Datacenter Tax Cuts: Improving WSC Efficiency Through Protocol Buffer Acceleration

Dinesh Parimi
University of California, Berkeley
dineshp@berkeley.edu

William J. Zhao
University of California, Berkeley
william_zhao@berkeley.edu

Jerry Zhao
University of California, Berkeley
jerryz123@berkeley.edu

***Abstract*—According to recent literature, at least 25% of cycles in a modern warehouse-scale computer are spent on common "building blocks" [1]. Referred to by Kanev et al. as a "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 protocol buffer serialization and parsing. Protocol buffers are an open-source mechanism for representing structured data developed by Google, and used extensively in their datacenters for communication and remote procedure calls. While the source code for protobufs is highly optimized, certain tasks - such as the compression/decompression of integer fields and the encoding of variable-length strings - bottleneck the throughput of serializing or parsing a protobuf. These tasks are highly parallelizable and thus prime candidates for hardware acceleration.

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.

***Index Terms*—serialization, warehouse-scale-computing, datacenter architecture, acceleration**

I. Introduction

A recent study of live data-center jobs running on over 20,000 Google servers identified common building blocks with collectively utilize a significant portion of the overall compute time. While datacenter workloads are typically very diverse, these building blocks were identified to be common to a wide range of workloads, indicating that architects should focus on accelerating these workloads to most directly impact datacenter performance.

At the same time, the end of Moore's law has largely limited the steady exponential increases in performance that have defined general-purpose-processors over the past decades. Instead architects must turn to purpose-designed accelerators and software/hardware codesign to seek out performance improvements.

These two phenomena motivate the development of purpose-built accelerators to reduce the burden of the "datacenter tax". We identify serialization as a key component of this tax that is understudied compared to other tasks (like compression, hashing, or kernel overheads).

The most popular framework for object serialization and transport is Protocol buffers, or protobuf [6]. Typically, code will serialize some set of objects stored in memory to a portable format defined by the protobuf compiler (protoc). The serialized objects can be sent to some remote device, which will then deserialize the object into its own memory to execute on. Thus protobuf is widely used to implement remote procedure calls (RPC), since it facilitates the transport of arguments across a network. Protobuf remains the most popular serialization framework due to its maturity, backwards compatibility, and extensibility. Specifically, the encoding scheme enables future changes to the protobuf schema to remain backwards compatible.

We also study an alternative serialization scheme, Cap'n Proto, which is lacks many of the encoding and backwards-compatibility features of protobuf. The results of this comparison will help us understand which steps in object serialization are the most expensive, and will inform the design of our accelerator.

This work makes the following contributions:

- Study of the performance bottlenecks in serialization and deserialization
- Comparison of two different serialization schemes, with analysis of the tradeoffs between both schemes
- Proposal of the design of a hardware accelerator for serialization and deserialization, as well as preliminary infrastructure work towards that design

II. Related Work

A. Datacenter bottlenecks

Given the prevalence of cloud infrastructure running on shared datacenters, there has been much recent work on characterizing performance bottlenecks on datacenter workloads. Kanev et al. study Google's C++ workloads, and characterize the library and system calls which dominate runtime [1]. Another Google paper characterizes specifically frontend stalls from ICache misses and branch mispredictions at the warehouse scale [4]. Researchers have also studied specific variants of warehouse scale computing, for instance "function-as-a-service" systems [13].

In this work, we plan to characterize and accelerate a globally relevant component of datacenter applications. Structured message formats are important for large-scale storage systems, communication over networks, and remote procedure calls, and methods for characterizing protocol buffers are broadly applicable to other message formats as well.

Identify applicable funding agency here. If none, delete this.

B. Serialization

Prior work has explored optimizing serialization at the software level. Efficient object serialization for Java programs has been studied extensively, due to the widespread use of Java as an application runtime [11], [12]. These works primarily focus on algorithmic improvements, rather than hardware acceleration. Furthermore, the protobuf format is language independent, as objects can be serialized in one language and deserialized in another. Therefore, language specific optimizations are not ideal for developing serialization accelerators. Furthermore, these optimizations rarely consider architectural and micro-architectural features like the cost of pointer-chasing or the penalty of branch mispredicts - events which can incur significant latency. However, these algorithmic optimizations provide a reasonable starting point for developing accelerators.

