Bertha: Tunneling through the Network API

Bertha: Tunneling through the Network API

Akshay Narayan♥, Aurojit Panda♠, Mohammad Alizadeh♥
Hari Balakrishnan♥, Arvind Krishnamurthy♣, Scott Shenker♦
♥ MIT CSAIL, ♠ NYU, ♣ University of Washington, ♦ UC Berkeley and ICSI

Abstract

Network APIs such as UNIX sockets, DPDK, Netmap, etc. assume that networks provide only end-to-end connectivity. However, networks increasingly include smart NICs and programmable switches that can implement both network and *application* functions. Several recent works have shown the benefit of offloading application functionality to the network, but using these approaches requires changing not just the applications, but also network and system configuration. In this paper we propose Bertha, a network API that provides a uniform abstraction for offloads, aiming to simplify their use.

ACM Reference Format:
Akshay Narayan, Aurojit Panda, Mohammad Alizadeh, Hari Balakrishnan, Arvind Krishnamurthy, Scott Shenker. 2020. Bertha: Tunneling through the Network API. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20), November 4–6, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422604.3425927

1 Introduction

Application developers today can choose from a wide range of network APIs, including traditional UNIX sockets, application libraries such as QUIC [35], kernel bypass libraries such as DPDK [15] and Netmap [55], accelerated kernel libraries such as XDP [26], and hardware-specific interfaces such as RDMA. However, all of these libraries build on the assumption that the network only provides *best-effort, end-to-end packet delivery*. Even higher level libraries, including RPC libraries such as gRPC [22], build on these low-level APIs and hence embody the assumption. While consistent with the original Internet *architecture* [56], it does not reflect today's network *infrastructure*, which includes on-server offloads such as SmartNICs and in-network offloads such as programmable switches that can implement *both application and network stack* logic.

Recent research and commercial products have shown that application functions such as key-value store caches [31], consensus protocols [37], and transport protocols [6, 12]—which were traditionally not considered a part of the network—can be implemented using offloads [14], and that such implementations can improve application performance and reduce resource requirements. Existing network APIs, which focus on end-to-end delivery, do not provide applications a way to invoke SmartNIC or in-network offloads. As a result, using offloads requires close coordination between the application developer, offload developer, system administrator, and network operator (Figure 1). This need for such coordination is a barrier to the adoption of these approaches. Similarly, adopting new offloads or changing how offloads are implemented requires changing the application, and a further round of coordination. Thus using offloads also ossifies applications, and makes upgrades more complex. In this paper, we describe an extensible network API that embraces the use of network offloads and simplifies applications adopting offloads.

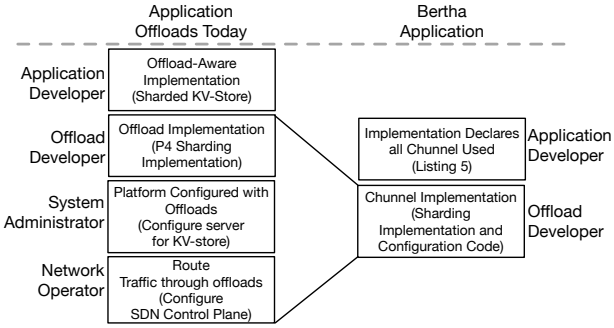

Figure 1: **Current methods of configuring network functionality must stretch across multiple abstraction boundaries.**

To develop a more general interface, we turn to the model of tunnels: network tunnels, as in VXLAN and the like, provide end-to-end delivery with some additional functions (such as encryption or labels), while being *transparent* to all but the tunnel end points and *composable*. Our proposal, called Bertha (Figure 1), uses a tunnel-like abstraction called a Chunnel to build an extensible network interface that requires no manual changes to network or system configuration when deploying applications that use on-server or in-network offloads. Each Chunnel type represents application-relevant functionality that can potentially be offloaded to the network. Application developers specify offloads as a directed acyclic graph (DAG) of Chunnels, and offload developers *register* available Chunnels with the Bertha discovery service. Bertha selects where Chunnel functionality is implemented when establishing a connection and appropriately updates system and network configurations to enable their use. We show examples of how applications can benefit from these interfaces in §3.2.

A key insight behind our proposal is that while existing work often treats on-server and in-network offloads differently, they are nearly indistinguishable from an application's perspective. Thus, it is possible to provide a unified abstraction for nearly all communication-oriented application offloads, regardless of where or how they are implemented. In this paper, we describe the Bertha interface and a prototype implementation. Not only does Bertha make it easier for applications to use new offloads, and improve portability, but choosing implementation when connections are

HotNets '20, November 4–6, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8145-1/20/11.
https://doi.org/10.1145/3422604.3425927

established also opens up new avenues for placement, e.g., allowing servers to offload processing to clients. We quantify these benefits in §5 and discuss other opportunities enabled by Bertha in §6.

