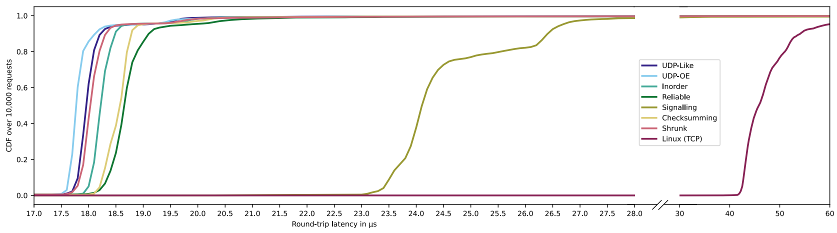


Fig. 21 CDF of the round-trip latency over 10,000 requests for the Nginx application with the 7 configurations.

response over multiple messages each requiring its own acknowledgment. Thanks to NetBlocks's design of exposing the same API to the application regardless of the generated protocol, we are also able to run NGINX with a state-less UDP-like protocol without any source modifications despite being originally written for TCP. To our knowledge, this is the first time NGINX has been run with a header as small as 4 bytes. For the comparison Linux implementation, we use TCP because the two protocols are not compatible in Linux.

Underwater Robotics For the next evaluation, we run our generated stacks with the DESERT [37] underwater simulator to evaluate the effect of feature selection on throughput in a low-bandwidth environment.

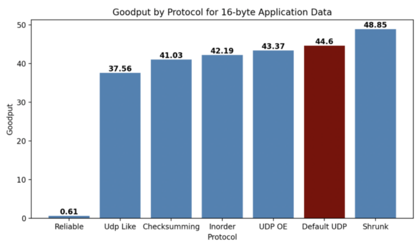


Fig. 22 Goodput measurement for the underwater robotics sensor simulations with DESERT. The payload size is 16 bytes.

Since the latency of the communication is bottlenecked by the acoustic medium, the host side overhead is negligible and our techniques don't affect latency. However, being able to compactly pack the headers helps us better utilize the low-bandwidth channel. For our evaluation, we set up a simulation between two hosts where one host repeatedly sends a 16-byte payload to the other host over a 4800 bps channel. The 16-byte payload mimics sensor measurement readings on the robot. Figure 22 shows the measured Goodput [33] for our custom protocols and the default UDP implementation in DESERT. The goodput (i.e., the number of useful application bytes transmitted per second without considering retransmissions) for the Shrunk scheme is the highest. By minimizing the header sizes, we better utilize the channel for the actual payload. The default UDP implementation in DESERT has been optimized for the underwater scenario and is the second best. As we add more features, the goodput gradually decreases and is extremely low when reliability is enabled. This is because, with reliability, the receiving host also has to send packets for the acknowledgments which adds contention to the channel and the underlying MAC protocol cannot utilize the channel optimally. This further elaborates the point that we need custom protocol generators like NetBlocks. With just TCP and UDP, if the system requires inorder delivery, the operator is forced to use TCP and sacrifice the performance with the acknowledgments and handshakes.

Discussion These three applications and scenarios show that when minimizing latency or maximizing throughput is of concern, creating a custom protocol with NetBlocks can have a real impact. It shows that each network feature comes with a price, and creating a custom protocol with only the salient features your application needs is much better than the two-sizes-fits-all approach with TCP and UDP. We also demonstrate that NetBlocks compiler generates highly optimized kernel by-pass code; thus, the latency results are much better than the hand-tuned Linux implementations.

6 RELATED WORKS

Optimizing networks stack to obtain the best out of the hardware has a long history of research both in implementing high-performance hand-tuned libraries for TCP/IP/UP [1, 14, 19, 20, 26, 27, 38, 43] and manually creating custom protocols to better fit the applications and environment [24, 34, 36, 42, 43]. Researchers have proposed algorithms for improving existing features like congestion control [2, 13, 18, 31, 40]. However, the handwritten implementations that add or improve a few features are not able to keep up with the rapidly changing end-to-end application needs that require custom tweaking of the full protocol.

DSLs provide a succinct input representation to the end-user while generating low-level high-performance code with optimizations and features tailored for the domain [10, 11, 22, 29, 41, 48, 50].

DSL compiler techniques have also been applied within the broader network domain for creating custom protocols [3, 4, 16, 23, 30, 35, 49] as well as for optimizing network applications [17]. While these network DSLs allow customizing some features or layers of the protocol, they are not able to perform whole stack optimization. In particular, P4 [4], the most popular application of compiler techniques to the networking domain, only focuses on packet forwarding and stateless packet processing for programmable switches, while NetBlocks is able to customize features like reliability on the host network stack. Consequently, NetBlocks hosts maintain complex network states including re-delivery buffers and timers. Similarly, Rubik [35] proposes a DSL for programming network stacks for middleboxes. However, Rubik only focuses on optimizing bi-directional traffic flows in middleboxes while also requiring intricate knowledge of compilers. In contrast, NetBlocks customizes the host network stack. ClickNF [16] extends the ideas in Click [30] to host network stacks. Similar to Rubik, ClickNF takes a traditional compiler approach. Consequently, making adoption of new features and libraries challenging, unlike NetBlocks where new C/C++ libraries can simply be ported as new modules by non-compiler experts.

