A Hardware Accelerator for Protocol Buffers

Sagar Karandikar
UC Berkeley, Google
USA

Chris Leary
Google
USA

Chris Kennelly
Google
USA

Jerry Zhao
UC Berkeley
USA

Dinesh Parimi
UC Berkeley
USA

Borivoje Nikolić
UC Berkeley
USA

Krste Asanović
UC Berkeley
USA

Parthasarathy Ranganathan
Google
USA

ABSTRACT

Serialization frameworks are a fundamental component of scale-out systems, but introduce significant compute overheads. However, they are amenable to acceleration with specialized hardware. To understand the trade-offs involved in architecting such an accelerator, we present the first in-depth study of serialization framework usage at scale by profiling Protocol Buffers ("protobuf") usage across Google's datacenter fleet. We use this data to build HyperProtoBench, an open-source benchmark representative of key serialization-framework user services at scale. In doing so, we identify key insights that challenge prevailing assumptions about serialization framework usage.

We use these insights to develop a novel hardware accelerator for protobufs, implemented in RTL and integrated into a RISC-V SoC. Applications can easily harness the accelerator, as it integrates with a modified version of the open-source protobuf library and is wire-compatible with standard protobufs. We have fully open-sourced our RTL, which, to the best of our knowledge, is the only such implementation currently available to the community.

We also present a first-of-its-kind, end-to-end evaluation of our entire RTL-based system running hyperscale-derived benchmarks and microbenchmarks. We boot Linux on the system using FireSim to run these benchmarks and implement the design in a commercial 22nm FinFET process to obtain area and frequency metrics. We demonstrate an average 6.2× to 11.2× performance improvement vs. our baseline RISC-V SoC with BOOM OoO cores and despite the RISC-V SoC's weaker uncore/supporting components, an average 3.8× improvement vs. a Xeon-based server.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; **Cloud computing**; • **Hardware** → **Communication hardware, interfaces and storage**; **Application-specific VLSI designs**.

MICRO '21, October 18–22, 2021, Virtual Event, Greece
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480051

KEYWORDS

serialization, deserialization, hardware-acceleration, warehouse-scale computing, hyperscale systems, profiling

ACM Reference Format:
Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolić, Krste Asanović, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3466752.3480051

1 INTRODUCTION

As traditional hardware scaling slows, scale-out systems become increasingly attractive for resource-intensive workloads. However, harnessing the power of scale-out platforms requires dealing with datacenter-specific overheads, collectively dubbed the "datacenter tax" [28]. Many of these overheads stem from the fundamental need to communicate between software components (i.e., services) in a distributed environment, which is commonly achieved via remote procedure call (RPC). Because the remote callee cannot directly access the caller's memory space to read arguments and supply a response, and may even be written in a different programming language, exchanged data must undergo conversion to and from a shared interchange format, via *serialization* and *deserialization* operations. In addition to inter-service communication via RPC, serialization and deserialization are also commonly used when persisting data to durable storage.

To ensure that serialization and deserialization are handled in a principled way across the multitude of services and data producers/consumers running in a warehouse-scale computer, service developers employ a common serialization framework, which ensures interoperability between components by pairing a standardized wire format with language-specific APIs that allow applications to produce and consume serialized objects. A vast number of these frameworks have been created [1, 5, 7–11, 13], constituting a large design space encompassing trade-offs in performance, flexibility, ease-of-use, backwards compatibility, and schema evolution. In a hyperscale context, backwards compatibility and schema evolution become particularly important to manage complexity, build reliable systems, and ensure long-term accessibility of data persisted to durable storage [12, 14].

Naturally, this functionality comes at a performance cost—prior work has shown that around 5% of fleet-wide cycles in Google's

Warehouse Scale Computers (WSCs) were spent in the Protocol Buffers ("protobuf") serialization framework in 2015 [28]. In 2020, Facebook identified that serialization and deserialization consume on average over 6% of cycles across seven key microservices in their fleet [40].

Fortunately, the warehouse-scale context is a natural environment for hardware specialization [2, 19, 21, 22, 27, 31, 38] as the cost of building custom processors is amortized over the high volume of deployed hardware systems. To understand the trade-offs and opportunities in hardware acceleration for serialization frameworks, we present the first in-depth study of serialization framework usage at scale by characterizing protobufs usage across Google's datacenter fleet (Section 3) and use this data to construct HyperProtoBench, an open-source[1] benchmark representative of key serialization-framework user services at scale (Section 5.2). In doing so, we also identify key insights that challenge common assumptions about serialization framework usage (Section 3.9).

We use these insights to co-design hardware and software to develop a novel hardware accelerator for protobuf message serialization and deserialization, implemented in Chisel RTL [18] and integrated into a Linux-capable RISC-V SoC [15] (Section 4). Applications can easily harness the accelerator, as it integrates with a modified version of the open-source protobuf library and is wire-compatible with standard protobufs. We have fully open-sourced[2] our RTL, which, to the best of our knowledge, is the only such implementation currently available to the community.

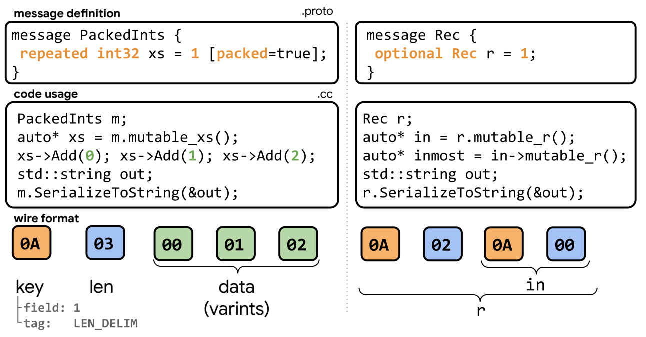
We also present a first-of-its-kind end-to-end evaluation of our entire RTL-based system running hyperscale-derived benchmarks and microbenchmarks (Section 5 and Appendix A). We boot Linux on the system using FireSim [29] to run these benchmarks and implement the design in a commercial 22nm FinFET process to obtain area and frequency metrics. We demonstrate an average 6.2× to 11.2× performance improvement vs. our baseline RISC-V SoC with BOOM OoO cores [43] and despite the RISC-V SoC's weaker uncore/supporting components, an average 3.8× improvement vs. a Xeon-based server.

In addition to advancing the state-of-the-art in serialization framework acceleration, this work is the first to demonstrate the power of combining a data-driven hardware-software co-design methodology based on large-scale profiling with the promise of agile, open hardware development methodologies [24, 30]. In this vein, our entire evaluation flow (RTL, benchmarks, including hyperscale-derived benchmarks, and supporting software and simulation infrastructure) has been open-sourced for the benefit of the research community and our results have been reproduced by external artifact evaluators (Appendix A).

2 PROTOBUF BACKGROUND

The protobuf library is an open-source, schema-oriented, data and service description system [11]. Protobufs are widely used for service-oriented design in modern hyperscale systems, including at Google. Protobufs are also used for in-memory data representation, persisting data to durable storage, and as a schema for columnar storage (e.g. Google's Dremel/BigQuery [32, 33]). A protobuf user

[1]https://github.com/google/HyperProtoBench. See Appendix A for archival URL.
[2]https://github.com/ucb-bar/protoacc. See Appendix A for archival URL.



```
message PackedInts {
 repeated int32 xs = 1 [packed=true];
}
```

```
message Rec {
 optional Rec r = 1;
}
```

```
PackedInts m;
auto* xs = m.mutable_xs();
xs->Add(0); xs->Add(1); xs->Add(2);
std::string out;
m.SerializeToString(&out);
```

```
Rec r;
auto* in = r.mutable_r();
auto* inmost = in->mutable_r();
std::string out;
r.SerializeToString(&out);
```

Figure 1: Encodings with repeated and recursive types. Empty messages (`inmost`) take no bytes in encoded form.

Performance-similar Types	**Protobuf Type** (includes repeated of each type)	**Sizes** (bytes)
bytes-like	`bytes`, `strings`	See Fig. 4c buckets
varint-like	`{s,u}int{64,32}`, `int{64,32}`, `enum`, `bool`	1-10, by 1
float-like	`float`	4
double-like	`double`	8
fixed32-like	`fixed32`, `sfixed32`	4
fixed64-like	`fixed64`, `sfixed64`	8

Table 1: Classification of protobuf field types.

defines the contents of a message in a `.proto` file written in the protobuf language, either `proto2` or `proto3`[3]. The protobuf compiler (`protoc`) ingests `.proto` files and generates language-specific code to allow user programs to populate, read, and perform other operations on protobuf messages.

2.1 Message structure

2.1.1 Schema and message definition. A protobuf message is a collection of fields. In the protobuf schema, each message field has a type, name, field number, and potential qualifiers including `optional`, `required`, and `repeated` (with `packed` for a more efficient encoding). Scalar field types include doubles, floats, various variable and fixed-width integer types, bools, strings, and bytes. The "Protobuf Type" column in Table 1 lists these types. A field's type can also be a user-defined message type, allowing for messages to contain sub-messages; messages may be nested arbitrarily deeply and recursively structured. The `repeated` qualifier marks that a field is a vector of elements of its assigned type, which can also be a user-defined message type. The top row of Figure 1 shows two example message definitions.

This structure enables forward portability and schema evolution. Namely, fields are numbered for stability across field name changes, and fields may be optionally present, enabling sparsity for deprecated/unused fields. Schema evolution, upgrade paths, and host language integration are critical for productively using a serialization framework at hyperscale, where services cannot be

[3]As discussed in Section 3.3, the vast majority of protobuf usage in Google's fleet is `proto2`. Thus, "protobufs" implicitly refers to `proto2` in the rest of this paper.

monolithically upgraded, and persisted data must be highly available for long periods of time [14].

2.1.2 Wire format. Before we discuss the wire format, it is important to note that variable-length integers ("varints") are used heavily in the protobuf wire format. The protobuf varint algorithm repeatedly consumes 7 bits at-a-time in a loop from the least-significant side of a fixed-width input value until no non-zero bits remain. For each 7-bit group, it outputs a byte containing the original bits and a continuation bit, which if set, indicates that more bytes follow. As we will see, varint handling is a prime candidate for acceleration—fixed-function hardware can easily handle varint encoding/decoding in a single cycle.

