Datacenter Tax Cuts: Improving WSC Efficiency Through Protocol Buffer Acceleration

Dinesh Parimi, William Zhao, Jerry Zhao
University of California, Berkeley

Berkeley Architecture Research

Abstract

At least 25% of cycles in a modern warehouse-scale computer are spent on common building blocks termed “the datacenter tax”. These tasks are prime candidates for hardware acceleration, as even modest improvements here can generate immense cost and power savings given the scale of such systems. One of these tasks is serialization and parsing of protocol buffers, an open-source mechanism for representing structured data used extensively in Google’s datacenters for communication and remote procedure calls.

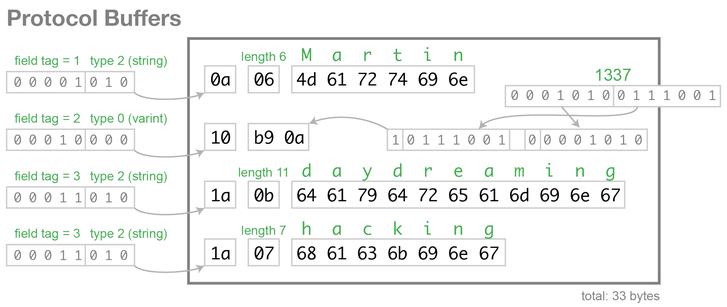
In this work, we isolate and identify key speed bottlenecks in processing protocol buffers, and propose architectural features to address them. We also isolate software- and system-level features that can improve performance, by comparing against alternative structured data buffer formats.

Introduction

As more and more services move away from mobile and desktop devices into the cloud, it has become critical to identify and remedy key bottlenecks limiting datacenter performance. Simultaneously, the end of Moore’s law has compelled computer architects to develop novel accelerators and seek software/hardware codesign to achieve performance gains.

We seek to tackle **protobuf serialization,** a key “datacenter tax” as identified by Google researchers. We primarily study the behavior of the **protobuf serialization algorithm** and compare it to another approach to serialization, **Cap’n Proto.**

Our results identify many of the performance bottlenecks in serializing large objects via protobuf, and we identify opportunities for acceleration that will guide the design of our accelerator.



Methods

The first approach to profiling consisted of a series of microbenchmarks - large protos of a single datatype analyzed using C++ timers and the Linux perf utility to provide insight into relative type performance, execution hotspots, and memory performance. These were run on a single-core 3.0 GHz Intel Xeon Platinum 8000-series processor with 32 GiB of RAM.

The second approach to profiling consisted of comparing serialization performance between 2 data serialization technologies: Protobuf and Cap’n Proto. Protos used in the top starred repositories on github were scraped, ported to Cap’n Proto, and compared. In this evaluation, we looked at serialization time and the resulting data size. The evaluation was performed on VM with a single core Intel i7 6000 series processor and 16GB RAM.

Results

Processing floating point fields during serialization and parsing was significantly faster than integer fields. Integers use a variable length (varint) encoding that compresses near-zero values, but incurs some overhead. Runtime data recorded by perf points to a few throughput bottlenecks: a method to calculate the size of the serialized field represents 10-17% of the serialization time, while the actual serialization method incurs overhead by looping over each byte of the integer. During parsing, the methods VarintParse/VarintParseSlow64 represented 32-47% of parsing time.

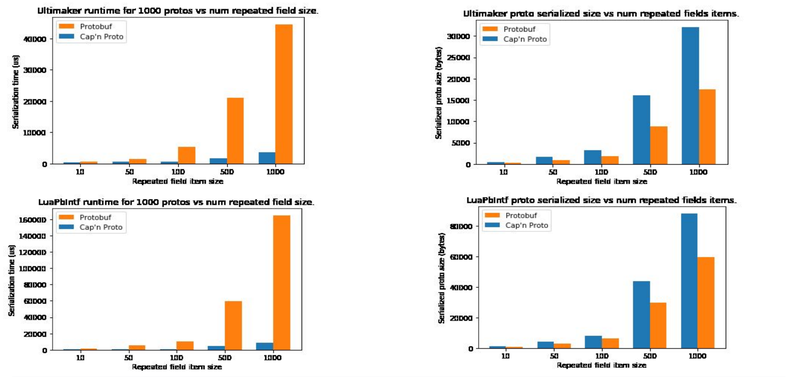
Processing string fields was an order of magnitude slower than floating-point numbers of comparable size. In this case, there were two culprits:

- Serializing/parsing string fields also produced an order of magnitude more memory traffic. This is because strings are heap-allocated, so traversing pointers to the heap adds considerable time cost.
- Protobuf verifies that the byte string is a valid UTF-8 string - an expensive process that represents 14% of cycles in serialization and 20% in parsing.

Table 1. Relative performances of different field

Field type	Int32	Int64	UInt32	UInt64	Float	Double	String (4B)
Avg. time to serialize 1000 fields (us)	43	46	38	44	12	17	133
Avg. time to parse 1000 fields (us)	38	57	36	56	13	18	217
Serialization memory traffic - read (GiB)	2.80	5.40	2.80	5.40	2.68	5.21	30.78
Serialization memory traffic - write (GiB)	3.16	5.59	3.16	5.59	2.66	4.79	3.22
Parsing memory traffic - read (GiB)	5.04	9.42	5.11	9.41	4.56	8.64	33.25
Parsing memory traffic - write (GiB)	2.00	4.03	2.04	4.00	1.96	4.01	25.72

Figure 1. Relative performance between protos among 2 serialization technologies.



Results

The struct in the Ultimaker experiment consists mostly of floats and another internal struct which itself consists of an int and a float. Protobuf can efficiently encode the floats and thus beats Cap’n Proto in the serialization payload size by nearly a factor of 2.
The struct in the LuaPbIntf experiment consists of a mixture of various primitive datatypes and 2 repeated fields of strings and ints. The compression ratio of Protobuf vs Cap’n Proto was significantly better in the Ultimaker experiment (54%) whereas in the LuaPbIntf experiment it was (67%). This was likely due to Protobuf’s inefficient encoding of ints and its inefficient handling of statically sized repeated fields.
We were only able to benchmark 2 experiments due to the fact that Cap’n Proto doesn’t do encoding at all during serialization which made writing protos for Cap’n Proto quite inflexible. Protos which had holes in their numbered fields could not be faithfully ported and benchmarked.

Accelerator Plan

Although we were not able to produce a base accelerator as part of this work due to time and AWS constraints, the results of this work inform the design of the accelerator we will pursue in the coming months.

- BOOM state-of-the-art baseline open-source core
- Multi-level cache hierarchy with prefetchers into L1 and L2
- RoCC instruction based command stream

Deserialization looks like compression, so we will focus on serialization initially.