A Hardware Accelerator for Protocol Buffers

Sagar Karandikar$^{1,2}$, Chris Leary$^2$, Chris Kennelly$^2$, Jerry Zhao$^1$, Dinesh Parimi$^1$, Boživoje Nikolić$^1$, Krste Asanović$^1$, Parthasarathy Ranganathan$^2$

$^1$UC Berkeley, $^2$Google

sagark@eecs.berkeley.edu

What is Protocol Buffers ("protobuf")?

- Google’s serialization framework
- Open-source and also widely used outside Google
  - https://github.com/protocolbuffers/protobuf
- Serialization/Deserialization are foundational operations in Warehouse Scale Computers (WSCs). Two key use cases:

<table>
<thead>
<tr>
<th>01</th>
<th>Inter-service communication via RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Serialize arguments to send RPC request</td>
</tr>
<tr>
<td></td>
<td>• Deserialize return values from received RPC response</td>
</tr>
<tr>
<td></td>
<td>• Potentially high fan-out</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>02</th>
<th>Formatting data for storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Storage APIs commonly expect data as contiguous seq. of bytes</td>
</tr>
<tr>
<td></td>
<td>• Serialize in-mem service/app data to seq. of bytes for storage</td>
</tr>
<tr>
<td></td>
<td>• Deserialize seq. of bytes read from storage to in-mem format usable by services</td>
</tr>
</tbody>
</table>
## Why accelerate protobuf?

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>5%</td>
<td>Of fleet-wide cycles spent in protobuf</td>
<td>Google, 2015 [1]</td>
</tr>
<tr>
<td>6%</td>
<td>Of cycles in key microservices spent in serialization/deserialization</td>
<td>Facebook, 2020 [2]</td>
</tr>
<tr>
<td>9.6%</td>
<td>Of fleet-wide cycles spent in protobuf</td>
<td>Google, now [this work]</td>
</tr>
</tbody>
</table>
An analysis of fleet-wide protobuf usage at Google

Key insights for serialization framework and serialization accelerator design, based on fleet-wide profiling at Google.
Profiling insight #1: Acceleration opportunity

3.45% of fleet-wide cycles could be offloaded by a protobuf ser/des hardware accelerator.

Fleet-wide C++ protobuf cycles by operation. Total pie = 8.4% of fleet-wide cycles.
Profiling insight #2: HW feasibility

proto3 was released in mid-2016, but 96% of protobuf bytes serialized/deserialized remain defined in proto2.

Usage of serialization framework APIs and formats tends to be stable over time, making hardware acceleration viable.

(1) An analysis of fleet-wide protobuf usage at Google
Profiling insight #3: Near-core, not PCIe-attached

A protobuf accelerator is most amenable to being placed near the CPU core.

- **% of cycles not RPC-related**
  - 83% of deserialization cycles and 64% of serialization cycles are not RPC-related.

- **Accesses into in-memory rep. ill-suited to PCIe**
  - Commonly small, irregularly strided, multiple chained pointer derefs. Compounded by deserialization serial processing.

- **In-memory rep. is commonly sparsely populated**
  - 90% of messages fleet-wide only contain values for less than 52% of their defined fields.

- **Handling other encapsulations**
  - A SmartNIC must handle all encapsulations between proto msg. and frame egress/ingress.

A common alternative proposal is to place a protobuf accelerator on a PCIe-attached NIC.

(1) An analysis of fleet-wide protobuf usage at Google
Profiling insight #4: A variety of message shapes

- 56% of messages are 32B or less
  - Accelerator should operate at entire msg. granularity, near-core, without CPU intervention
- Lots of data is in large byte fields
  - Accelerator needs efficient memcpy support
- Ser/des CPU cycles split across many field types
  - Accelerator needs to efficiently handle all field types
- 99.999% of bytes of protobuf data are at sub-message depth 25 or less

(1) An analysis of fleet-wide protobuf usage at Google
An analysis of fleet-wide protobuf usage at Google

Key insights for serialization framework and serialization accelerator design, based on fleet-wide profiling at Google.

An open-source hyperscale protobuf benchmark

HyperProtoBench, an open-source benchmark suite representative of key protobuf users at Google.
github.com/google/HyperProtoBench
HyperProtoBench: Open-source protobuf benchmarks representative of key protobuf-user services at Google

Select the top five fleet-wide serialization and deserialization users. Four services appear in both = six benchmarks.

Collect fleet-wide protobuf message “shape” information for each service.

Construct protobuf message shape distributions based on the per-service profiling data.

Per-selected service, generate a .proto file with message definitions and a C++ benchmark that constructs, mutates, and ser/des messages.

github.com/google/HyperProtoBench
An analysis of fleet-wide protobuf usage at Google

Key insights for serialization framework and serialization accelerator design, based on fleet-wide profiling at Google.

github.com/google/HyperProtoBench

An open-source hyperscale protobuf benchmark

HyperProtoBench, an open-source benchmark suite representative of key protobuf users at Google.

github.com/google/HyperProtoBench

An open-source RTL protobuf accelerator

A novel, open-source protobuf accelerator design aligned with profiling insights, implemented in RTL and integrated into a RISC-V SoC w/BOOM OoO core.

github.com/ucb-bar/protoacc
Generating “programming” for a protobuf accelerator

- **Accelerator Descriptor Tables (ADTs)**
  - Describe the layout of messages in application memory
  - Automatically generated by modified `protoc` compiler
  - Created/populated *once* at application load time
  - One per message-*type*