2 Overview

We begin by providing an overview of Bertha and how it simplifies applications accessing offloads.

Chunnels are the core abstraction provided by Bertha. Each Chunnel type represents a single communication-oriented function, and an application using Bertha specifies a set of such functions for a connection using a DAG of Chunnel types (§3). A Chunnel type can encompass a variety of functions, but should meet the following goals:

Application-relevant. Each Chunnel must provide a capability relevant to some application, i.e., functions that application developers explicitly opt to use. It should also be possible for an application developer to opt out of the functions provided by a particular Chunnel type. This requirement precludes Chunnel types that cannot be bypassed, e.g., network security, authentication, billing, etc.

Host fallback. All Chunnel functionality should be implementable entirely by software running at the end host. Note that this fallback implementation might provide worse performance than one implemented in specialized hardware. Furthermore, fallback implementations are not the primary implementation used in practice; they merely ensure that applications can function in the absence of a better implementation.

Minimal. Chunnels should ideally be minimal, performing only one or a few communication-oriented functions. However, minimality is subjective: for example, several NICs now offer TCP offload engines [12], which are designed to offload all of TCP's many functions including reliability, in-order delivery, congestion control, etc. TCP offload engines often do not support offloading portions of this functionality. Similarly, most applications either adopt all of TCP's functions, or none of them. In this case ease-of-use and ease-of-offloading make a single TCP Chunnel more appealing than a set of finer grained Chunnels.

Composable. Chunnels should be composable, allowing an application to combine functionality from multiple Chunnel types in a single connection. The ability to compose Chunnels allows applications to combine multiple functions implemented in network offloads, and thus decouples application evolution from offload creation. In §6 we also describe how reasoning about compositions of Chunnels can enable further optimizations.

Beyond these requirements, in order to be useful Chunnels must provide functionality that is useful to more than one application, and in this paper we focus on those Chunnels which can be accelerated through the use of hardware or software offloads. Our focus on application-relevant functionality distinguishes us from prior work in NFV orchestration and frameworks such as DOA [61] and NetCalls [57], which focus on network functionality rather than application functionality. This impacts Bertha's design in two crucial ways: first, Chunnels used in a Bertha connection only impact data sent over that connection and cannot impact security or other policies that apply to the host in general; second, no calls into Bertha can ever conflict with (nor modify) network policies.

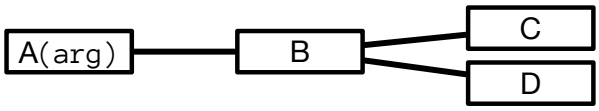

Figure 2: **The Chunnel specified in §3.1.**

Having described what a Chunnel is, we next turn to the question of where Chunnels are implemented. Our intention when designing Bertha is to allow applications to opt-in and use functionality implemented in NIC ASICs, FPGAs, SmartNIC CPUs, and programmable network switches, and on the host's network data path using libraries such as XDP, etc. Network operators, system administrators and offload developers register accelerated implementations of Chunnel types with a Bertha discovery service (§4.2). When an application establishes a connection, Bertha queries the discovery service to determine all available implementations for each Chunnel type, and decides which implementation to use based on an operator-provided scheduling policy. Applications use the software fallback implementation for a Chunnel type when no network or host provided implementation can be used, e.g., because no implementations are registered, resources required by registered implementations are already occupied, or policy constraints prevent the use of offloads.

3 Interface and Examples

Next we detail Bertha's application interface in §3.1 and provide examples of its use in §3.2. For ease of reference, we summarize Bertha-related terminology in Table 1. When adopting Bertha, applications need to use Bertha's application interface: for existing applications this means they *must be modified* to replace existing network code. However, once an application has adopted Bertha, no additional changes are required to take advantage of new offloads.

In what follows, we describe the application interface using Rust syntax, since our initial prototype is written in Rust. However, the Bertha architecture and runtime makes no assumptions about programming language used and other languages can be used.

3.1 Interface

Bertha provides a userspace library similar to UNIX sockets that two or more Bertha applications – e.g., a client and a server or a client and several replicated servers – can use to communicate.