XML serialization has also been studied extensively [10]. As a system with similar applications in storing/transmitting structured data conforming to a schema, many concepts applicable to XML serialization can be applied here. Similar efforts to develop hardware acceleration for XML [3] have had some success. However, there are some significant differences - XML is a heavily string-based, human-readable format, so serialization to XML involves many string conversions. XML is also more verbose than the protocol buffer format in terms of the size of the serialized byte string.

C. Accelerators

Prior work has explored the design and implementation of accelerators targeting tasks involving data-structure manipulation and exploration. Numerous groups have developed FPGA and ASIC-based accelerators for various compression algorithms, including GZIP [7], LZ77 [5], Snappy [8], and XML [3]. While data compression typically involves a serial traversal of the data, object serialization requires data-structure-aware pointer-chasing combined with encoding. Serialization should also be more memory-bound than compression, since theoretically serialization is primarily memory copying and localized encoding, while compression typically involves hash-table lookups.

Our work also shares characteristics with prior work on data-structure aware hardware, specifically with respect to the memory hierarchy. Prior work has explored data-structure aware compression and caching [14] [15]. In each of these works, some software-defined data-structure informs hardware behaviors beyond the instruction level. In this work, we aim to characterize how data-structure-aware hardware design can accelerate object serialization and deserialization.

The accelerator this work plans to inform is inspired by prior work on hardware acceleration for garbage collection [9]. The authors of that work designed a decoupled hardware accelerator to traverse the reference graph of a program concurrently with program execution. The acceleration of the graph traversal on hardware resulted in substantially improved overall program performance, due to reduced time spent in garbage collection. Our work aims to inform a similar accelerator, which would perform a limited set of data-structure traversal tasks to accelerate cases where a general purpose core struggles.

III. Protocol Buffer Formats

A. Protobuf

The baseline implementation of the protocol buffer standard is open-sourced by Google, and includes support for most commonly used languages (C++, C#, Java, Python, Dart, Go) - though in this work, we will primarily consider the C++ implementation as it is generally the most performant.

Data structures, called **messages**, are specified in a .proto file as a series of fields with strong types and optional modifiers:

- **required**, which causes a runtime exception if the field is left uninitialized when a serialization is attempted;
- **optional**, which may be left unset, and assigns a default value to the field if unset; and
- **repeated**, which can store an arbitrary-length ordered list of items in a single field.

The necessary classes in the target language are then automatically generated using **protoc**, the provided utility for compiling proto definitions. Protoc also allows for the use of custom plugins to modify the generated code, adding extra features where necessary. In the C++ implementation, each message class provides a standard API by which fields can be set, reset, or (in the case of repeated fields) appended to, along with methods to serialize to a string representation and parse from string.

B. Cap'n Proto

Cap'n Proto exists as an open source implementation only and is intended for use in C++. Data structures are defined as structs which can support field - sequence number pairs as well as enum definitions. Variable length fields are represented as lists. Once the proto datastructure is defined, the 'capnp' tool is used to generate a header and source file pair to be included in the application. One of the main features that separates Cap'n Proto from Protobuf is the fact that there isn't really any encoding/decoding overhead as the struct representation can be directly converted to bytes. That being said, the encoding is not platform specific as the data is fixed width, fixed offset, use little endian, and are properly aligned [16].

It is also important to note that because of the simplicity of "encoding", new fields always have to be appended to the struct as the sequence numbers need to be monotonically increasing without gaps. These restrictions sacrifice some flexibility of the protobuf format but allow Cap'n Proto to make more efficient implementation choices (Cap'n Proto uses arena allocation to put the proto into large continuous segments of memory).

IV. Evaluation Methodology

A. Goal

In order to evaluate the potential of a hardware accelerator for improving serializing performance, we sought to compare the performance of Protobuf with Cap'n Proto. Since Cap'n Proto advertises that it does not perform many of the tasks with Protobuf performs (for example, non-trivial encoding and decoding), we argue that Cap'n Proto approximates an implementation of Protobuf where the encoding and decoding steps are hardware accelerated to take 0 cycles. Thus, in our evaluation Cap'n Proto is a proxy for a perfect, instantaneous protobuf accelerator. We analyze the speedup from Cap'n Proto to determine the potential for our hardware accelerator.