On the wire, protobuf messages appear as a sequence of bytes containing a set of (key, value) pairs that represent fields in the message. Each field's key is a varint-encoded version of the field number concatenated with a three-bit *wire* type. Wire types can be one of: varint (field types `{s,u}int{64,32}`, `int{64,32}`, `enum`, and `bool`), 64-bit (field types `double` and `(s)fixed64`), length-delimited (field types `string`, `bytes`, sub-messages, and packed repeated fields), start group (deprecated), end group (deprecated), and 32-bit (field types `float` and `(s)fixed32`). A critical observation from this mapping is that the wire type is *not* sufficient to determine the language/schema type of a field. For the 32-bit and 64-bit wire types, C++ values are directly copied into the wire format. For the varint wire type, the *varint encoding* is applied to the C++ values before they are copied to the wire format. The values of the length-delimited wire type first contain a varint-encoded length in bytes, which represents either the length of a string or byte array, the length of a sub-message, or the length of a packed repeated field. This length is followed by either the string or bytes data, the wire-format version of a sub-message, or encoded values in a packed repeated field. Finally, unpacked repeated fields appear on the wire as multiple (key, value) pairs that all have the same key. The bottom row of Figure 1 shows examples of two messages encoded in the wire format.

2.1.3 In-memory format. As previously mentioned, given a message schema, the protobuf compiler will generate language-specific code for each message type. For example, for C++, the compiler generates a class for each message type which encapsulates the field data. Users expect to work with protobuf messages as standard C++ objects: scalar fields are stored as the expected C++ primitive type, string/byte fields are stored as `std::string`s, repeated fields are stored similar to vectors, and sub-messages are stored as pointers to objects of their corresponding type. All members are wrapped in accessors (e.g., setters, getters). The middle row of Figure 1 shows examples of two messages used in C++ code.

2.2 Serialization and deserialization

The two key operations in protobufs are *serialization* and *deserialization*. Serialization converts the in-memory, language-specific protobuf message representation (Section 2.1.3) to the standard protobuf wire format (Section 2.1.2). This wire-format version of a message can then be exchanged with any other program that uses protobufs, regardless of programming language, host machine, operating system, and compiler. To unpack the wire format into a usable object again, the *de*serialization process converts a wire-format message back to the in-memory language-specific protobuf object.

Serialization and deserialization are inverse operations, but deserialization is more complex for two reasons. Firstly, deserialization is inherently a serial process: the deserializer receives a single stream of bytes and the key (and potentially, value) of the N^{th} field in the encoded format must be decoded before the $(N+1)^{th}$ field, as the location of the $(N+1)^{th}$ field is unknown until the size of the N^{th} field is known (based on wire-type or explicit length). On the other hand, serialization has ample opportunity for parallelism: serialization of individual fields can be performed in parallel with one final serial step that concatenates the serialized fields into one output buffer.

Secondly, deserialization requires the accelerator to construct objects in the in-memory language format (including e.g., `std::string` objects in C++) and allocate memory for them; serialization only needs to *traverse* language-format objects.

2.3 Arena allocation

One notable performance optimization available in upstream protocol buffers is arena allocation [4], which reduces message construction/destruction overheads by pre-allocating a large chunk of memory called the *arena*. Allocation of individual messages in the arena is simplified to a pointer increment. The accelerator we implement uses its own form of arena allocation, as discussed in Section 4.3.

3 PROFILING PROTOBUF USAGE AT SCALE

In this section, we explore the usage of Protocol Buffers at scale across Google's datacenter fleet to motivate requirements for a hardware accelerator for serialization and deserialization, and quantify accelerator design trade-offs.

3.1 Data sources

We rely on three internal data sources at Google to glean insights on protobuf usage at scale: Google-Wide Profiling (GWP) CPU cycle profiles, `protobufz`, and `protodb`.

3.1.1 Google-Wide Profiling (GWP) CPU cycle profiles. CPU cycle profiles are collected from machines across Google's fleet using Google-Wide Profiling (GWP) [39]. The collected profiles include workload names, stack traces, and cycle counts, which allow us to identify where CPU time is spent in software. In particular, this data allows us to identify how much time is spent in different operations inside the protobuf library and generated code, including serialization, deserialization, and others.

3.1.2 `protobufz`*.* The `protobufz` sampler provides dynamic (i.e., runtime) information about the structure of protobuf messages that are serialized and deserialized throughout the software stack running on Google's datacenter fleet. GWP randomly chooses machines to visit; when a machine is visited, the protobuf message sampler runs for several seconds and randomly selects top-level messages to be sampled. A top-level message is defined as a message on which deserialize or serialize is called directly; that is, a sub-message only appears in the data if its parent is also chosen.

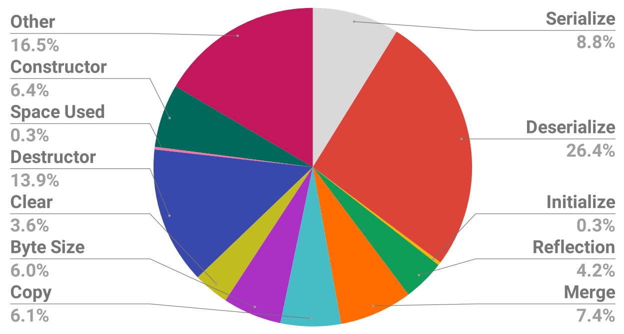


Figure 2: Fleet-wide C++ protobuf cycles by operation.

When a top-level message is sampled, complete information about the message and its sub-messages is captured. This includes sizes and types of all present fields, including fully-qualified names for sub-message types. The protobufz data also includes the path of the .proto file in which the protobuf message is defined. This allows reconstruction of the complete hierarchy of a sampled message and joining the dynamic protobuf structure data with other data sources.

3.1.3 protodb. The protodb database provides static information about all .proto files defined in Google's codebase. This allows us to collect detailed information about each defined message type, such as the version of the protobufs language a message is defined against, whether repeated fields are packed, and the range of field numbers defined in a message.

3.2 What is the opportunity for fleet-wide CPU cycle savings?

Using GWP CPU cycle profiles, we find that protobuf operations constitute 9.6% of fleet-wide CPU cycles in Google's infrastructure. These cycles are dominated by C++ protobuf usage: 88% of fleet-wide protobuf cycles are spent in *C++* protobufs. As a result, we will focus on C++ protobufs in the rest of this work. Section 7 discusses future support for other languages.

Figure 2 shows the classification of cycles spent within C++ protobufs, by operation. A few notable items are immediately visible. Firstly, deserialization alone is a significant contributor to overall CPU cycles—2.2% of fleet-wide CPU cycles are spent in C++ protobuf deserialization. Serialization cycles are also significant, with serialization in C++ consuming 1.25% of fleet-wide CPU cycles[4]. Because these are relatively coarse-grained operations, they are natural avenues to explore for acceleration opportunities. The "other" operator in Figure 2 represents a miscellany of glue code that is not clearly amenable to acceleration. This work focuses on the task of accelerating C++ protobuf serialization and deserialization, presenting the opportunity to accelerate/offload 3.45% of CPU cycles across Google's fleet. Section 7 discusses several other protobuf operations, which are relatively straightforward to accelerate once deserialization and serialization are handled.

[4]Virtually all calls to Byte Size occur during serialization, so this accounts for Serialization's 8.8% of protobuf cycles and Byte Size's 6.0% of protobuf cycles in Figure 2.

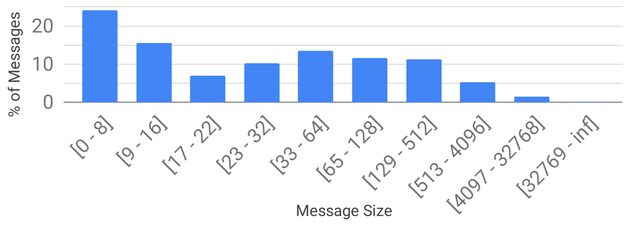


Figure 3: Fleet-wide top-level message size distribution.

3.3 Which proto version should we implement?

As discussed in Section 2, two versions of the Protocol Buffers language are currently supported, proto2 and proto3. Although proto3 was released in mid-2016, 96% of protobuf bytes serialized/deserialized in Google's fleet remain defined in the proto2 language. Therefore, we target proto2 in our accelerator design. This also suggests that usage of serialization framework APIs and formats tends to be stable over time, making hardware acceleration viable.

3.4 Should we optimize accelerator placement for the RPC stack?

To understand where to place a protobuf accelerator in the system (e.g., in-core, near-core, as a bus peripheral, CXL, PCIe, etc.), we would like to know how serializations and deserializations are initiated. One commonly assumed source of protobuf usage is the RPC stack. In Google's fleet, we find that only 16.3% of deserialization cycles are from the RPC stack and only 35.2% of serialization cycles are from the RPC stack. This challenges the common assumption that a protobuf accelerator should be placed on a PCIe-attached NIC. Instead, it is clear that other serialization and deserialization users (e.g. storage users) must be accounted for when deciding where to place a protobuf accelerator in the system.

3.5 What is the granularity of operations the accelerator needs to handle?

Another factor when deciding accelerator placement is understanding the offloading overhead that can be tolerated, which depends on offload granularity. While we do not have a mechanism to directly attribute cycle counts to individual serialization and deserialization operations, we can observe the distribution of top-level message sizes (including their sub-messages) as a proxy.

Figure 3 shows the distribution of message sizes observed in Google's fleet. Buckets are labeled with their inclusive byte bounds; that is, the [0 - 8] bucket counts the number of messages where the total encoded message size (including all sub-messages) was 0 to 8 bytes. Interestingly, the vast majority of messages are very small: 24% of messages are 8 bytes or less, 56% of messages are 32 bytes or less, and 93% of messages are 512 bytes or less. Based on this distribution, a near-core accelerator is likely necessary to efficiently handle the vast majority of messages. Also notable is that protobuf benchmarks used by prior work [36] tend to focus on only a small part (e.g., one bucket) of this distribution.

While message count is important, it is also important to keep in mind the *volume of data* in each of the message-size buckets. While we cannot directly collect this data due to infrastructure limitations, we can see that the `[32769 - inf]` bucket, which represents 0.08% of messages, contains *at least* 13.7× as many message *bytes* as the `[0 - 8]` bucket. This volume of data encoded in large messages could tolerate a higher offload overhead, while still observing a speedup. We will return to the discussion of accelerator placement trade-offs in Section 3.9.

3.6 What types of data movement and field encodings should the accelerator support?

In addition to the acceleration opportunities inherent in parsing or constructing protobuf message *structure* in hardware, there may be opportunities to speed up the processing of individual field values, depending on the commonly used field types in the fleet.