To use Bertha, applications must create a connection endpoint, the Bertha equivalent of a socket. In our Rust prototype, this is done by calling `bertha::new`, which takes two arguments: an endpoint name which aids in debugging and a directed acyclic graph (DAG) of Chunnel types which dictates the sequence of processing steps that should be performed on any data sent or received from this endpoint. In our prototype the Chunnel type DAG is specified within a `wrap!` macro, and Chunnels are sequenced using the `|>` operator. As with dataflow graphs in other contexts (e.g., data processing), branching and merging operations are performed through the use of specific Chunnel types. The code below constructs a new connection endpoint (`foo`):

```
bertha::new("foo",wrap!(A(arg)|>
 B(B::args([C(),D()]))))
```

Term	Explanation	Example
Chunnel	Abstraction representing a piece of network-oriented application functionality.	Sharding
Offload	Specialized hardware implementing one or more Chunnels.	Tofino Switch
Fallback Impl.	Default Chunnel implementation that can run on the end host.	XDP Program
Chunnel DAG	Application's specification of its component Chunnels.	Listing 4
Scope	Constraint on where a Chunnel should be implemented.	Local scope (§3)

Table 1: **Glossary of common terms related to Chunnels.**

```
let srv = bertha::new("container-app",
  wrap!(local_or_remote()))
  .listen(SocketAddr(addr, port));
```

Listing 1: **Local-fastpath routing. If the connecting client is on the same host, the connection uses more efficient IPCs.**

```
let conn = bertha::new("ordered-multicast-client",
  wrap!(serialize() |> ordered_mcast()))
  .connect(endpts); // endpts is a list of addresses.
```

Listing 2: **The `ordered_mcast` Chunnel can automatically create a multicast group among the nodes in the consensus group.**

Endpoint `foo` is constructed with the Chunnel DAG shown in Figure 2. Some Chunnels (e.g., `A` and `B` above) require input parameters, and Bertha allows applications to provide these as arguments (e.g., `arg` as an argument to `A`) when constructing such a Chunnel. While we do not show it in this example, an application can also specify *scoping* constraints for subgraphs of the Chunnel DAG; these constrain where the Chunnel is implemented. Bertha allows offload developers and operators to provide multiple implementations of each Chunnel type and Bertha decides which implementation to use at runtime. The Bertha runtime forwards any arguments provided for a Chunnel type to the selected implementation.

Similar to existing APIs, Bertha supports two types of connection endpoint: *server* and *client*. A server can call `listen` in order to wait for a client, and a client can `connect` to a waiting server endpoint. A connection is established when a client connects to a server. During connection establishment, all endpoints exchange the Chunnel DAG they were provided and decide which implementation of each Chunnel should be used for this connection (§4). Note that Chunnel types are bound to implementations at the point where a *connection is established*, and as a result a single application might use several different implementations of the same Chunnel type.

3.2 Example Uses

Local Fast-Path Transfers. Prior work [33, 66] has shown that sending messages between containers can add significant overheads since all data between two containers must traverse the host network stack to ensure that the network API remains independent of placement decisions and to provide container isolation. Using efficient inter-process communication (IPC) mechanisms can reduce their impact.[1] In Listing 1, we show how Bertha can enable this efficient communication between containers while preserving interface uniformity; work such as Slim [66] has shown how to preserve isolation in this setting. In the listing, the `local_or_remote` Chunnel uses fast IPC calls when transferring data between containers on the same node and datagrams otherwise. We evaluate this approach in §5.

Serialization. Serialization is often an important source of overheads when building distributed applications. As a result, several recent libraries [4, 7, 21] have been developed that reduce serialization overheads on modern processors, and existing libraries such as Protobufs [23], and Apache Thrift [5] have seen increasingly aggressive optimizations, including the use of FPGA offloads [28], to improve their performance. However, using new implementations or new libraries given an application currently requires application developers to (at least) rebuild their applications, and can involve non-trivial code changes. Modeling serialization as a Bertha Chunnel can ameliorate Bertha-application's challenge of adopting new serialization implementations, including ones that are hardware accelerated. This is because (as noted above) given a Chunnel type, the Bertha runtime picks the best available implementation and falls back on the application's supplied implementation when no alternative is available. The use of a serialization Chunnel changes the connection's interface: applications send and receive objects rather than bytes.

Network-Assisted Consensus. There is a rich body of work on accelerating consensus protocols, including the use of network offloads for packet ordering [37, 52]. Listing 2 shows a potential component of a Speculative Paxos (or NOPaxos) implementation specifying the use of a network-ordering Chunnel (`ordered_mcast`). Note, that since one end of this connection involves multiple endpoints, the argument passed into connect is a vector containing endpoints addresses (including ports), and initial discovery and negotiation involves all endpoints.