B. Protobuf

Performance evaluation of the protobuf library consisted of two types of benchmarks:

- micro-benchmarks, which were used to identify serialization/parsing performance by datatype and isolate specific performance bottlenecks; and
- macro-benchmarks that utilize popular protos from open source projects, intended to provide a set of realistic protos as a baseline for comparison against other serialization schemes and evaluation of the accelerator.

1) Micro-benchmarks: Micro-benchmarking consisted of serializing and parsing simple protos with a single repeated field of different sizes, and measuring latency and throughput as a function of size. For example, the micro-benchmark used to profile Int32 performance was:

```
message Int32Pkt {
    repeated sint32 field1 = 1 [packed =
        false];
}
```

Each microbenchmark was run with a range of sizes for the repeated field array, for a large number of iterations to mask any overhead from the timer. During runtime, execution hotspots and memory traffic were recorded using the Linux perf utility, allowing for detailed analysis of good candidates for acceleration.

The Xeon portion of the micro-benchmark suite was run on a single core of an AWS EC2 c5n.metal instance - a 3.0 GHz Intel Xeon Platinum 8000-series processor with 32 GiB of RAM, using the standard Linux AMI. In order to provide baseline data for the design and evaluation of the accelerator, the suite was also cross-compiled to RISC-V and run on a simulated Rocket Chip core with a three-level cache hierarchy and simulated DRAM timing model. This simulation leveraged the FireSim FPGA-accelerated simulator infrastructure to run a Buildroot-based disk image on Rocket Chip. A similar methodology will later be used to evaluate the higher-performance BOOM (Berkeley Out-of-Order Machine) core, which will be the basis of the final accelerator design.

2) Common Proto benchmark: Proto benchmarks consisted of serializing protos from some of the most popular repositories on Github that used C++ protos as well as Google's own benchmarks. The purpose of this was to select actual workloads and then port the proto to Cap'n Proto for comparison. Serialization time and encoded bytestream size were measured with the modified parameters being the number of entries in repeated fields/ list size as well as the well as the proto batch size. Protos were populated with random data with length 50 strings. Although many Protobuf protos were benchmarked, only a few could be faithfully ported to Cap'n Proto due to the restrictions on field sequence ordering and only two protos could be properly compared using the 2 technologies.

V. Evaluation Results

A. Protobuf

1) Serialization: Based on Fig. 1, the in-order Rocket core performs substantially worse on serialization than the Xeon processor - by a factor of 3-6, depending on the benchmark. This suggests that Rocket is not a good platform on which to develop the accelerator compared to BOOM, because the less efficient core may prove to be a bottleneck.

Floating point protos serialized 4-5 times more quickly than integers of the same size on the Xeon processor. Fig. 2 points to additional overhead in the core serialization function, as well as a function that calculates the size of the encoded integer. This is necessary because integers are compressed during serialization by encoding as **varints**. The varint encoding algorithm is:

1) Store seven bits from the number (start with least significant) into the LSbs of the next byte of the varint encoded value.
2) If more bytes are required to represent the full number, set the MSb of the varint byte to 1 and repeat.

For example, to encode the value 300 (in binary, 0b0000000100101100):

1) The first byte is 0b10101100 or 0xAC - a 1 concatenated with the 7 LSbs of the binary value.
2) The second byte is 0b00000010 or 0x02 - a 0 (since no more bytes are needed) concatenated with the next 7 LSbs,

The full varint encoding is 0xAC02, compressing the value from 4B to 2B. Near-zero values are compressed very efficiently. However, calculating the size of the integer field and then performing the encoding takes a significant amount of CPU cycles.

String fields, meanwhile, serialized 10 times slower than floating-point fields of the same byte size. As before, examining the breakdown of function overheads during the string serialization benchmark in Fig. 2 is telling - a collective 35% of cycles are spent on tasks related to verifying the integrity of the UTF-8 string, a task that's expensive in software as it requires looping through the string byte-by-byte.