3.6.1 Which field types are most commonly used? The various protobuf field types discussed in Section 2 present differing opportunities for acceleration. For example, handling an `int64` field requires encoding or decoding two varints, the key and value, which is expensive in compute-per-byte terms on a CPU. On the other hand, handling a large `bytes` field is relatively cheap as it only requires encoding or decoding two varints, the key and length, and then `memcpy`ing a large amount of useful data.

Figure 4a shows the proportion of observed fields of the most frequently used primitive types across Google's fleet. In this plot, sub-messages are accounted for via the primitive fields they contain, but are not noted as separate fields themselves. Looking at field counts, we see very promising avenues for acceleration. Firstly, over 56% of fields are a form of varint (`int32`, `int64`, `enum`, `bool`, `uint64`), which are well-suited to acceleration. There are also a significant number of `string` and `bytes` fields, which can benefit from acceleration depending on field size.

3.6.2 Which field types account for the most data volume? Field *counts* do not necessarily present the full picture. Ideally, we would like to know the total number of CPU cycles spent serializing and deserializing each field type. Unfortunately, the fleet-wide profiling mechanisms do not provide this level of detail. However, as a proxy, we can instead obtain the number of bytes of data attributed to each field type, fleet-wide. Figure 4b presents this data.

Startlingly, we see a very different picture when looking at the weighted (by bytes of data) field-type breakdown. `Bytes`, `string`, and repeated `bytes` and `string` fields constitute over 92% of the bytes of protobuf messages handled. If these fields tend to be very large, then the cost of handling a varint (for the field's key) is relatively small compared to the cost of performing a `memcpy` and therefore there is less opportunity for acceleration beyond `memcpy` acceleration and offloading.

3.6.3 How large are `bytes` fields? To better understand the breakdown of this large amount of `bytes` and `string` data in protobuf messages, we collect data on the distribution of `bytes` field sizes, as shown in Figure 4c. Figure 4c uses the same bucket bounds as Figure 3; a slice labeled `0-8` in Figure 4c represents the percentage of `bytes` fields that were 0 to 8 bytes (inclusive) in size. Not labeled are the `4097-32768` and `32769-inf` buckets, which constitute 1.3% and 0.06% of observed fields respectively. In this view, we can see that small `bytes` fields dominate in terms of count, but data *volume* is a different story; the `32769-inf` bucket contains at least 7.2× as many *bytes* of data as the `0-8` bucket.

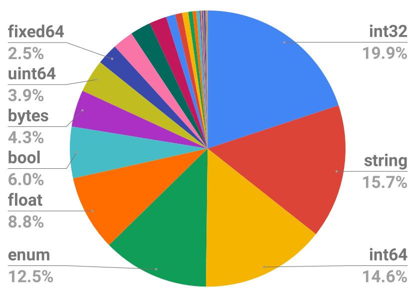
3.6.4 Which field types are responsible for the most CPU cycles in serialization and deserialization? The data so far paints a murky picture of where opportunities for protobuf acceleration lie. To better understand how time is spent in protobuf serialization and deserialization fleet-wide, we develop a model that converts from counts and bytes of different field types into CPU cycles (or time) spent handling each type. To enable this, we first group together protobuf field types that require a similar amount of "work" to be serialized or deserialized, as shown in Table 1. Within the bytes-like and varint-like groups, we subdivide by field size since as discussed earlier, size can have a significant impact on serialization and deserialization performance. For varint-like fields, the fleet-wide `protobufz` histogram data provides exact labels on size bins, so we can directly determine how much data each of the varint sizes (1 to 10 bytes) contribute to the overall number of protobuf message bytes. For bytes-like fields, the profiling system collects 10 buckets with ranges shown in Figure 4c. To interpolate field sizes from the buckets for bytes-like fields, we select the midpoint of each bucket to represent the size of each field in the bucket, and then adjust the size of the largest bucket (32769 to infinity bytes) as necessary to obtain the total number of bytes of bytes-like fields. Altogether, this process classifies the fleet-wide bytes-of-protobuf message data into 24 slices based on pairs of `[field-type-like, size]`.

Next, for each of these 24 pairs, we construct a protobuf microbenchmark to measure serialization and deserialization performance in terms of time spent per-byte of encoded data. Combining these results with the fleet-wide bytes-per-field-type data, we obtain estimated deserialization and serialization time (or cycles) spent per-field-type across Google's fleet.

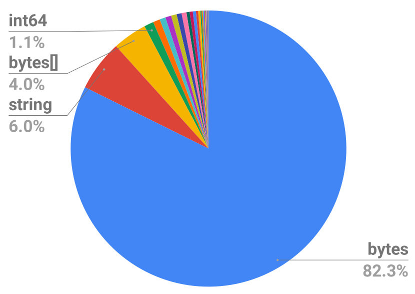
Figure 5 shows the estimated breakdown of deserialization time across the fleet. Several important insights can be derived from this analysis. Firstly, we notice that there is no single silver-bullet—*the accelerator will need to improve deserialization performance across the swath of field types and sizes.* Furthermore, the cases where the CPU performs best (large bytes-like fields) are a relatively small proportion of overall deserialization cycles—only 14% of time is spent deserializing protobuf data at higher than 1GB/s. While somewhat counter-intuitive, the difference in bytes-percentage between Figure 4b (amount of data) and Figure 5 (cycles) arises precisely because handling of large bytes-like fields on a CPU is so much faster per-byte than for example, a small varint-like or small bytes-like field; in our microbenchmarks, the large bytes-like field is 100-500x faster to handle per-byte. Figure 6 paints a similar picture for serialization. Although the largest byte bucket is relatively more significant than in the deserialization case, there is still ample opportunity in other field types. Overall, this analysis demonstrates that there are significant opportunities for acceleration in protobuf deserialization and serialization apart from fast `memcpy`.

3.7 What is the ideal accelerator programming interface?

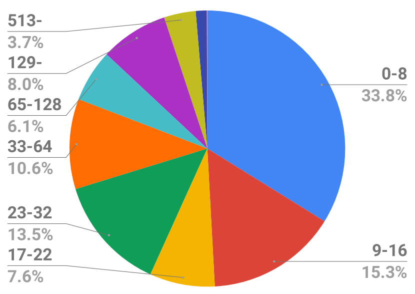
To enable serialization frameworks to generate programming information for a serialization/deserialization accelerator, prior work [36]



(a) % of fields observed by type.



(b) % of message bytes observed by type.



(c) % of bytes fields observed by field size.

Figure 4: Fleet-wide field type and bytes field breakdowns.

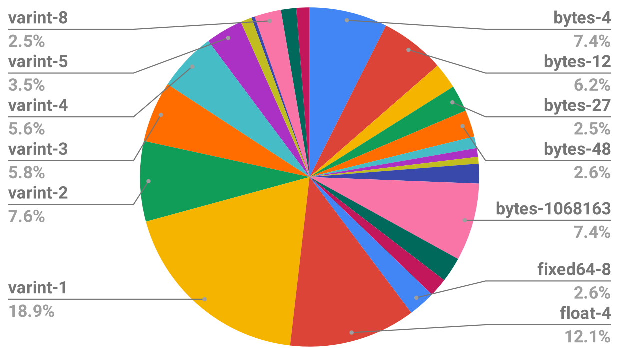


Figure 5: Estimated deser. time by field type, fleet-wide.

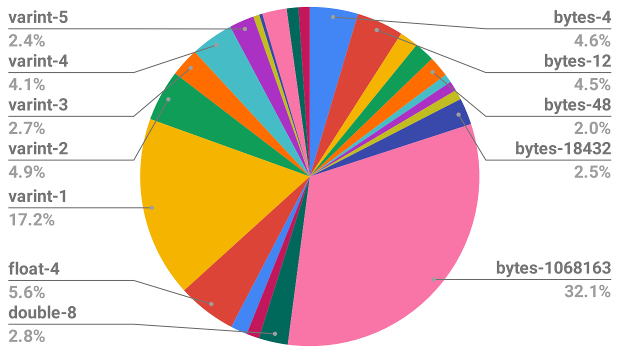


Figure 6: Estimated ser. time by field type, fleet-wide.

has suggested dynamically constructing per-*message-instance* programming tables of type/address (with implicit field presence) information for each populated field in a message to be serialized. While this can simplify accelerator implementation, this requires the protobuf compiler to add computationally expensive schema-management code to all generated field setters and clear methods that previously consisted of only cheap loads and stores. In contrast, our approach is to produce one Accelerator Descriptor Table (ADT) per-*message-type* (Section 4.2), resulting in a drastic reduction in programming table state. Our ADTs are automatically generated by the protobuf compiler and fully populated when the program is loaded, removing the need to inject costly schema-management code into all field setters and clear methods.

With our fixed, *per-message-type* ADTs, however, separate state is required to maintain field-presence information (i.e., whether or not a field has been set in a particular message object) for serialization purposes. We modify the internal per-message-instance `hasbits` bit field already generated by the protobuf compiler, to a *sparse* representation, so that the accelerator can directly index into it by field number.

More quantitatively, while prior work [36] *writes* an extra 64 bits per-present-field (a conservative assumption for the size of a schema entry) compared to our design, our design *reads* an extra bit per-field in the range of defined field numbers (due to the sparse `hasbits` representation) compared to the prior work. Thus, a *field number usage density* (= average # of present fields for a message type divided by the range of defined field numbers for that type) value of greater than $\frac{1}{64}$ (which falls in the "0.00" bucket in Figure 7) favors our accelerator design; Figure 7 shows that at least 92% of observed messages fleet-wide have a density greater than $\frac{1}{64}$, heavily favoring our accelerator design. We will build on this discussion in Sections 4.2 and 4.5.3, where we discuss our accelerator programming tables and serializer frontend design.

3.8 How do we size sub-message metadata tracking structures in the accelerator?

Another important question that will arise when designing a protobuf accelerator is that of handling sub-messages. Recursing into a sub-message in hardware requires maintaining additional state per-level of hierarchy (Section 4.4.9 and Section 4.5.3), which can become expensive. Fortunately, we find that across Google's fleet, 99.9% of bytes of protobuf data handled are at depth 12 or less, with 99.999% at depth 25 or less. We also find that the maximum observed depth is less than 100. This suggests that a small amount of state can be allocated on-chip in the accelerator to handle the vast majority of message data, while trapping or spilling to DRAM is acceptable to handle less common cases.