Anycast. IP Anycast has traditionally been used (most popularly, with the DNS root name servers) to geo-shard requests by routing them to the closest host advertising that IP. However, due to routing instability, many developers instead opt to use DNS for this purpose [38, 47, 48]. Implementing anycast using a Bertha tunnel allows applications to dynamically choose between DNS-based and IP-anycast based approaches depending on where they are deployed.

Load Balancing, Sharding, and Routing. Finally, in Listing 3 we show an example of more complex Chunnel composition. We consider a storage service [2, 3, 46] where a service exposes a single external address, but requests sent to this address are routed to one of several backend shards for processing. In current implementations the task of steering the packet to the current backend is either performed by an application load balancer—e.g., Amazon's Application Load Balancer (ALB) [29], the F5 Load balancer [17], ProxySQL [54], McRouter [46]—or by logic hardcoded at the client. Both approaches have drawbacks: the load balancer

[1] Windows [41] implements a fast-path for localhost TCP connections which avoids reliability and congestion control (There is an unmerged proposal for a similar feature on Linux [11]) However, the use of network namespacing precludes socket libraries from using these optimizations because each network namespace has a separate loopback interface.

```
let rsm1 = bertha::new("rsm1",
  ordered_mcast()).connect(nodes1);
let rsm2 = bertha::new("rsm2",
  ordered_mcast()).connect(nodes2);
let srv = bertha::new("service", wrap!(
  shard(shard::args([rsm1, rsm2], shard_fn))))
  .listen(SocketAddr(extern_addr, port));
```

Listing 3: **Request routing.**

```
let shards = vec![s1, s2, s3];
let shard_fn = |p: Pkt| {
  p.dst_port = hash(p.payload[10..14]) % 3 }
let srv = bertha::new("my-kv-srv", wrap!(
  shard(shard::args(choices: shards),
  fn: shard_fn) |> reliable())).listen(addr, port);
```

Listing 4: **Sharding server. The use of datagram-based transport allows offloads to avoid terminating connections.**

```
// Run during client initialization, registers a Chunnel.
bertha::register_chunnel("reliable", ReliableChunnel,
  bertha::endpoints::Both, bertha::scope::Application);
fn connect(addr: Address, port: Port) -> Chunnel::Conn {
  bertha::new("client_conn", wrap!()).connect(addr, port)
}
// Gets a key from the key-value store.
fn get_key(k: Key) -> Value {
  let c = connect(kv_addr, kv_port);
  c.send(Kvs::Serialize(Kvs::Op::Get, k));
}
```

Listing 5: **Code for a sharding client. The client endpoint specifies no Chunnels, and the set of Chunnels used is dictated entirely by the server.**

can turn into a bottleneck in the former, while the latter greatly complicates resharding. Thus, it might be beneficial to use an approach that allows both modalities, using client sharding when resharding is unlikely and load balancing otherwise. Unfortunately, current interfaces make it hard to deploy such a hybrid approach.

Bertha enables the use of hybrid load balancing approaches by binding to an implementation only when a connection is established. Furthermore, as we showed above, Bertha also provides Chunnels implementing ordered multicast for state machine replication (RSM) for fault tolerance.

4 Design

We next describe Bertha's architecture, and illustrate it by walking through how Bertha instantiates a connection. We do so in the context of a sharded key-value store (Listing 4) and its client (Listing 5).

In Bertha, applications register fallback implementations for Chunnels when launched. For example, Line 2 of Listing 5 shows the client registering a reliable transport Chunnel type. The reliability Chunnel is specified by the server connection, and is thus implicitly used by the client connection. In practice, we expect Bertha applications will link against libraries that provide fallback implementations for common Chunnels, e.g., mTCP [30] for a TCP Chunnel.

Fallback implementations should be designed to execute on all hosts where an application can be deployed, and should thus only assume access to widely-supported software and hardware capabilities such as SIMD instructions. As explained in §3, operators can register accelerated variants, including ones which use offloads with the Bertha discovery service (§4.2). The Bertha runtime is responsible for selecting an appropriate implementation during connection negotiation (§4.3), which occurs when the connection is established (line 10).

4.1 Bertha Runtime

The Bertha runtime, an application library, is responsible for implementing the Chunnel DAG specified by applications when establishing a connection. To do so, it takes as input the Chunnel DAG specified by the application, and queries the Bertha discovery service (§4.2) to find all available implementations for each Chunnel type in the DAG. The set of implementations found might includes ones that run in software on one (or more) connected end-hosts, in SmartNICs, or in a programmable switch or other device in the network. The runtime then uses the negotiation protocol §4.3 to decide which of these implementations to use. We envision that the Bertha runtime will eventually also enable interoprability with other network APIs including UNIX sockets, but we defer this question to the future.