Serialization of strings also incurs substantial memory latency due to pointer traversal. String fields are-heap allocated,

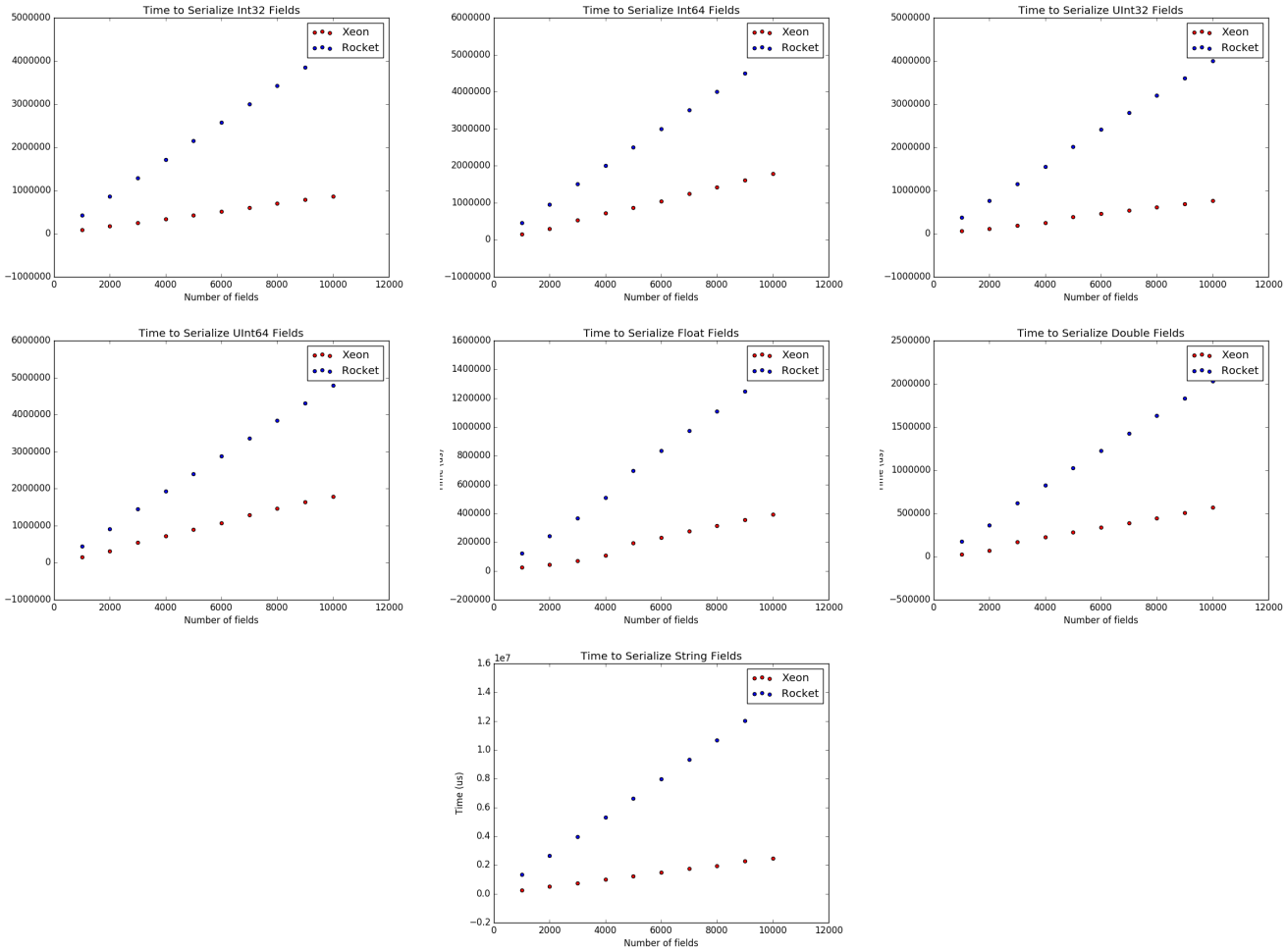

Fig. 1. Time to serialize each field type based on number of fields.

so serializing them involves jumping back and forth between the stack and the heap. Fig. 3 demonstrates that serializing string fields produces almost an order of magnitude more read traffic than write traffic, which is a direct consequence of this pointer chasing.

2) Parsing: Relative speeds based on datatype are largely similar for parsing and serialization. Benchmarks run on Rocket were generally 3 to 6 times slower compared to their Xeon counterparts. Integer fields were 3 to 6 times slower to parse compared to floating-point fields of the same byte size; an examination of function overheads for these micro-benchmarks (Fig. 5) points to varint encoding once again bottlenecking throughput, in a way that scales substantially with the size of the integer field. Effectively a decompression algorithm, the decoding step is inefficient in software for the same reasons as the encoding step, but is well suited to a small and efficient hardware implementation.