Multi-stage programming is a promising approach to minimizing the complexity of the compiler implementation and reusing the optimizations written as a library [6, 44, 46]. BuildIt [5] is a multi-stage programming framework that is suitable for the network domain because it is written as a library in C++ and generates high-performance C or C++ code. Since a lot of existing network libraries and application-specific implementation is already written in C++, the effort required to move to a compiler is minimal. The BuildIt framework also has an accompanying multi-stage debugger called D2X [7] that makes debugging the generated code and by extension the DSL easier. BuildIt's staging capabilities applied to aspect-oriented programming pattern helps create low-cost modular abstractions. Aspect-oriented programming and its application have been studied a lot in the past [32]. DSL approaches have also been used to solve this problem [28, 47] but they often require writing the layout specification in a separate language than the network specification. NetBlocks combines both problems into a single DSL input that is easy to customize and extend. To our knowledge, NetBlocks is the first compiler for the network domain that can be extended by network experts like a library.

7 CONCLUSION

We present NetBlocks, a modular and extensible network DSL compiler for generating custom protocols using BuildIt. To customize binary layouts in packets, we introduce the idea of staging layouts alongside code. NetBlocks paves the way for design and deployment of newer generation ad-hoc protocols that better suit the needs of the applications and environment. Our techniques also open up the conversation for applying staging and related compiler techniques to other systems domains in a way that *i*) it is accessible to practitioners who have no compilers knowledge and *ii*) it can reuse the existing knowledge and code base for writing high-performance DSL compilers.

ACKNOWLEDGMENTS

We would like to thank Dhruv Saraff, Imperial College London for his contribution to the Layout Optimization Implementation. This work was supported by the ACE Center for Evolvable Computing Research Center, a JUMP 2.0 Center Center co-sponsored by SRC, DARPA and other companies; Intel and NSF PPoSS Grant CCF-2217064; DARPA PROWESS Award HR0011-23-C-0101; NSF SHF Grant CCF-2107244.

ARTIFACTS AVAILABILITY STATEMENT

NetBlocks and all its components are available open-source [8] under the MIT License. All the source code for NetBlocks can be found under the `net-blocks` directory. The `README` in the repository contains instructions to build all dependencies and applications and evaluate their performance. The README also has instructions on adding a new toy module to extend the system.

REFERENCES

[1] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 93–109. https://www.usenix.org/conference/nsdi20/presentation/arashloo

[2] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: Load and State-Aware Receive Side Scaling. In *Proc. CoNEXT*. https://doi.org/10.1145/3359989.3365412

[3] Saleem Bhatti, Edwin Brady, Kevin Hammond, and James McKinna. 2009. Domain Specific Languages (DSLs) for Network Protocols (Position Paper). In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*. 208–213. https://doi.org/10.1109/ICDCSW.2009.64

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *Proc. SIGCOMM* (2014). https://doi.org/10.1145/2656877.2656890

[5] Ajay Brahmakshatriya and Saman Amarasinghe. 2021. BuildIt: A type based multistage programming framework for code generation in C++. In *Proc. CGO*. https://doi.org/10.1109/CGO51591.2021.9370333

[6] Ajay Brahmakshatriya and Saman Amarasinghe. 2022. GraphIt to CUDA Compiler in 2021 LOC: A Case for High-Performance DSL Implementation via Staging with BuilDSL. In *Proc. CGO*. https://doi.org/10.1109/CGO53902.2022.9741280

[7] Ajay Brahmakshatriya and Saman Amarasinghe. 2023. D2X: An eXtensible conteXtual Debugger for Modern DSLs. In *Proc. CGO*. https://doi.org/10.1145/3579990.3580014

[8] Ajay Brahmakshatriya, Chris Rinard, Manya Ghobadi, and Saman Amarasinghe. 2024. *Artifacts for the PLDI 2024 paper: NetBlocks: Staging Layouts for High-Performance Custom Host Network Stacks*. https://github.com/BuildIt-lang/net-blocks-pldi24-artifacts and https://zenodo.org/records/11099781.

[9] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proc. SIGCOMM*. https://doi.org/10.1145/3452296.3472888

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proc. OSDI*. https://doi.org/10.5555/3291168.3291211

[11] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization. In *Proc. PLDI*. https://doi.org/10.1145/2345156.2254079

[12] Nginx community. [n. d.]. Nginx - a high-performance HTTP server. https://www.nginx.com/

[13] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. 2011. Proportional Rate Reduction for TCP. In *Proc. SIGCOMM*. https://doi.org/10.1145/2068816.2068832

[14] Adam Dunkels. 2023. uIP. https://github.com/adamdunkels/uip

[15] The Wireshark Foundation. 2023. WireShark. https://www.wireshark.org/

[16] Massimo Gallo and Rafael Laufer. 2018. ClickNF: a Modular Stack for Custom Network Functions. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 745–757. https://www.usenix.org/conference/atc18/presentation/gallo