Finally, the Bertha runtime, which has access to the entire DAG of Chunnels that comprise a connection can transform Chunnel DAG, e.g., by combining several Chunnels, in order to further optimize connections. We discuss this and other optimization opportunities in §6.

4.2 Bertha Discovery

The Bertha discovery service is responsible for tracking the set of implementations available for each Chunnel type. Offload developers (or network operators and system administrators) can register implementations for a Chunnel type by interacting with the Bertha discovery service; the Bertha runtime queries the discovery service in order to determine available implementations.

Chunnel implementations specify scoping constraints—e.g., a Chunnel can only be implemented on the same host as an application (`bertha::scope::Application`)—and constraints on where it must be implemented—e.g., whether the Chunnel requires functionality at both ends (`endpoints::Both`) of a connection. Implementations also provide initialization and teardown functions and a function that returns an implementation priority and set of resource requirements. The latter two are used when deciding which implementation to use at runtime.

A Chunnel's initialization function is responsible for configuring the system and network so that that the application can use the selected Chunnel implementation. The teardown function is similarly responsible for changing system and network configuration when an application terminates. These functions thus abstract over and automate tasks performed by system and network operators today. They can e.g., call operating system tools (e.g., `ethtool`) or invoke APIs on orchestrators and SDN controllers which are commonly deployed in current clusters. Note that initialization and teardown might vary across deployments for a given Chunnel type.

4.3 Chunnel Negotiation

During Chunnel negotiation the runtime first checks that a connection is feasible, i.e., DAGs on both ends are compatible and hosts are reachable. The runtime then chooses among the available implementations for each Chunnel a connection uses. Bertha makes this choice based on each implementation's priority and resource requirements and an operator-supplied policy function. In our current prototype, we assume a simple policy function that prefers client-provided implementations over server-provided implementations, and set implementation priorities to prefer kernel bypass and hardware accelerated implementations over standard implementations. We expect to support more complex policies in the future (§6).

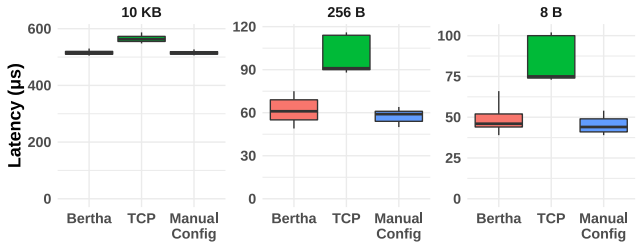

Figure 3: **Bertha allows the client to take advantage of more efficient IPC than inter-container TCP connections on the same host (in this case, Unix pipes) when available, without configuration. The boxplots show the median latency, with boxes extending to p25 and p75 and whiskers to p5 and p95. Note the different y-axis scales. The RPC latencies the Bertha client achieves are comparable to an application with Unix pipes.**

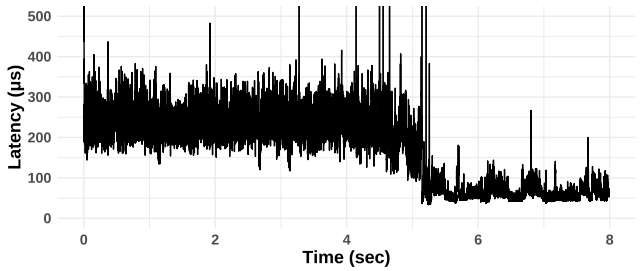

Figure 4: **Since clients resolve names dynamically, they can discover closer server instances without additional configuration.**

In the case of Listing 5, the negotiation process for the reliability Chunnel first checks whether compatible implementations are available at both client and server; the connection fails in the absence of the implementations. The runtime then chooses the highest priority implementation at each of the server and the client.

5 Benefits

The main benefit of adopting Bertha is that it simplifies the path for applications to adopt newer software and hardware-based offloads. In this section we focus on quantifying the benefit of adopting such advances. For this position paper, we focused on quantitative benefits in the absence of hardware offloads. We plan to explore offloads based on SmartNICs and other in-network components in the future.

Container Networking. We implemented the container fast-path Chunnel described in §3.2 in 750 lines of Rust. The Chunnel is restricted to the host scope. Connections that use this Chunnel and connect applications on the same host transfer data using UNIX named sockets, while connection between applications on different hosts make use of standard network sockets.

In Figure 3, we evaluate the benefit of this approach using a simple ping application and varying request sizes. In this experiment, a client makes a connection to the server on the same host, and measures the latency of 3 requests on that connection. We repeat this measurement across 10000 connections. Establishing a Bertha connection requires two additional IPC round trips to query the discovery service and negotiate the connection mechanism. However, subsequent messages on an established connection do not encounter additional latency. Despite this overhead, the Bertha implementation has latency similar to a specialized implementation that hardcodes the use of IPCs.