3.9 Key insights for accelerator design

To conclude this section, we outline the key insights from our profiling study that impact the design of a protobuf accelerator:

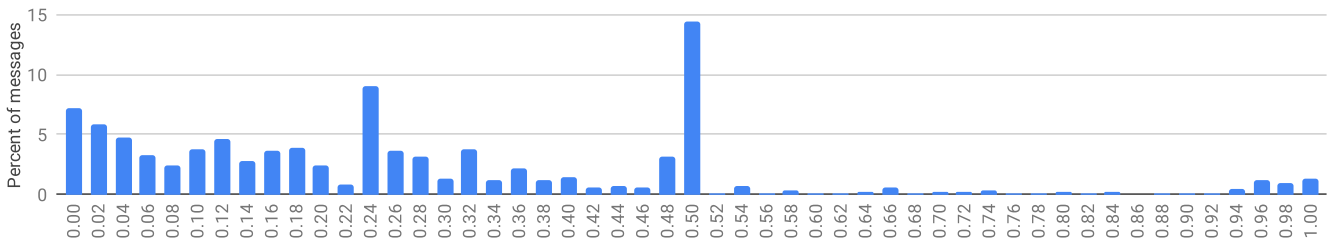


Figure 7: Field number usage density distribution for all message types, weighted by # of observed msgs. of each type.

• A hardware accelerator for protobuf serialization/deserialization could eliminate up to 3.45% of fleet-wide cycles at Google, a significant savings at scale (Section 3.2).
• Usage of serialization framework APIs and formats tends to be stable over time, making hardware acceleration viable (Section 3.3).
• A protobuf accelerator is most amenable to being placed near the CPU core. A common proposal is to place the accelerator on a PCIe-attached NIC. This is unlikely to be fruitful for several reasons:
– Over 83% of deserialization cycles and over 64% of serialization cycles in Google's fleet are not RPC-related and offloading them over PCIe would introduce significant unnecessary data movement (Section 3.4).
– Accesses into the in-memory protobuf representation performed during serialization and deserialization are ill-suited to being performed over PCIe (due to its high latency [34]). The accesses are commonly small and irregularly strided (e.g. ints, floats) or require multiple chained pointer dereferences (strings/bytes/repeated/sub-messages). This is particularly problematic for deserialization, which must process the serialized input sequentially, field-by-field (Section 3.6.4).
– The in-memory representation is commonly sparsely populated, so an optimization such as bulk-copying an entire in-memory protobuf object over PCIe is too wasteful. In a similar analysis as Section 3.7, we find that over 90% of messages fleet-wide only contain values for less than 52% of their defined fields, on average.
– To make on-NIC acceleration truly worthwhile, a SmartNIC must also handle all encapsulations between protobuf serialization/deserialization and frame egress/ingress.
• Trying to achieve acceleration at individual field-granularity (only accelerating varint processing or `memcpy`) is unlikely to be fruitful—a protobuf accelerator will need to understand complete message structure (e.g. processing fields in parallel during serialization), handle a wide variety of field types efficiently (Section 3.6.4), *and* be able to handle fast `memcpy` (Section 3.6.3).
• To program our accelerator, we will use fixed, per-type schema tables combined with dynamic, per-instance presence-tracking bit fields. This scheme is more memory and CPU efficient than prior work [36] (Section 3.7).
• To handle submessages in our accelerator, we will only need to maintain on-chip sub-message context stacks of depth 25 for most messages (Section 3.8).

4 ACCELERATOR DESIGN

This section details the design and implementation of our protobuf accelerator, consisting of the deserializer and serializer units, as well as the software modifications required to exercise the accelerator within the context of our complete accelerated RISC-V SoC design.

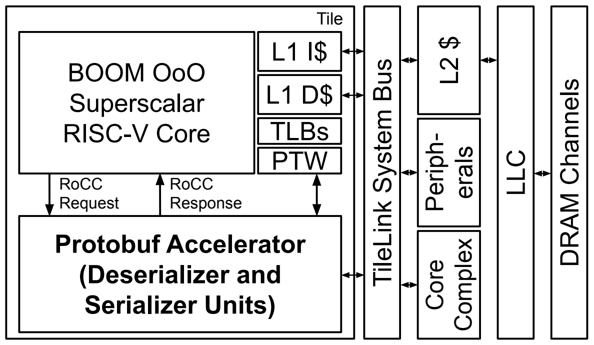


Figure 8: Top-level block diagram of our RISC-V SoC with an OoO superscalar core and protobuf accelerator.

4.1 System overview

The protobuf accelerator is implemented in Chisel RTL [18] and incorporated into the Chipyard RISC-V SoC generator ecosystem [15]. Figure 8 shows the overall architecture of the accelerated SoC. We configure the SoC to use BOOM, an OoO superscalar RISC-V core with performance comparable to ARM A72-like cores [43].

The accelerator receives commands directly from the BOOM application core in the SoC via the RoCC interface [3, 16], which allows the CPU to directly dispatch custom RISC-V instructions in its instruction stream to the accelerator with low latency (ones-of-cycles). These *RoCC instructions* [16] can supply two 64-bit register values from the core to the accelerator. The accelerator accesses the same unified main memory space as the CPU using the coherent 128 bit-wide TileLink system bus [25]. Accesses to main memory made by accelerator components go through the *memory interface wrappers* shown in Figures 9 and 10. These maintain TLBs and interact with the page-table walker (PTW) to perform translation and thus allow the accelerator to use virtual addresses. These also manage tracking OoO responses from the system bus and support a configurable number of outstanding requests, depending on memory system characteristics and resource constraints. Lastly, as shown in Figure 8, all memory accesses made by the accelerator go through the L2 and LLC, which are shared with the application core. Putting these pieces together, offload overhead is minimal: apart from the custom instructions that perform a serialization or deserialization, only a `fence` instruction is required between the user program operating on a protobuf and the accelerator operating on a protobuf.

4.2 Software changes to the protobuf library

We modify the protoc compiler to automatically generate *Accelerator Descriptor Tables (ADTs)*, which encode the layout of a protobuf *message type* in application memory and information about its fields. There is one ADT per-*message-type*, rather than per-message-*instance*, and ADTs are populated when the program is loaded, avoiding adding code to the critical path of setting or clearing message fields in user code. When the serialization or deserialization of a message is dispatched to the accelerator, the message's type's ADT is also passed to the accelerator.

Each ADT contains three regions. The 64B header region contains layout information at the message-level, consisting of: (1) a pointer to a default instance (or vptr value) of the message type, (2) the size of C++ objects of the message type, (3) an offset into message objects for an array of field-presence bit fields (hasbits), and (4) the min and max field number defined in the message. The second ADT region consists of 128-bit wide entries that represent each field in the message type, indexed by field number. Each entry consists of the following details for a field: (1) the field's C++ type and whether the field is repeated, (2) the offset where the field begins in the in-memory C++ representation of the message, and (3) for sub-message fields, a pointer to the sub-message type's ADT. The final ADT region is the is_submessage bit field, an array of bits that indicates if a field is a sub-message. This is used to reduce complexity in the serializer, since it can know when it needs to switch contexts into a sub-message without waiting for a full ADT entry read.

In addition to ADT information, the serialization unit in the accelerator must also know which fields in a given C++ protobuf message are actually populated. The protobuf library tracks this information using the private hasbits member of each C++ protobuf message object. Each bit in the hasbits bit field represents the "presence" of a particular field. protoc represents hasbits *densely*, but supporting a dense packing in the accelerator would require significant overhead (e.g. a mapping table indexed by field number, introducing an additional 32-bit read per-field). Based on our profiling insights in Section 3.7, we find that the dense packing optimization is not significantly helpful in the common cases seen at scale. Thus, to improve accelerator efficiency, we make a different hardware/software co-design trade-off for the accelerator context; we modify the representation of the hasbits bit field such that the accelerator can directly index into it, based on field number. To save memory in the common case where field numbers are contiguous but start at a large number, we provide the accelerator with the minimum defined field number in a message type, with respect to which it calculates field-number offsets.

4.3 Accelerator memory management

To remove the CPU from the critical path of serialization and deserialization, the accelerator will need to manage a memory region in which it allocates and populates deserialized C++ message objects and serialized message outputs. Similar to how an arena is constructed in advance when using arena allocation for software-only protobuf processing (Section 2.3), the application program pre-allocates arena memory regions for the accelerator and passes their pointers to the accelerator via two custom RoCC setup instructions ({ser,deser}_assign_arena). In the rest of this paper, we will refer to standard upstream protobuf arenas (i.e., those from Section 2.3) as *software arenas* and arenas given to the accelerator as *accelerator arenas*.

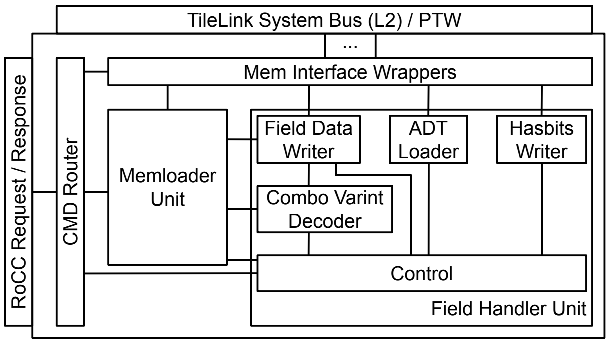


Figure 9: Deserializer unit top-level block diagram.

4.4 Deserializer unit

The deserializer unit is responsible for receiving a serialized protobuf (as a pointer to a sequence of bytes) and decoding it to populate a corresponding C++ object of that message's type. Figure 9 shows the block-level design of the deserializer unit.

To maintain compatibility with standard protobuf software APIs, we expect that the top-level C++ protobuf message object is allocated by the user code (e.g. in the software arena). Any internal objects (sub-messages, strings, and repeated fields) are allocated by the accelerator in the accelerator arena.

4.4.1 Dispatching a deserialization from the CPU. To begin deserialization of a message, the CPU issues two custom instructions through the RoCC interface. The first instruction, deser_info, supplies a pointer to the ADT of the message type being deserialized and a pointer to the top-level destination message object for the accelerator to populate. The second instruction, do_proto_deser, supplies a pointer to the serialized input buffer, the smallest defined field number in the message type, and the length of the input buffer, and kicks off deserialization in the accelerator. Once these instructions are issued, the CPU can perform other work, issue more deser_info and do_proto_deser pairs, or issue a block_for_deser_completion instruction, which is committed after all in-flight deserializations are completed. This is a flexible middle ground that allows for batching deserializations, without requiring SW to unnecessarily poll for completion.