[17] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *Proc. SIGCOMM*. https://doi.org/10.1145/3387514.3405879

[18] Monia Ghobadi, Soheil Hassas Yeganeh, and Yashar Ganjali. 2012. Rethinking End-to-End Congestion Control in Software-Defined Networks. In *Proc. HotNets*. https://doi.org/10.1145/2390231.2390242

[19] Altran Group. 2012. picoTCP. http://picotcp.altran.be/

[20] FreeRTOS Group. 2023. FreeRTOS-Plus-TCP. https://github.com/FreeRTOS/FreeRTOS-Plus-TCP

[21] NS2 Group. 2023. NS2: The Network Simulator. https://www.isi.edu/nsnam/ns/

[22] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* (2019). https://doi.org/10.1145/3355089.3356506

[23] Hermann Hüni, Ralph Johnson, and Robert Engel. 1995. A Framework for Network Protocol Software. *SIGPLAN Not.* (1995). https://doi.org/10.1145/217839.217875

[24] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 127–140. https://www.usenix.org/conference/nsdi20/presentation/hwang

[25] Junsu Jang and Fadel Adib. 2019. Underwater Backscatter Networking. In *Proc. SIGCOMM*. https://doi.org/10.1145/3341302.3342091

[26] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[27] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[28] Katai-IO. 2022. Katai Struct Compiler. https://github.com/kaitai-io/kaitai_struct_compiler/

[29] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler *(Proc. OOPSLA)*. https://doi.org/10.1145/3133901

[30] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* (2000). https://doi.org/10.1145/354871.354874

[31] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 514–528. https://doi.org/10.1145/3387514.3406591

[32] Mohit kumar, Akashdeep sharma, and Sushil Garg. 2009. A study of aspect oriented testing techniques. In *2009 IEEE Symposium on Industrial Electronics & Applications*, Vol. 2. 996–1001. https://doi.org/10.1109/ISIEA.2009.5356308

[33] James F. Kurose and Keith W. Ross. 2009. *Computer Networking: A Top-Down Approach* (5th ed.). Addison-Wesley Publishing Company, USA. https://doi.org/10.5555/2584507

[34] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proc. SIGCOMM (SIGCOMM '17)*. https://doi.org/10.1145/3098822.3098842

[35] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. 2021. Programming Network Stack for Middleboxes with Rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 551–570. https://www.usenix.org/conference/nsdi21/presentation/li

[36] Ilias Marinos, Robert N. M. Watson, and Mark Handley. 2013. Network Stack Specialization for Performance. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (College Park, Maryland) *(HotNets-XII)*. Association for Computing Machinery, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2535771.2535779

[37] Riccardo Masiero, Saiful Azad, Federico Favaro, Matteo Petrani, Giovanni Toso, Federico Guerra, Paolo Casari, and Michele Zorzi. 2012. DESERT Underwater: An NS-Miracle-based framework to design, simulate, emulate and realize test-beds for underwater network protocols. In *2012 Oceans - Yeosu*. 1–10. https://doi.org/10.1109/OCEANS-Yeosu.2012.6263524

[38] Aravind Menon and Willy Zwaenepoel. 2008. Optimizing TCP Receive Performance. In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) *(ATC'08)*. USENIX Association, USA, 85–98. https://www.usenix.org/conference/2008-usenix-annual-technical-conference/optimizing-tcp-receive-performance

[39] Greg Miller and Kevin Thompson. 1998. the nature of the beast : recent traffic measurements from an Internet backbone. https://api.semanticscholar.org/CorpusID:262494192

[40] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proc. SIGCOMM*. https://doi.org/10.1145/2785956.2787510

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines *(Proc. PLDI)*. https://doi.org/10.1145/2491956.2462176

[42] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/sapio

[43] Dmytro Syzov, Dmitry Kachan, Kirill Karpov, Nikolai Mareev, and Eduard Siemens. 2019. Custom UDP-Based Transport Protocol Implementation over DPDK. *Proceedings of the International Conference on Applied Innovations in IT* 7. https://doi.org/10.25673/13476

[44] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proc. PEPM*. https://doi.org/10.1145/258993.259019

[45] K. Thompson, G.J. Miller, and R. Wilder. 1997. Wide-area Internet traffic patterns and characteristics. *IEEE Network* 11, 6 (1997), 10–23. https://doi.org/10.1109/65.642356

[46] Vlad Ureche, Tiark Rompf, Arvind Sujeeth, Hassan Chafi, and Martin Odersky. 2012. StagedSAC: A Case Study in Performance-Oriented DSL Development. In *Proc. PEPM*. https://doi.org/10.1145/2103746.2103762

[47] Kenton Varda. 2008. *Protocol Buffers: Google's Data Interchange Format*. Technical Report. Google. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[48] Eelco Visser. 2008. *WebDSL: A Case Study in Domain-Specific Language Engineering*. https://doi.org/10.1007/978-3-540-88643-3_7

[49] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proc. SOSR*. https://doi.org/10.1145/3050220.3050234

[50] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL *(Proc. OOPSLA)*. https://doi.org/10.1145/3276491

Received 2023-11-16; accepted 2024-03-31