Dynamic Name Resolution. Because the `route_local` Chunnel checks whether a local server instance is available each time a connection is established, it allows clients to switch over to host-local instances when available. This is analogous to the behavior provided by IP anycast and the Bertha anycast Chunnel (§3.2). In Figure 4, when the client starts, the only server running is placed on a remote machine. As a result, it uses the full network stack when sending RPC requests, and they traverse the network. At $t = 4$ sec., an instance of the server is started locally; subsequent client connections choose the local instance and communicate using UNIX domain sockets. As a result, the subsequent requests have lower latency.

Sharding. We implement a sharding Chunnel in 2100 lines of Rust and 200 lines of C (an XDP fallback implementation). Our server application is a key-value store which uses the hashmap implementation from Rust's standard library and serialization from the widely-used `bincode` crate atop UDP RPCs. When the server creates a sharding Chunnel, it provides a list of shards and a sharding function as input. When the server listens on a connection with a sharding Chunnel it provides a canonical address on which the server listens for connections. We register three different shard Chunnel implementations: an accelerated XDP implementation run on the same machine as the server, and in-application fallback implementations in the server and client.

We evaluate the sharding Chunnel in Figure 5. Our evaluation uses two clients and one server[2]. We implement shards using threads, assigning one thread per shard. We measure the p95 latency over 300000 YCSB [13] requests (workload A, read-heavy) with a uniform distribution of keys. We evaluate performance in four scenarios:

Client Push The client applications uses the fallback implementation to compute the correct shard and the forwards the request directly to the shard. This improves sharding scalability, and eliminates bottleneck. This scenario is also a case where the presence of a fallback implementation improves performance, even in the absence of offloads.

Server Accelerated Shards are computed at the server using the XDP sharding implementation. All clients forward requests to the XDP sharding process, which then forwards it to the appropriate shard. This reduces client overheads, but creates a bottleneck at the server.

Mixed One client uses client push, while the other uses the server accelerated implementation. This shows a case where differences in client configuration result in different implementations being picked by different connections. The performance in this case is a mix of the previous two versions.

Server Fallback Finally, we show the case where the server's fallback implementation is used. The lack of an accelerated implementation and the need to handle traffic from all clients, results in poor performance, but still provides correctness.

Our results show that Bertha can switch between accelerated implementations without performance degradation, and demonstrate the benefits of these different approaches.

[2]Each with 4-core Intel Xeon E5-1410, 64 GB of memory, Linux kernel version 5.4.0, and a Mellanox ConnectX3-Pro NIC.

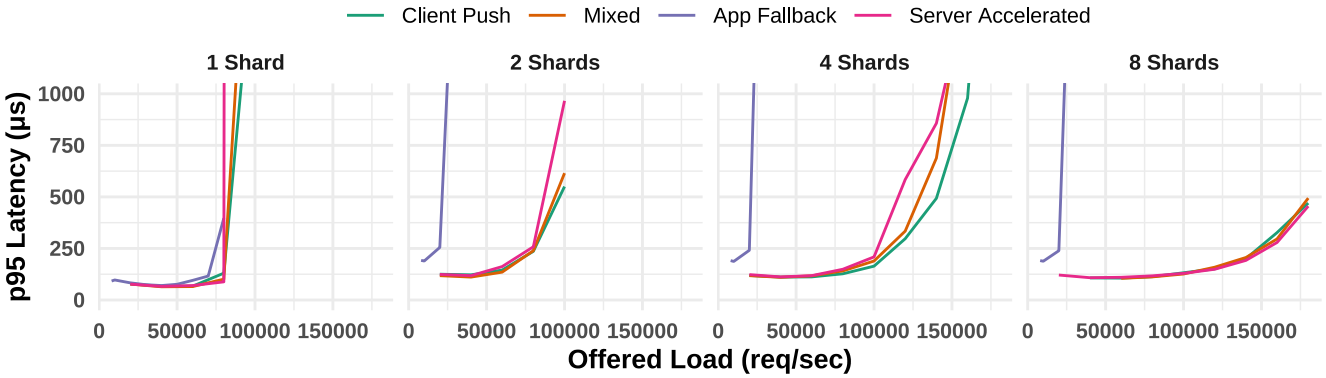

Figure 5: **Exposing sharding information to clients allows the user to dynamically choose between using client and server implementations.**

6 Research Directions

We now discuss some open questions related to Bertha.