4.4.2 Memloader unit. Once a do_proto_deser instruction is dispatched to the accelerator, the accelerator begins loading serialized buffer contents from memory using the memloader unit. The memloader exposes a decoupled streaming interface to the rest of the pipeline that allows the consumer to accept a consumer-dictated amount of data per-cycle, up to 16B. A full 16 bytes of buffered data are always exposed on this interface, since the number of bytes the consumer will wish to consume is data-dependent.

4.4.3 Field-handler unit.

The field-handler unit implements the core parsing logic required to convert the serialized buffer contents into an in-memory C++ object for the user program to consume. The field handler control is implemented as a state machine that, in a loop, parses a field's key (the `parseKey` state), blocks for detailed type information from the ADT entry for the field (the `typeInfo` state), and then moves into a set of states that handle parsing and writing the field's value based on its detailed type information.

4.4.4 Field-handler unit: `parseKey` state.

Each key is encoded as a varint, which can be up to 10B wide. The field-handler unit contains a combinational varint decoder, which can directly peek at the next 10B of the serialized buffer via the memloader's variable-width consumer interface. The varint parser emits the decoded key (as a 64-bit-wide `uint`) and the encoded length N, so the memloader can discard the N-byte key at the end of the cycle. As described in Section 2, the key consists of two components, the field type and the field number. At the end of the `parseKey` cycle, the field handler dispatches a request to the ADT loader containing the ADT base address for this message type and the field number of the field. The field handler also dispatches a request to the `hasbits` writer, which will set the appropriate bit in the C++ object's `hasbits` bit field to indicate that the field is present in the message.

4.4.5 Field-handler unit: `typeInfo` state.

After the `parseKey` state, the accelerator moves to the `typeInfo` state. This state serves to block on the response from the ADT loader in order to obtain detailed type information. Once the response is received, the logic in this state dispatches to one of four state classes: final write states for scalar fields, string allocation and copy states, repeated-field handling states, or sub-message handling states.

4.4.6 Field-handler unit: final write states for scalar fields.

This set of states handles writes for scalar field types: the varint, 64-bit, and 32-bit protobuf wire types. At the end of this stage, the decoded field data is written into memory. The write address is available from the ADT entry previously received in the `typeInfo` state. The decoded value and size depend on the detailed type being handled, which is known from the loaded ADT entry.

To handle the varint wire type, the same combinational varint parser from the `parseKey` state generates a fixed-width value and supplies the number of bytes consumed back to the memloader consumer interface. The ADT entry distinguishes whether the output type is 32-bits or 64-bits wide and signed or unsigned. For signed varints, the decoded value is passed through an additional combinational zig-zag [6] decoding unit.

4.4.7 Field-handler unit: string allocation and copy states.

String and byte fields and the other field types we will discuss in the remainder of this section introduce a new wrinkle into the deserialization process—instead of relying on user code to have allocated destination memory, the accelerator must handle memory allocation in the accelerator arena assigned to it by the user program.

Our accelerator constructs string objects that are compatible with modern versions of `libstdc++`, which allows user code to directly operate on strings in the deserialized protobuf message as if it were deserialized by the software protobuf library. The accelerator first decodes and consumes the varint-encoded string length. It then constructs the string object and depending on the length, a separate array for the string contents (i.e. the common small string optimization). A pointer to the newly allocated string object is written into the offset in the C++ message object that is obtained from reading the field's ADT entry. Next, the accelerator consumes the string contents from the memloader and writes them into the allocated buffer in memory.

4.4.8 Field-handler unit: repeated-field handling states.

Our accelerator also handles packed and unpacked repeated fields. Packed repeated fields are handled in a similar vein as strings, since they are also represented as length-delimited values. Unpacked values are handled by creating a tagged open-allocation region when the first element in an unpacked repeated field is seen. As more key-value pairs with the same tag are received in the serialized representation, they are copied into the open allocation region. When the accelerator encounters either the end of the current message or a different unpacked repeated field, it closes-out the open allocation region and writes out a final length in *elements* into the repeated-field object in application memory.

4.4.9 Field-handler unit: sub-message handling states.

As described in Section 2.1, protobuf messages can contain sub-messages. So far, the accelerator has relied on several pieces of information that are supplied by the CPU via RoCC instructions to perform deserialization: the ADT pointer for the top-level message's type, a pointer to the user-allocated C++ object in which the deserialized top-level message should be written, the smallest defined field number in the message type, and the length of the serialized top-level message input in bytes. Going forward, we will refer to these elements as *message-level metadata*.

The deserialization process for sub-messages requires consuming the serialized sub-message content in a depth-first manner, which means we must preserve message-level metadata for each message on the path between the current sub-message and the top-level message. Given the depth-first parsing order, we maintain a hardware stack to track message-level metadata during deserialization. The accelerator always uses the message-level metadata at the top of the metadata stack, allowing reuse of the entire pipeline for sub-message decoding.

Putting these pieces together, the sub-message parsing state prepares the accelerator to consume the serialized sub-message output by modifying the stack entries and by performing memory allocation. In this state, the accelerator first decodes the serialized sub-message field's header, which contains the varint-encoded length of the serialized message in bytes. As with other fields, the ADT entry for the field has already been fetched and contains a pointer to the ADT of the *sub-message's* type. Using this pointer, the accelerator fetches metadata from the aforementioned header region of the ADT for the sub-message type, which contains a pointer to a default instance (or `vptr`) of the type and the size of the type. Given this information, the accelerator allocates and initializes a new C++ object for the deserialized sub-message data and writes a pointer to the newly allocated object into the parent object's field pointer. Finally, the accelerator pushes new entries onto the message-level metadata stacks. When the setup is completed, the accelerator returns to the `parseKey` state, where it begins parsing and populating the *sub-message*.

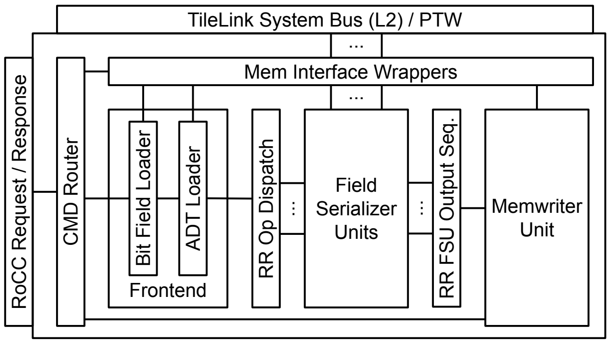


Figure 10: Serializer unit top-level block diagram.

As the sub-message is being processed and input data is consumed, the accelerator updates the total consumed serialized input length. When this length is equal to the top entry in the stack of the serialized message lengths, the sub-message parsing is completed. Popping an entry from each stack returns the accelerator to parsing the parent message.

4.5 Serializer unit

The protobuf accelerator's serializer unit converts a C++ protobuf object populated by a user application into a serialized sequence of bytes. Figure 10 shows the block-level design of the serializer unit.

4.5.1 Field serialization order and serializer memory management. One counter-intuitive but critical note about field serialization order is that the accelerator iterates through fields in *reverse* field number order and writes the serialized output from high-to-low addresses. This produces byte-wise identical output as a software serializer serializing in increasing field number order and writing output from low-to-high addresses, but drastically simplifies the process of populating the length of sub-messages (which appear before the fields in a sub-message). The accelerator arena internally contains two memory regions for serialization: (1) a buffer in which to allocate and write serialized output data and (2) a buffer in which to store pointers to the start of each serialized output in (1).

4.5.2 Dispatching a serialization from the CPU. To dispatch a serialization operation, like before, the user program issues two RoCC instructions. The `ser_info` instruction supplies the offset of the `hasbits` field in the C++ protobuf message object to serialize and the largest and smallest *defined* field numbers for the message type. The `do_proto_ser` instruction supplies a pointer to the top-level ADT of the protobuf message to serialize and a pointer to the C++ representation of the protobuf message to serialize and kicks off a serialization. Like deserialization, the CPU can perform other work, issue more `ser_info` and `do_proto_deser` pairs, or issue a `block_for_ser_completion` instruction, which is committed after all in-flight serializations are completed. After completion, the user program can call a function to get a pointer to the *N*th serialized output (and its length) from the arena.

4.5.3 Frontend. When the accelerator receives the RoCC instructions to initiate a serialization, the accelerator frontend uses the supplied register values to initialize a set of stacks (for sub-message support) that maintain context information for the message being serialized.

Next, the accelerator frontend loads the `is_submessage` and `hasbits` bit fields from memory in parallel, iterates through the fields bit-by-bit, and issues an ADT load request whenever a field is present. For non-sub-message fields, the frontend simply loads ADT information and issues a handle-field-op to the remainder of the pipeline. If a present field is a sub-message, the frontend first updates the current message's context information in the stack. Then, the frontend loads the ADT entry for the sub-message field and the sub-message pointer itself. This information is then pushed onto the context stacks. The handle-field-ops issued to the rest of the pipeline contain a depth field, which allows the memwriter unit to determine when a new sub-message has started. Once these housekeeping steps are completed, the frontend then resumes regular field handling as described previously. After the frontend handles the message's smallest defined field number, it issues a special handle-field-op with field number zero (which the protobuf specification prevents from being used for a user-defined field) to indicate to the remainder of the pipeline that the (sub-)message has been completed. When the end of a (sub-)message is reached, the frontend pops from the context stacks and continues with the parent message (or signals top-level message completion).

4.5.4 Field serializer units. Next, the individual handle-field-ops from the frontend are dispatched round-robin to a set of field serializer units, which produce serialized key, value pairs for individual fields. They load the C++ representation of the field data to serialize from memory, encode it if necessary (e.g. encoding integers as varints), and then make the serialized field data available to their output ports in chunks of parameterizable width. The field serializers also construct and emit the key for each non-sub-message field that is part of the serialized output. Due to space constraints, we do not detail how each individual field type is handled. However, the process of serializing values of each field type is effectively the reverse of deserializing a field of the corresponding type (without needing to perform allocation and C++ object construction), which is discussed in depth in Sections 4.4.6 to 4.4.9.