Parsing a string field is about 16 times slower than a floating point field of the same size, and incurs a large amount of memory traffic in both the read and write directions. The pointer chasing penalty described during serialization would cause both read and write traffic to be extremely high, as each heap-allocated field would have to be fetched, modified, and written back. This points to poor cache performance that would benefit from prefetching or manually-managed scratchpad memory.

Once again, UTF-8 structural and validity checking is responsible for a significant percentage of string parsing overhead. This checking is potentially more important for parsing than it is for serialization - parsed data will typically be received over a network or read off a disk, both of which are potentially lossy and prone to corruption. This check is not parallelizable, as a valid UTF-8 encoding has inter-byte dependencies, but relatively simple to perform iteratively in hardware.

B. Cap'n Proto

Two protos were compared between Protobuf and Cap'n Proto: Ultimaker and LuaPbIntf. The Ultimaker Proto consists of 12 float entries, and a repeated field of a submessage which itself consists of a float and an 64 bit integer. LuaPbIntf consists of 10 fields of various datatypes as well as 2 repeated

Function name	% of samples
Int32	
InternalSerializeWithCachedSizesToArray	53%
SInt32Size	14%
Int64	
InternalSerializeWithCachedSizesToArray	69%
SInt64Size	10%
UInt32	
InternalSerializeWithCachedSizesToArray	52%
UInt32Size	17%
UInt64	
InternalSerializeWithCachedSizesToArray	72%
UInt32Size	16%
Float	
InternalSerializeWithCachedSizesToArray	28%
Double	
InternalSerializeWithCachedSizesToArray	25%
String	
UTF8GenericScanFastASCII	21%
InternalSerializeWithCachedSizesToArray	11%
IsStructurallyValidUTF8	9%
VerifyUtf8String	3%
UTF8GenericScan	2%

Fig. 2. Breakdown of functions with the highest overhead for each serialization micro-benchmark, as captured by perf. Expressed as a percentage of total cycles in benchmark function, so overhead of benchmark setup is included.

	Read traffic (GiB)	Write traffic (GiB)
Int32	2.80	3.16
Int64	5.40	5.59
UInt32	2.80	3.16
UInt64	5.40	5.59
Float	2.68	2.66
Double	5.21	4.79
String	30.78	3.22

Fig. 3. Memory traffic for serialization micro-benchmarks by field type.

fields: string and integer. From the first graph in figures 4 and 5, it is apparent the inefficient encoding of strings and integers heavily impacts the serialization time of the Protobuf struct when compared to Cap'n Proto. When there are more than 1000 items in the repeated field, the serialization time takes more than 2 orders of magnitude longer to execute.

On the other hand, the encoding of Protobuf leads to a more efficient bytestring when compared to Cap'n Proto. However, the compression of Ultimaker, which consists mostly of floats ended up yielding a better compression ratio than the encoding of the LuaPbIntf proto which was rather heterogeneous with repeated fields consisting of integers and strings.

Cap'n Proto serves as a good lower bound for runtime as it essentially performs a memcpy from the struct in memory to yield the bytestream and performs no special encoding whatsoever. The simplicity of its serialization scheme also serves as a good upper bound for comparing space efficiency since it doesn't perform any compression whatsoever.

VI. Discussion

Our analysis of the protocol buffers library points to several opportunities for hardware acceleration. Both integer and string fields are compute-bound rather than memory-bound; that is, they fail to saturate memory bandwidth even on a high-performance Xeon core. This suggests that there are enough speed bottlenecks in the serialization and parsing algorithms for hardware acceleration to produce significant speedup. Furthermore, inefficiencies in memory accesses such as the pointer chasing while serializing strings are prime candiates for memory prefetching units.

When looking at the vast difference in runtime and space efficiency for serialization between Protobuf vs Cap'n Proto, it is important to consider the influence a programmer can have over the serialization process. First and foremost, the design choice of what proto technology to use should be a consideration itself. Protobuf's main strengths come in flexibility, backwards compatibility, and consistency. This all comes at a cost of efficiency. Cap'n Proto, in order to do serialize efficiently, requires sequence numbers in a chronological order without any any gaps. This makes it hard to retroactively add fields when updating the proto, but it enables the speedup seen in figures 4 and 5. Additionally, when considering the space efficiency of the structs, choices in what datatypes are being serialized can increase performance as well. One can cast represent inefficient datatypes as other efficient datatypes and reverse the representation on the parsing end to potentially yield a performance boost in space and time.