Scheduling and Placement. One of Bertha's responsibilities is to aid in Chunnel negotiation, the process by which Chunnel implementations are chosen. Making this decision involves not only a policy specification from the application, but also knowledge about network state and resource utilization. For example, if two programs can benefit from offloading functionality to a P4 switch, but the switch only has capacity for one, the Bertha runtime must choose between these two applications. Note that Chunnel priorities alone are insufficient to accomplish this goal. This problem exists even when using recently proposed techniques [33, 34, 42, 62, 64] including SR-IOV for sharing hardware resources. One approach to addressing this challenge is to borrow techniques from the multi-resource scheduling literature [19, 20, 39], and we plan to examine this in future work.

Performance Optimization. In what we described thus far, application developers are responsible for deciding what Chunnels to use, and the order in which they are applied. The Bertha runtime, which has visibility into the entire sequence of Chunnels a connection's data traverses, enables optimization techniques which further improve performance. Possible optimizations include (a) reordering the DAG in order to reduce the amount of data transferred between offloads (e.g., PCIe bus and network traversals); (b) combining multiple Chunnels in order to take advantage of hardware capabilities; (c) eliminating unnecessary or redundant Chunnels; and (d) specializing Chunnel implementations based on their operating context.

For example, consider a Bertha connection with the pipeline `encrypt |> http2 |> tcp` running on a host where a SmartNIC can be used to offload encryption and TCP functionality. When implemented as specified, the Bertha runtime must either use a fallback implementation for encryption or incur a 3× increase (NIC-CPU-NIC) in the amount of data sent over PCIe to the NIC. Reordering this pipeline as `http2 |> encrypt |> tcp` allows the use of the offloaded implementation without increased PCIe overhead. Because Bertha decides on a Chunnel implementation when establishing a connection and in coordination with all endpoints, reordering is safe in this case. Similarly, if the SmartNIC did not explicitly offer separate offloads for encryption and TCP, but did offer one for TLS, Bertha could reorder and then merge the last two Chunnels, allowing the application to take advantage of the TLS offload. Bertha can similarly also adopt techniques similar to Floem [51] to split a complex Chunnel and partially offload its implementation. DAG optimization techniques have been used in other domains including data-intensive processing [50] and deep neural nets [24], but have so far not been applied to the network dataplane.

Deployment Concerns. How does Bertha operate when connecting hosts in different ASes? The key challenge is trust, since a host might end up relying on a Chunnel implementation in a different network. A possible solution is to adopt techniques such as program attestation [9, 27, 36] and proof carrying code [43, 44], which allow remote hosts to check semantics of the running program. Unfortunately, translating these techniques so they can be used with constrained hardware devices such as switches, FPGAs, and SmartNICs is challenging and requires additional research.

7 Related Work

Our approach closely resembles the approach taken by ONNX [49], Tensorflow [1] and other machine learning frameworks which represent models as DAGs and enable offload-use when possible.

Our work is enabled by the long line of prior work on active networking [60], network function virtualization (NFV) [16], and programmable switches and SmartNICs [10, 31, 37, 53, 58] that enable programmability in the network fast path. Recent work on virtualizing programmable network devices and accelerators, including HyPer4 [25], P4Visor [65], AmorphOS [32] and AvA [63] have made it easier to share these offloads among applications. Our focus in this work has been on simplifying the use of these advances in applications.

Finally, the widespread adoption of software-defined networking (SDN) [40] has prompted the development of languages including Frenetic [18], Flowlog [45], VeriCon [8], Merlin [59], etc. which provide programmatic control over network configuration. This line of work, which focuses on an interface for changing *network configuration* is complementary to our work and might be useful during Chunnel initialization.

8 Acknowledgements

We thank the anonymous reviewers and the following people for their valuable feedback: Srinivas Narayana, Amy Ousterhout, Shoumik Palkar, Deepti Raghavan, Justine Sherry, Anirudh Sivaraman, and Shivaram Venkataraman. This work was supported in part by funding from VMware, Intel, and NSF grants 1407470, 1526791, 1563826, 1704941, 1817115, 2028832, and 2029037.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] Amazon. Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/.

[3] Andrew Morgan. How MySQL is able to scale to 200 Million QPS - MySQL Cluster. http://highscalability.com/blog/2015/5/18/how-mysql-is-able-to-scale-to-200-million-qps-mysql-cluster.html.

[4] Apache. Arrow. https://arrow.apache.org/.

[5] Apache. Thrift. https://thrift.apache.org/.

[6] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs . In *NSDI*, 2020.

[7] C. P. Authors. Cap'n Proto Cerealization Protocol. https://capnproto.org/.

[8] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-Defined Networks. In *PLDI*, 2014.