4.5.5 Memwriter unit. The next stage of the pipeline consumes serialized field data from the parallel set of field serializer units in round-robin fashion and sequences the output into one output stream to feed to the memwriter unit, which writes data to memory. The memwriter also handles the aforementioned special handle-field-ops that indicate the beginning and end of (sub-)messages. The memwriter maintains a stack of the lengths of the messages currently being handled and pushes and pops from the stack as the handle-field-ops with a new depth or with field number zero are received. When an op with field number zero is received (which signals end-of-message), the memwriter injects the sub-message's key, which includes the sub-message's serialized length. The need to inject this key affirms why the output buffer is populated from high-to-low address—we must see all serialized sub-message fields before we know the length of the entire serialized sub-message. When an end-of-message op is received for a top-level-message, the memwriter also writes the current output pointer (the address of the front of the completed serialized message) into the next slot in the buffer of serialized message pointers.

5 EVALUATION

We evaluate our complete accelerated system implemented in RTL using two sets of benchmarks: (1) microbenchmarks that exercise a variety of protobuf features/types and (2) HyperProtoBench, a benchmark suite representative of key serialization framework users at scale. To enable running these benchmarks directly on our RTL design, we run FPGA-accelerated simulations of the design using FireSim [29], which provides high-performance, deterministic, and cycle-exact[5] modeling of the design, while cycle-accurately modeling I/O, including DRAM [20].

For comparison, each benchmark is run on three systems: the baseline single-core BOOM-based[6] RISC-V system modeled at 2 GHz core frequency ("riscv-boom"), the same RISC-V system with our accelerator attached ("riscv-boom-accel"), also modeled at 2 GHz core and accelerator frequency (based on the critical path results in Section 5.3), and one core (2 HT) of a Xeon E5-2686 v4-based server ("Xeon"), running at 2.3 GHz base/2.7 GHz turbo.

5.1 Microbenchmarks

To understand accelerator performance on the various field types supported by protobufs, we developed a set of microbenchmarks that test the performance-similar field types shown in Table 1. We also created μbenchmarks to evaluate performance on messages containing sub-messages and repeated fields. Where appropriate (e.g. varints and strings), we also break-down benchmarks by field size. Each μbenchmark tests either serialization or deserialization of messages containing a fixed number of fields of a particular protobuf field type. For varints, doubles, floats, and their repeated equivalents, we set this to five fields per message, so that the middle-sized non-repeated varint's μbenchmark message falls roughly at the median of message sizes shown in Figure 3. All other μbenchmarks use one field per-message. Each benchmark performs a timed batch of deserializations and serializations, operating on a pre-populated set of serialized messages or C++ message objects respectively, and reports throughput by dividing the total amount of *serialized* message data consumed/produced by the time to process the batch.

5.1.1 Deserialization. Figure 11a shows the results of running deserialization μbenchmarks for field types that do not require memory allocation in the accelerator. To some degree, all examined systems exhibit the behavior that deserialization throughput of varints increases with the size of the varint field. This is due to a variety of factors, including underutilization of memory bandwidth with small loads, fixed-overhead of handling a field (e.g. key handling), and in the case of the accelerator, single-cycle decoding of all varints. Summarizing these results, we find that our accelerated system performs on average 7.0× faster than the BOOM-based system and 2.6× faster than the Xeon.

Figure 11c shows the results of running deserialization microbenchmarks for field types that require the accelerator to perform memory allocation, including repeated fields, strings, and sub-messages. In this figure we also see performance improvements across the board. A key reason for improved performance in these benchmarks is the accelerator's ability to directly allocate memory without requiring CPU intervention. Also, as mentioned in Section 3, the long-string deserialization case essentially becomes a `memcpy`, which the accelerator handles well. Summarizing these results, we find that the accelerated system performs on average 14.2× faster than the BOOM-based system and 6.9× faster than the Xeon-based system.

5.1.2 Serialization. Figure 11b shows the results of running serialization μbenchmarks for field types that are "inline" in C++ message objects. In practice, this is the exact distinction between non-allocated and allocated field types discussed in the deserializer results, however we do not re-use this terminology for clarity. While other platforms show a less consistent increase in throughput based on varint size, the accelerated system shows increased performance as varint size increases. This is similarly due to the improved bandwidth utilization due to larger loads as well as the accelerator's ability to encode fixed-width C++ integer formats into a varint in a single-cycle. We also note that due to this fact, floats and doubles perform similarly to equivalently sized varint fields. Summarizing these results, we find that the accelerated system performs on average 15.5× faster than the BOOM-based system and 4.5× faster than the Xeon.

Figure 11d shows the results of running serialization μbenchmarks for field types that are not "inline" in the top-level C++ message object. Similarly to deserialization, one notable result is the very-long and long sizes of string fields, which both essentially become `memcpy` operations. The accelerator again performs well here, but it is interesting to note that the Xeon also performs extremely well on the very-long-string benchmark, notably better than the deserialization case. Summarizing these results, we find that the accelerated system performs on average 10.1× faster than the BOOM-based system and 2.8× faster than the Xeon.

5.1.3 Overall microbenchmark results. To get a sense of the overall performance improvement our accelerator achieves across a variety of field types, we take the geometric mean of the results reported for the four classes of μbenchmark shown above, for each of the two hosts we compare against. We find that on average, the accelerated system performs 11.2× better than the BOOM-based system and 3.8× better than the Xeon-based system.

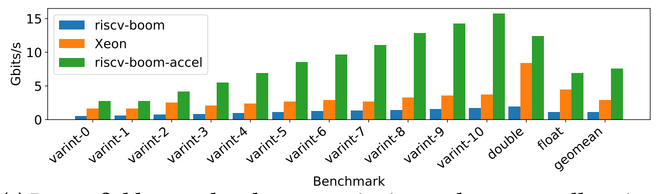
5.2 HyperProtoBench: Open-source Google fleet-representative protobuf benchmarks

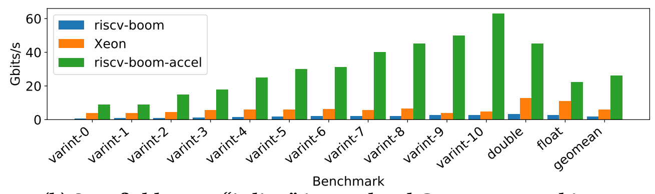
To gain a better understanding of how our design behaves at scale and to enable more productive research in serialization frameworks by providing insight on how these frameworks are used in a hyperscale context, we have open-sourced *HyperProtoBench*, a collection of benchmarks that represent a significant portion of fleet-wide protobuf deserialization and serialization cycles at Google.

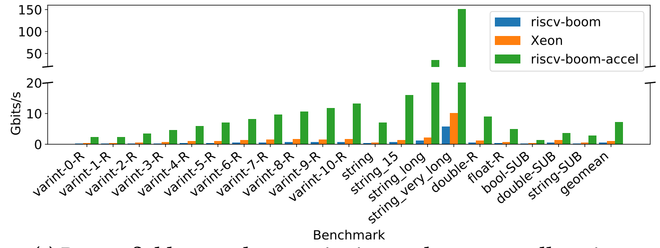
To construct these benchmarks, we collect samples from Google's live production fleet that describe the "shape" of protobuf messages used, per service, using the same mechanisms as described in Section 3. This shape data includes information about which messages are being serialized/deserialized, which fields are set in those messages, the sizes and types of those fields, and the message hierarchy.

[5]All components of the RISC-V SoC written in RTL, including our accel. design, are modeled bit-by-bit and cycle-by-cycle exactly as they would perform in silicon taped-out using the same RTL.

[6]In particular, we use a high-end configuration of SonicBOOM, which performs comparably on IPC with ARM Cortex A72-like cores when running SPEC2017 and achieves higher CoreMarks/MHz than A72-like cores running CoreMark [43].


(a) Deser., field types that do not require in-accel. memory allocation.


(b) Ser., field types "inline" in top-level C++ message objects.


(c) Deser., field types that require in-accel. memory allocation.

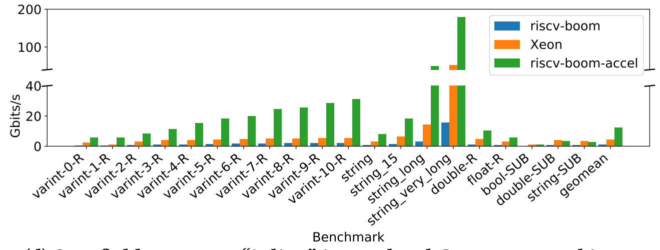

(d) Ser., field types not "inline" in top-level C++ message objects.

Figure 11: Protobuf microbenchmark results.

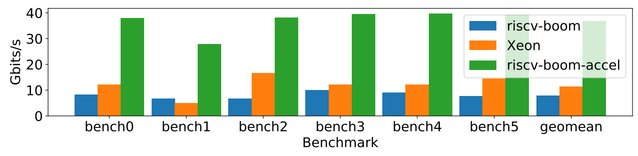


Figure 12: HyperProtoBench deserialization results.

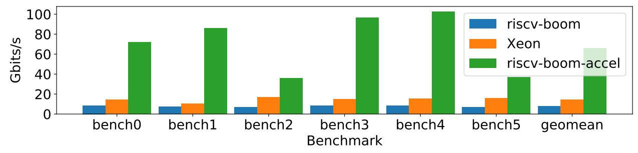


Figure 13: HyperProtoBench serialization results.

Given this input, an internal synthetic protobuf benchmark generator fits a distribution to the input data and then samples from it to produce a benchmark that is representative of a selected production service. For each service, the generator produces a `.proto` file with message definitions representative of those used in the production service and generates a C++ benchmark that constructs, mutates, and serializes/deserializes the protobuf messages appropriately.

To cover as many of the total fleet-wide protobuf serialization and deserialization cycles as possible, we use fleet-wide profiling data to determine the five heaviest users of protobuf deserialization and the five heaviest users of protobuf serialization. In aggregate, these services cover over 13% of fleet-wide deserialization cycles and 18% of fleet-wide serialization cycles. For each of these services, we construct a synthetic benchmark representative of its protobuf usage. This collection of benchmarks comprises *HyperProtoBench*.

Figures 12 and 13 show the results of running the HyperProtoBench deserialization and serialization benchmarks respectively, on the same collection of three systems ("riscv-boom", "riscv-boom-accel", and "Xeon"). We find that our accelerated system achieves on average 6.2× performance improvement compared to our baseline RISC-V SoC with OoO (ARM A72-like) cores and 3.8× performance improvement compared to the Xeon-based system. Extrapolating from the fleet-wide cycles spent in serialization and deserialization, this would result in a savings of over 2.5% of fleet-wide cycles, which at scale translates to hundreds of millions of dollars in savings, across the industry [17, 40].