A. Protobuf Assembly Analysis

To better pinpoint the exact performance bottlenecks of Protobuf serialization on CPU, we profile the serialization code at the assembly level and analyze at which instructions the core spends most of its time.

In Fig. 9 we see that the core spends an unusual amount of time on the data-dependent branch in the VarInt serialization loop. This branch is totally unpredictable, since the number of iterations of the serialization loop is dependent on the value being serialized. Thus the core spends over 23% of the serialization time for integers executing this cmp instruction, or resolving a misprediction on the result of this cmp instruction. We also observe that the rest of the instructions dominating runtime are simple arithmetic shifts, which correspond to the right-shift by 7 in the VarInt encoding scheme. These shifts, while they require many instructions to execute, can be easily statically encoded in hardware as a fixed-function accelerator.

We also analyze the serializeDouble code in 10, and see that it does not spend most of its time in a data-dependent branch. The add-immediate likely dominates the trace because it is being used to increment the pointer into the array of doubles, while the mov is moving the doubles from the struct into the serialized format. In the end, this segment is fundamentally performing a memcpy, with no data-dependent branches on the contents of the double fields.

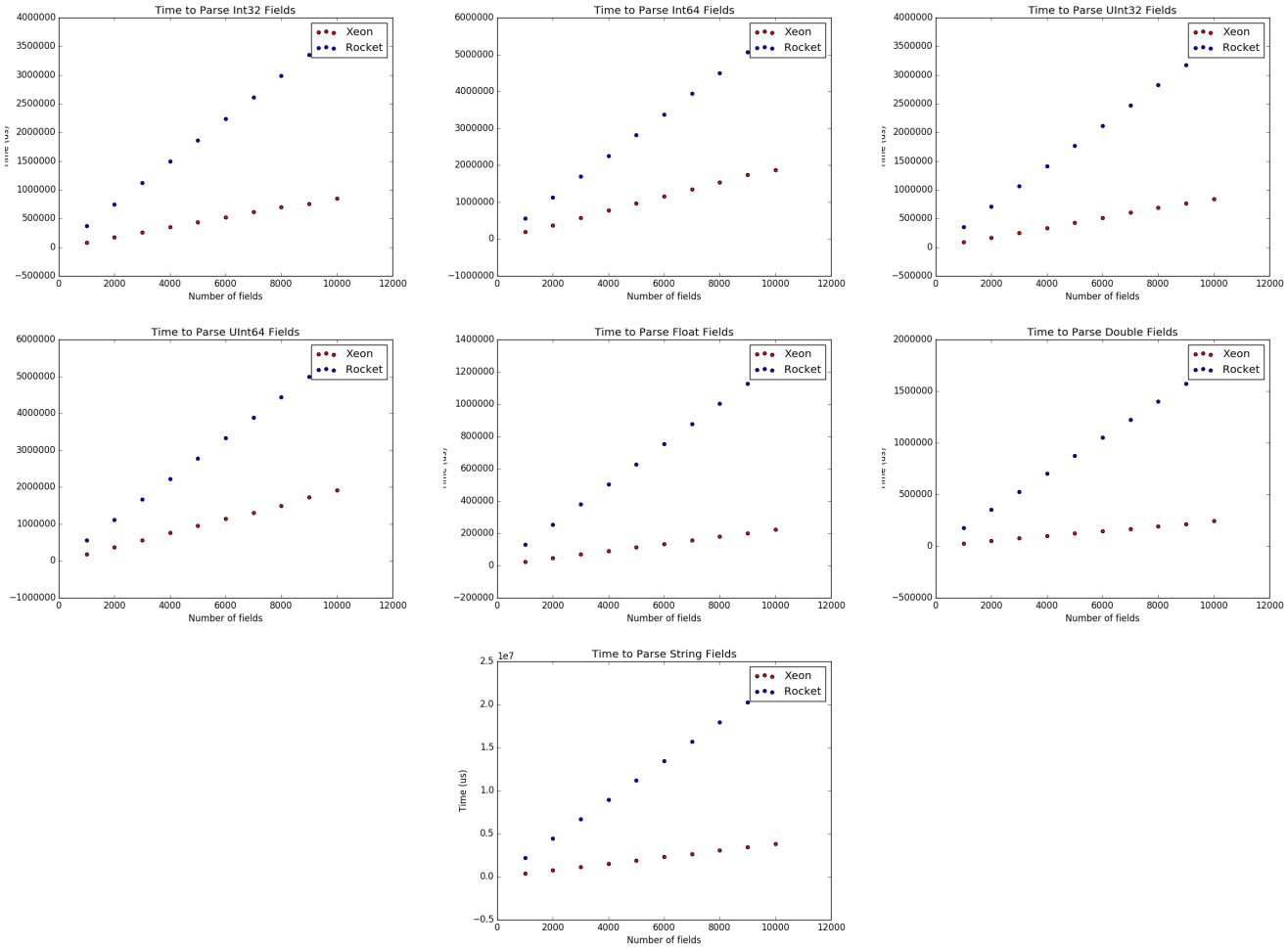

Fig. 4. Time to parse each field type based on number of fields.

VII. Hardware Proposal

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

A. Base Architecture

We plan to develop our accelerator as a co-processor attached to a BOOM+RocketChip SoC. The baseline BOOM+RocketChip SoC should closely mimic the behavior of a commercial Xeon server-class chip, and display similar performance characteristics.

For the core of our SoC, we use the open source BOOM out-of-order RISC-V core. Like commercial server cores, the BOOM core will aggressively execute instructions speculatively and out-of-order. The BOOM core also can fetch, decode, and execute multiple instructions per cycle. For this project, we sought to improve the performance of the BOOM core to make it more closely match the performance of a commercial Xeon. This involved reimplementing the instruction fetch unit to be of higher performance. The state-of-the-art TAGE prediction algorithm was implemented in BOOM, and lowers BOOM's branch misprediction rate to be competitive with commercial cores.