- **hasbits bit field**
  - HW serializer must know which fields are present (i.e., set) in the in-memory C++ representation
  - Standard protobuf C++ message objects *already track presence via hasbits*
    - A set of bits where the bit corresponding to a field is set if the field is present
  - Modify `protoc` to emit sparse hasbits encoding for efficient accelerator access
  - One per message-*instance*

*In contrast, the closest prior work [3] relies on per message-*instance* “schemas” maintained by code added to all field setters/clear methods, adding significant CPU/memory overhead.*
Profiling insight #5: Protobuf accelerator programming tradeoffs

Our software modifications for accelerator support are more efficient in CPU/memory overhead terms than prior work [3] for over 92% of fleet-wide messages.
Protobuf accelerator: System overview

- Accelerator written in Chisel RTL [4]
- Integrated into Chipyard RISC-V SoC generator [5]
- SonicBOOM OoO RISC-V as application core [6]
  - IPC-comparable on SPEC17 with ARM Cortex A72-like cores
- Accelerator interfaces:
  - Custom RISC-V instructions from core via RoCC
  - Coherent memory access (via L2) using 128-bit TileLink
  - Page-table walker access (accelerator operates on virtual addresses)

(3) An open-source RTL protobuf accelerator
Protobuf accelerator: Deserializer overview

Low-level SW API:
- One RoCC setup instruction to supply accelerator arena (amortized across several messages)
- Two non-blocking RoCC instructions to kick-off deserialization
- One RoCC instruction to block on all in-flight deserializations

Key features/insights:
- Combinational varint decode, sub-message support (on-chip metadata stack sized to 25 based on profiling)
- Automatically allocates/constructs C++ objects for sub-messages, `std::string` for strings (including SSO), repeated field objects in accel. arena
- Also populates `hasbits` required for serialization-side
Protobuf accelerator: Serializer overview

Low-level SW API:
- One RoCC setup instruction to supply accelerator arena (amortized across several messages)
- Two non-blocking RoCC instructions to kick-off serialization
- One RoCC instruction to block on all in-flight serializations

Key features/insights:
- Combinational varint encode, sub-message support (on-chip metadata stack sized to 25 based on profiling)
- Fields are handled in reverse field # order and data is written from high-to-low address: more efficient handling of length-delimited lengths
- Handling of individual fields parallelized across multiple field serializer units

(3) An open-source RTL protobuf accelerator
Key Contributions

1. An open-source RTL protobuf accelerator
   - A novel, open-source protobuf accelerator design aligned with profiling insights, implemented in RTL and integrated into a RISC-V SoC w/BOOM OoO core. [github.com/ucb-bar/protoacc](http://github.com/ucb-bar/protoacc)

2. An open-source hyperscale protobuf benchmark
   - HyperProtoBench, an open-source benchmark suite representative of key protobuf users at Google. [github.com/google/HyperProtoBench](http://github.com/google/HyperProtoBench)

3. An analysis of fleet-wide protobuf usage at Google
   - Key insights for serialization framework and serialization accelerator design, based on fleet-wide profiling at Google.

4. A reproducible, end-to-end evaluation
   - HyperProtoBench and μbmarks running on RTL-impl. of accelerated system, cycle-exactly simulated in FireSim. Up to 6.9x improvement vs. Xeon, 15.5x vs. BOOM. Reproduced by artifact evaluators.
Fully open-sourced, reproduced by artifact evaluators.

**FireSim [7] FPGA-accelerated simulation**
- boot Linux, run bmarks modeled @ 2 GHz

**Microbenchmarks**
- 3.8x average vs. Xeon (up to 6.9x)
- 10.1x average vs. BOOM (up to 15.5x)

**HyperProtoBench**
- See next slide

**RTL implementation**
- protobuf accelerator and RISC-V SoC

**ECAD tools**
- commercial 22nm FinFET process

**Area**
- Deserializer: 0.133 mm$^2$
- Serializer: 0.278 mm$^2$

**Frequency**
- Deserializer: 1.95 GHz
- Serializer: 1.84 GHz

**MICRO-54 Distinguished Artifact Winner!**
HyperProtoBench results

A savings of 2.5% of fleet-wide cycles

Accelerator achieves geomean
6.2x improvement vs. SonicBOOM
3.8x improvement vs. Xeon E5-2686 v4 *

* despite RISC-V SoC’s weaker uncore/supporting components

(4) A reproducible end-to-end evaluation
Conclusion

An open-source RTL protobuf accelerator
A novel, open-source protobuf accelerator design aligned with profiling insights, implemented in RTL and integrated into a RISC-V SoC w/BOOM OoO core.
github.com/ucb-bar/protoacc

An analysis of fleet-wide protobuf usage at Google
Key insights for serialization framework and serialization accelerator design, based on fleet-wide profiling at Google.

A reproducible, end-to-end evaluation
HyperProtoBench and μbmarks running on RTL-impl. of accelerated system, cycle-exactly simulated in FireSim. Up to 6.9x improvement vs. Xeon, 15.5x vs. BOOM. Reproduced by artifact evaluators.

An open-source hyperscale protobuf benchmark
HyperProtoBench, an open-source benchmark suite representative of key protobuf users at Google.
github.com/google/HyperProtoBench
References


A Hardware Accelerator for Protocol Buffers

Sagar Karandikar\textsuperscript{1,2}, Chris Leary\textsuperscript{2}, Chris Kennelly\textsuperscript{2}, Jerry Zhao\textsuperscript{1}, Dinesh Parimi\textsuperscript{1}, Bořivoje Nikolić\textsuperscript{1}, Krste Asanović\textsuperscript{1}, Parthasarathy Ranganathan\textsuperscript{2}

\textsuperscript{1} UC Berkeley, \textsuperscript{2} Google

sagark@eecs.berkeley.edu