5.3 ASIC critical path and area

To estimate the ASIC critical path and area results for our accelerator design, we run the design through synthesis for a commercial 22nm process. The deserializer achieves a frequency of 1.95 GHz with a silicon area of 0.133 mm^2. The serializer achieves a frequency of 1.84 GHz with a silicon area of 0.278 mm^2.

6 RELATED WORK

Optimus Prime [36] presents an accelerator for serialization/deserialization. Their design requires adding code to all protobuf setters and clear methods to construct/manage their per-*message-instance* schema tables for accelerator programming, which introduces significant memory/compute overhead. As discussed in-depth in Sections 3.7, 4.2, and 4.5.3, our work instead uses per-*message-type* ADTs (created once at program load-time) for accelerator programming and uses the existing per-*message-instance* `hasbits` bit field in protobufs to track field presence, avoiding the overheads introduced by Optimus Prime. Further in contrast to our work, Optimus Prime focuses on the serialization process and does not cover the deserialization process in-depth, especially the complexity of managing memory and allocating/constructing C++ objects. Also, our work produces an open-source RTL design which is used as the single source of truth for all evaluation purposes; the RTL design is simulated at high performance using FireSim to gather benchmark performance data and evaluated for area/critical path. Finally, Optimus Prime uses three microbenchmarks for protobufs, part of the DeathStarBench benchmark [23] for Apache Thrift, and compares against ARM A57 cores while our work runs protobuf benchmarks derived from key Google services and compares against an (ARM A72-like) OoO RISC-V core and a Xeon server. As discussed earlier, the real-world data suggest several non-intuitive design trade-offs.

Cereal [26] presents an accelerator for serialization/deserialization of Java objects. Cereal requires modifications to the JVM and uses a custom wire format that is amenable to hardware acceleration. In contrast, our work maintains compatibility with the existing protobuf wire format and does not require modifications to the language implementation. Additionally, directly serializing language objects is not practical in a production-WSC running many services, since backwards compatibility becomes challenging. For example, fields are commonly added or removed from a message over time, which would alter object layout, requiring services to update in lock-step. The schema and compiler-based design of protobufs (Section 2) prevents these issues.

Two recent proposals, *Zerializer* [41] and *Breakfast of Champions* [37], suggest adding serialization and deserialization support to PCIe-attached NICs. The former suggests adding (but does not implement) a custom hardware accelerator, while the latter implements a proof-of-concept that re-uses existing NIC scatter-gather functionality to handle serialization and deserialization, but requires a custom zero-copy-friendly serialization API/format. While we place our accelerator near the CPU, it could easily be placed on a PCIe-attached NIC. We discuss placement trade-offs in Section 3.9.

HGum [42] and *Fletcher* [35] generate serialization/deserialization hardware for FPGA-CPU/FPGA communication. Unlike our work, HGum implements a custom serialization format while Fletcher generates hardware pipelines specific to a message schema that must be specified when hardware is constructed.

7 DISCUSSION AND FUTURE WORK

Instruction cache and branch predictor benefits. Reduced I$ pressure and reduced pressure on branch-prediction resources are often overlooked as benefits of protobuf offloading. `protoc` generates large amounts of branch-heavy code to handle serializations and deserializations in software. In some cases, a call to serialize or deserialize can even effectively act like an I$ and branch predictor flush. Offloading serialization and deserialization to an accelerator eliminates both of these pressures. This can save significant CPU cycles, potentially as many as accelerating protobufs itself.

Accelerating other protobuf operations. Figure 2 shows several other protobuf operations that consume a non-trivial number of CPU cycles, including merge, copy, clear, constructors, and destructors. Re-using the hardware building blocks from serialization and deserialization and adding new custom instructions for each, a future version of our accelerator would be able to handle merge, copy, and clear, addressing another 17.1% of fleet-wide C++ protobuf cycles. While we did not claim constructors (6.4% of fleet-wide protobuf cycles) as part of the fleet-wide acceleration opportunity for our accelerator, the accelerator *does* address some of these cycles, by constructing sub-message objects during deserialization. A small change to the protobuf API (software accepting a top-level message pointer from the accelerator) would allow the accelerator to fully offload all deserialization-related constructor cycles. Destructor cost (13.9% of protobuf cycles) can be addressed in software by fully migrating to arenas, which the accelerator already supports.

Future support for `proto3` *and non-C++ host languages.* To our knowledge, the only change needed for `proto3` support in our accelerator is adding support for UTF-8 validation of string fields during deserialization. Adding support for other host languages would require the accelerator to understand the layout of and construct in-memory protobuf message objects for new languages and their standard library components, like strings.

8 CONCLUSION

This work presented an end-to-end study of profiling and accelerating serialization and deserialization, two key datacenter tax components. To understand the trade-offs and opportunities in hardware acceleration for serialization frameworks, we presented the first in-depth study of serialization framework usage at scale by characterizing Protocol Buffers usage across Google's WSC fleet and used this data to construct HyperProtoBench, an open-source benchmark representative of key serialization-framework user services at scale. In doing so, we identified key insights that challenge prevailing assumptions about serialization framework usage.

We used these insights to develop a novel hardware-accelerator for protobufs, implemented in RTL and integrated into a RISC-V SoC. We have fully open-sourced our RTL, which, to the best of our knowledge, is the only such implementation currently available to the community.

We also presented a first-of-its-kind, end-to-end evaluation of our entire RTL-based system running hyperscale-derived benchmarks and microbenchmarks. We booted Linux on the system using FireSim to run these benchmarks and pushed the design through a commercial 22nm process to obtain area and frequency metrics. We demonstrated an *average* 6.2× to 11.2× performance improvement (sometimes up to 15.5×) vs. our baseline RISC-V SoC with BOOM OoO (ARM A72-like) cores and despite the RISC-V SoC's weaker uncore/supporting components, an *average* 3.8× improvement (sometimes up to 6.9×) vs. a Xeon-based server.

In addition to advancing the state of the art in serialization framework acceleration, this work is the first to demonstrate the power of combining a data-driven hardware-software co-design methodology based on large-scale profiling with the promise of agile, open hardware development methodologies. In this vein, our entire evaluation flow (RTL, benchmarks, including hyperscale-derived benchmarks, and supporting software and simulation infrastructure) has been open-sourced for the benefit of the research community and our results have been reproduced by external artifact evaluators.

ACKNOWLEDGMENTS

This work builds on profiling infrastructure work done by several engineering teams at Google (e.g., GWP, protobufz, and protodb) and we would like to thank our current and former colleagues in those teams, including Darryl Gove, Martijn Vels, and Chris Cummins. We would also like to thank Liqun Cheng, Aamer Mahmood, Tipp Moseley, Deepti Raghavan, Don Stark, and the anonymous reviewers and artifact evaluators for their paper feedback. We also thank Shane Knapp for helping to provide external evaluation platform access for artifact evaluation.

The information, data, or work presented herein was funded in part by ADEPT and RISE Lab industrial sponsors and affiliates and by NSF CCRI ENS Chipyard Award #2016662. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact appendix describes how to reproduce the protobuf accelerator evaluation results in Section 5 of this paper. As in Section 5, we will use FireSim FPGA-accelerated simulations to cycle-exactly simulate the entire RISC-V SoC containing the protobuf accelerator. We will boot Linux on this system and run both microbenchmarks and HyperProtoBench to collect accelerator performance metrics.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** AWS FPGA Developer AMI 1.6.1.
- **Hardware:** AWS EC2 instances: 1× `c5.9xlarge`, 1× `f1.16xlarge`, 1× `m4.large`.
- **Metrics:** Protobuf serialization/deserialization throughput (Gbits/s).
- **Output:** Serialization/deserialization performance plots.
- **Experiments:** FireSim simulations of protobuf accelerator incorporated into a RISC-V SoC, running serialization/deserialization microbenchmarks and HyperProtoBench.
- **How much disk space is required?:** 200 GB (on EC2 instance).
- **How much time is needed to prepare workflow?:** 2 hours (scripted installation).
- **How much time is needed to complete experiments?:** 3.5 hours (scripted run).
- **Publicly available:** Yes.
- **Code licenses:** Several, see download.
- **Archived:** https://doi.org/10.5281/zenodo.5433464, https://doi.org/10.5281/zenodo.5433448, https://doi.org/10.5281/zenodo.5433434, https://doi.org/10.5281/zenodo.5433410, and https://doi.org/10.5281/zenodo.5433364.

A.3 Description

A.3.1 How to access. The artifact consists of five git repositories preserved on Zenodo:

(1) **`firesim-protoacc-ae`**: Top-level FireSim simulation environment. (https://doi.org/10.5281/zenodo.5433464)
(2) **`chipyard-protoacc-ae`**: Chipyard RISC-V SoC generation environment. (https://doi.org/10.5281/zenodo.5433448)
(3) **`protoacc-ae`**: Protobuf accelerator design, software, and scripts. (https://doi.org/10.5281/zenodo.5433434)
(4) **`protobuf-library-for-accel-ae`**: Fork of protobuf library modified for accelerator support. (https://doi.org/10.5281/zenodo.5433410)
(5) **`HyperProtoBench`**: Protobuf serialization/deserialization benchmarks representative of key serialization-framework user services at scale, open-sourced for this paper. This is a fork of our upstream release (https://github.com/google/HyperProtoBench) customized for accelerator benchmarking. (https://doi.org/10.5281/zenodo.5433364)

Users need not download the latter four repositories manually—they will be obtained automatically from Zenodo when the first repository is set up in the next section.

A.3.2 Hardware dependencies. One AWS EC2 `c5.9xlarge` instance (also referred to as the "manager" instance), one `f1.16xlarge` instance, and one `m4.large` instance are required. The latter two will be launched automatically by FireSim's manager.

To optionally run FPGA builds (see Section A.7.2), two additional `z1d.6xlarges` are required, however we provide pre-built FPGA images to avoid the long latency (~10 hours) of this process.

A.3.3 Software dependencies. Installing `mosh` (https://mosh.org/) on your local machine is highly recommended for reliable access to EC2 instances. All other requirements are automatically installed by scripts in the following sections.

A.4 Installation

First, follow the instructions on the FireSim website[7] to create a manager instance on EC2. You must complete up to and including "Section 2.3.1.2: Key Setup, Part 2", with the following changes in "Section 2.3.1":

(1) When instructed to launch a `c5.4xlarge` instance, choose a `c5.9xlarge` instead.
(2) When entering the root EBS volume size, use 1000GB rather than 300GB.