[9] A. Baumann, M. Peinado, and G. C. Hunt. Shielding applications from an untrusted cloud with haven. In *TOCS*, 2015.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[11] Bruce Curtis. net-tcp: TCP/IP stack bypass for loopback connections. https://www.spinics.net/lists/netdev/msg210741.html.

[12] Chelsio Communications. TCP Offload Engine (TOE). http://www.chelsio.com/nic/tcp-offload-engine/.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[14] W. J. Dally, Y. Turakhia, and S. Han. Domain-specific hardware accelerators. *CACM*, July 2020.

[15] DPDK Authors. DPDK. https://www.dpdk.org/.

[16] ETSI. Network Functions Virtualisation. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[17] F5 Labs. Intelligent Application traffic Management. https://www.f5.com/products/big-ip-services/local-traffic-manager.

[18] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[19] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*, 2011.

[20] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *OSDI*, 2016.

[21] Google. Flatbuffers. https://google.github.io/flatbuffers/.

[22] Google. gRPC. https://grpc.io/.

[23] Google. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[24] Google. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla/.

[25] D. Hancock and J. E. van der Merwe. HyPer4: Using P4 to Virtualize the Programmable Data Plane. *CoNEXT*, 2016.

[26] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *CoNEXT*, 2018.

[27] M. Jakobsson. Secure remote attestation. *IACR Cryptol. ePrint Arch.*, 2018:31, 2018.

[28] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *ISCA*, 2020.

[29] Jeff Barr. New – Advanced Request Routing for AWS Application Load Balancers. https://aws.amazon.com/blogs/aws/new-advanced-request-routing-for-aws-application-load-balancers/.

[30] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[31] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*, 2017.

[32] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *OSDI*, 2018.

[33] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI*, 2019.

[34] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, and A. Vahdat. PicNIC: Predictable Virtualized NIC. In *SIGCOMM*, 2019.

[35] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*, 2017.

[36] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *EuroSys*, 2020.

[37] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*, 2016.

[38] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM CCR*, July 2015.

[39] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *SIGCOMM*, 2019.

[40] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, Mar. 2008.

[41] Microsoft. Fast TCP Loopback Performance and Low Latency with Windows Server 2012 TCP Loopback Fast Path. https://docs.microsoft.com/en-us/archive/blogs/wincat/fast-tcp-loopback-performance-and-low-latency-with-windows-server-2012-tcp-loopback-fast-path.

[42] Microsoft. Introduction to Hyper-V on Windows 10. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/.

[43] G. C. Necula. Proof-carrying code. In *POPL*, 1997.

[44] G. C. Necula and P. Lee. Safe Kernel Extensions without Run-Time Checking. In *OSDI*, 1996.

[45] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *NSDI*, 2014.

[46] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[47] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An end-host stack for service-centric networking. In *NSDI*, 2012.

[48] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A Platform for High-Performance Internet Applications. *SIGOPS Oper. Syst. Rev.*, August 2010.

[49] ONNX. Open Neural Network Exchange. https://onnx.ai/.

[50] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, S. Madden, and M. Zaharia. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. In *VLDB*, 2018.

[51] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *OSDI*, 2018.

[52] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *NSDI*, 2015.

[53] D. R. K. Ports and J. Nelson. When Should the Network be the Computer? In *HotOS*, 2019.

[54] ProxySQL. ProxySQL: A High Performance Open Source MySQL Proxy. https://proxysql.com/.

[55] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[56] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM ToCS*, Nov. 1984.

[57] J. Sherry, D. Kim, S. Mahalingam, A. Tang, S. Wang, and S. Ratnasamy. Netcalls: End Host Function Calls to Network Traffic Processing Services. UC Berkeley Technical Report No. UCB/EECS-2012-175. http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-175.html, 2012.

[58] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[59] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Provisioning Network Resources. In *CoNEXT*, 2014.

[60] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *SIGCOMM CCR*, October 2007.

[61] M. Walfish, J. Stribling, M. N. Krohn, H. Balakrishnan, R. T. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, 2004.

[62] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. Ports, and A. Panda. Multitenancy for Fast and Programmable Networks in the Cloud. In *HotCloud*, 2020.

[63] H. Yu, A. M. Peters, A. Akshintala, and C. J. Rossbach. AvA: Accelerated Virtualization of Accelerators. *ASPLOS*, 2020.

[64] P. Zheng, T. Benson, and C. Hu. P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *CoNEXT*, 2018.

[65] P. Zheng, T. Benson, and C. Hu. P4Visor: lightweight virtualization and composition primitives for building and testing modular programs. *CoNEXT*, 2018.

[66] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *NSDI*, 2019.