Once you have completed up to and including "Section 2.3.1.2" in the FireSim docs, you should have a manager instance set up, with an IP address and key. Use either `ssh` or `mosh` to login to the instance.

From this point forward, all commands should be run on the manager instance.

Begin by downloading the top-level repository from Zenodo, like so:

```
$ cd ~/
# Enter as a single line:
$ wget -O firesim-protoacc-ae.zip https://zenodo.org/
       record/5433465/files/firesim-protoacc-ae.zip
$ unzip firesim-protoacc-ae.zip
```

Next, run the following, which will initialize all dependencies and run basic FireSim and Chipyard setup steps (RISC-V toolchain installation, matching host toolchain installation, etc.):

```
$ cd firesim-protoacc-ae
$ ./scripts/first-clone-setup-fast.sh
```

This step should take around 1.5 hours. Upon successful completion, it will print:

```
first-clone-setup-fast.sh complete.
```

Once this is complete, run:

```
$ source sourceme-f1-manager.sh
```

Sourcing this file will have set up your environment to run the protobuf accelerator simulations.

Finally, in the FireSim docs, follow the steps in (only) "Section 2.3.3: Completing Setup Using the Manager"[8]. Once you have completed this, your manager instance is fully set up to run protobuf accelerator simulations.

[7] https://docs.fires.im/en/1.12.0/Initial-Setup/index.html
[8] https://docs.fires.im/en/1.12.0/Initial-Setup/Setting-up-your-Manager-Instance.html#completing-setup-using-the-manager

A.5 Experiment workflow

Now that our environment is set up, we will run the full artifact evaluation script, which does the following:

(1) On the manager instance, build the FireSim host-side drivers required to drive the FPGA simulation.
(2) On the manager instance, build our modified protobuf library, cross-compile all benchmarks we will run, and construct a Buildroot-based Linux distribution containing these benchmarks, which will be booted on the accelerated system.
(3) For isolated Xeon runs, launch an `m4.large`, run benchmarks on it and collect results, and terminate the `m4.large`.
(4) Run FireSim simulations, repeat the following for the three classes of benchmarks (accelerated serialization, accelerated deserialization, and plain BOOM):
  (a) Launch an `f1.16xlarge` instance.
  (b) Copy all simulation infrastructure to the F1 instance.
  (c) Run the set of benchmarks on 6 or 7 simulated systems in parallel (one `f1.16xlarge` has 8 FPGAs).
  (d) Copy results back to the manager instance.
  (e) Terminate the `f1.16xlarge` instance.
(5) On the manager instance, re-generate the accelerator performance plots in this paper, with data collected from your runs.

Note that this script will *not* rebuild FPGA images for the system by default, since each build takes around 10 hours. We instead provide pre-built images by default (see `config_hwdb.ini` in `$PROTOACC_FSIM`). If you would like to build your own images, see Section A.7.2, then return here.

Now, let's run the aforementioned full artifact evaluation script:

```
$ cd $PROTOACC_FSIM
$ ./run-ae-full.sh
```

This will take around 3.5 hours. When complete, it will print:

```
run-ae-full.sh complete.
```

The FireSim manager will have automatically terminated any instances it launched during this process, but please confirm in your AWS EC2 management console that no instances remain besides the manager.

A.6 Evaluation and expected results

Next, we will step through the plots generated from your run of `run-ae-full.sh` in the previous section.

A.6.1 Microbenchmark results. Results from your run will be located in the `$UBENCH_RESULTS` directory:

(1) Figure 11a: `nonalloc.pdf`
(2) Figure 11c: `allocd.pdf`
(3) Figure 11b: `nonalloc-serializer.pdf`
(4) Figure 11d: `allocd-serializer.pdf`
(5) Final speedup results: at the end of `process.py.log` and `process-serialize.py.log`

A.6.2 HyperProtoBench results. Results from your run will be located in the `$HYPER_RESULTS` directory:

(1) Figure 12: `hyper-des.pdf`
(2) Figure 13: `hyper-ser.pdf`
(3) Final speedup results for serialization and deserialization: near the end of the `SPEEDUPS` file

Once your evaluation is complete, manually terminate your manager instance in the EC2 management console and confirm that no other instances from the evaluation process are left running.

A.7 Experiment customization

A.7.1 Customizing the design. Since the protobuf accelerator is written in Chisel RTL, incorporated into the Chipyard RISC-V SoC generator ecosystem, and modeled at high-performance using FireSim, it can be experimented with in a wide-variety of contexts, including in multi-core systems, attached to in-order processors (instead of the superscalar OoO BOOM used here), and with different memory hierarchy configurations, to name a few. These parameters are too numerous to list here; see the FireSim docs[9], Chipyard docs[10], and tutorial slides[11] for these configuration options.

The protobuf accelerator RTL is located in the `$PROTOACC_SRC` directory and can be customized and improved as necessary.

A.7.2 Rebuilding FPGA images. We provide pre-built FPGA images for the designs in this paper (generated from the included RTL), encoded in the configuration files in the artifact.

Regenerating the supplied FPGA images can also be done by modifying the S3 bucket name in `$PROTOACC_FSIM/config_build.ini` to an unused bucket name (that the manager will create), then running `./buildafi.sh` in the `$PROTOACC_FSIM` directory. This will take around 10 hours, require two `z1d.6xlarge` instances, generate two new AGFIs (i.e., FPGA bitstreams on EC2 F1), and place their `config_hwdb.ini` entry in `$BUILT_HWDB_ENTRIES/[config name]`. To use the new AGFI, replace the existing entry in the `config_hwdb.ini` file in `$PROTOACC_FSIM` (or, for a new config, add it). If generating your own FPGA images, you must also set the correct value for `customruntimeconfig` in the `config_hwdb.ini` entry to obtain correct memory system performance:

```
customruntimeconfig=2GHz-runtime-conf-32MBLLC-qc.conf
```

When an FPGA build completes, the FireSim manager will automatically terminate the instances it launched during the build process, but please confirm in your AWS EC2 management console that no instances remain besides the manager. More details about the FireSim FPGA build process can be found in the FireSim docs[12]. Note that many of the FireSim manager build configuration files are in a non-standard location to simplify scripting for artifact evaluation. Open `buildafi.sh` to see their locations.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

[9]https://docs.fires.im/en/1.10.0/
[10]https://chipyard.readthedocs.io/en/1.3.0/
[11]https://fires.im/isca-2021-tutorial/
[12]https://docs.fires.im/en/1.10.0/Building-a-FireSim-AFI.html

REFERENCES

[1] [n. d.]. Apache Thrift. https://thrift.apache.org/.

[2] [n. d.]. AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[3] [n. d.]. Building Custom RISC-V SoCs in Chipyard. https://fires.im/micro19-slides-pdf/03_building_custom_socs.pdf.

[4] [n. d.]. C++ Arena Allocation Guide | Protocol Buffers | Google Developers. https://developers.google.com/protocol-buffers/docs/reference/arenas.

[5] [n. d.]. Cap'n Proto. https://capnproto.org/.

[6] [n. d.]. Encoding | Protocol Buffers | Google Developers. https://developers.google.com/protocol-buffers/docs/encoding#signed_integers.

[7] [n. d.]. Extensible Markup Language (XML). https://www.w3.org/XML/.

[8] [n. d.]. Flatbuffers. https://google.github.io/flatbuffers/.

[9] [n. d.]. FlexBuffers. https://google.github.io/flatbuffers/flexbuffers.html.

[10] [n. d.]. Introducing JSON. https://www.json.org/json-en.html.

[11] [n. d.]. Protocol Buffers | Google Developers. https://developers.google.com/protocol-buffers.

[12] [n. d.]. Updating a Message Type | Language Guide | Protocol Buffers | Google Developers. https://developers.google.com/protocol-buffers/docs/proto#updating.

[13] [n. d.]. YAML: YAML Ain't Markup Language. https://yaml.org/.

[14] Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea, and Adam Stubblefield. 2020. *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems.* O'Reilly Media.

[15] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[16] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[17] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19).* Association for Computing Machinery, New York, NY, USA, 462–473. https://doi.org/10.1145/3307650.3322234

[18] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012.* 1212–1221. https://doi.org/10.1145/2228360.2228584

[19] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189. https://doi.org/10.2200/S00874ED3V01Y201809CAC046

[20] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19).* Association for Computing Machinery, New York, NY, USA, 330–339. https://doi.org/10.1145/3289602.3293894

[21] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[22] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* 1–14. https://doi.org/10.1109/ISCA.2018.00012

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19).* Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[24] John Hennessy and David Patterson. 2018. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* 27–29. https://doi.org/10.1109/ISCA.2018.00011

[25] SiFive Inc. 2019. SiFive TileLink Specification. https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf.

[26] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. 2020. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* 322–334. https://doi.org/10.1109/ISCA45697.2020.00036

[27] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA).* 1–12. https://doi.org/10.1145/3079856.3080246

[28] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15).* Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[29] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* 29–42. https://doi.org/10.1109/ISCA.2018.00014

[30] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, Rimas Avizienis, Brian Richards, Jonathan Bachrach, David Patterson, Elad Alon, Bora Nikolic, and Krste Asanovic. 2016. An Agile Approach to Building RISC-V Microprocessors. *IEEE Micro* 36, 2 (2016), 8–20. https://doi.org/10.1109/MM.2016.11

[31] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. 2016. ASIC Clouds: Specializing the Datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).* 178–190. https://doi.org/10.1109/ISCA.2016.25

[32] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[33] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[34] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18).* Association for Computing Machinery, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[35] Johan Peltenburg, Jeroen van Straten, Lars Wijtemans, Lars van Leeuwen, Zaid Al-Ars, and Peter Hofstee. [n. d.]. Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow. In *29th International Conference on Field Programmable Logic and Applications.* https://doi.org/10.1109/FPL.2019.00051

[36] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1203–1216. https://doi.org/10.1145/3373376.3378501

[37] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 199–205. https://doi.org/10.1145/3458336.3465287

[38] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. 2021. Warehouse-Scale Video Acceleration: Co-Design and Deployment in the Wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 600–615. https://doi.org/10.1145/3445814.3446723

[39] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (2010), 65–79. https://doi.org/10.1109/MM.2010.68

[40] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 733–750. https://doi.org/10.1145/3373376.3378450

[41] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: Towards Zero-Copy Serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/3458336.3465283

[42] Sizhuo Zhang, Hari Angepat, and Derek Chiou. 2017. HGum: Messaging framework for hardware accelerators. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–8. https://doi.org/10.1109/RECONFIG.2017.8279799

[43] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).