For the memory system, we use a 3-level cache hierarchy, with cache sizes matching those of commercial cores. The first level cache is part of the BOOM core, and has access bandwidth of 16bytes per cycle, similar to recent desktop processors. The second level cache is SiFive's open-source L2, which can be configured to be 4MB in capacity, with multiple banks to improve bandwidth. The third-level cache is represented by FireSim's LLC cache timing model, which accurately models the L2 penalty by keeping a small "cache" of tags in the software driver for the simulation. In addition, we place a next-line prefetcher between the L1 and L2, and a stream-prefetcher between the L2 and L3.

Substantial work was performed on improving this baseline system throughout the semester. Given that Protobuf serialization is primarily a bottleneck for datacenter processors, which are typically high-performance out-of-order cores, it is insufficient to design an accelerator around an in-order core.

Throughout the semester, we had to resolve several

Function name	% of samples
Int32	
_InternalParse	27%
VarintParseSlow64	24%
VarintParse	24%
Int64	
VarintParseSlow64	54%
_InternalParse	19%
VarintParse	16%
UInt32	
VarintParse	27%
VarintParseSlow64	26%
_InternalParse	25%
UInt64	
VarintParseSlow64	56%
_InternalParse	17%
VarintParse	15%
Float	
_InternalParse	35%
Double	
_InternalParse	27%
String	
UTF8GenericScanFastAscii	22%
IsStructurallyValidUTF8	10%
VerifyUTF8	9%
InlineGreedyStringParser	4%
InlineGreedyStringParserUTF8	4%
_InternalParse	3%
UTF8GenericScan	2%
VerifyUtf8String	2%

Fig. 5. Breakdown of functions with the highest overhead for each parsing micro-benchmark, as captured by perf. Expressed as a percentage of total cycles in benchmark function, so overhead of benchmark setup is included.

	Read traffic (GiB)	Write traffic (GiB)
Int32	5.04	2.00
Int64	9.42	4.03
UInt32	5.11	2.04
UInt64	9.41	4.00
Float	4.56	1.96
Double	8.64	4.01
String	33.25	25.72

Fig. 6. Memory traffic for parsing micro-benchmarks by field type.

memory-system bugs that blocked running memory-intensive compute workloads on BOOM. These bugs were coherency issues related to the L1 data cache's interactions with the cohesive L2, and were resolved by connecting BOOM to a automated memory trace generating tool. We also had to improve BOOM's branch prediction performance. A RTL implementation of the TAGE branch predictor improved IPC by 80%, and makes BOOM comparable to commercial Linux-capable cores.

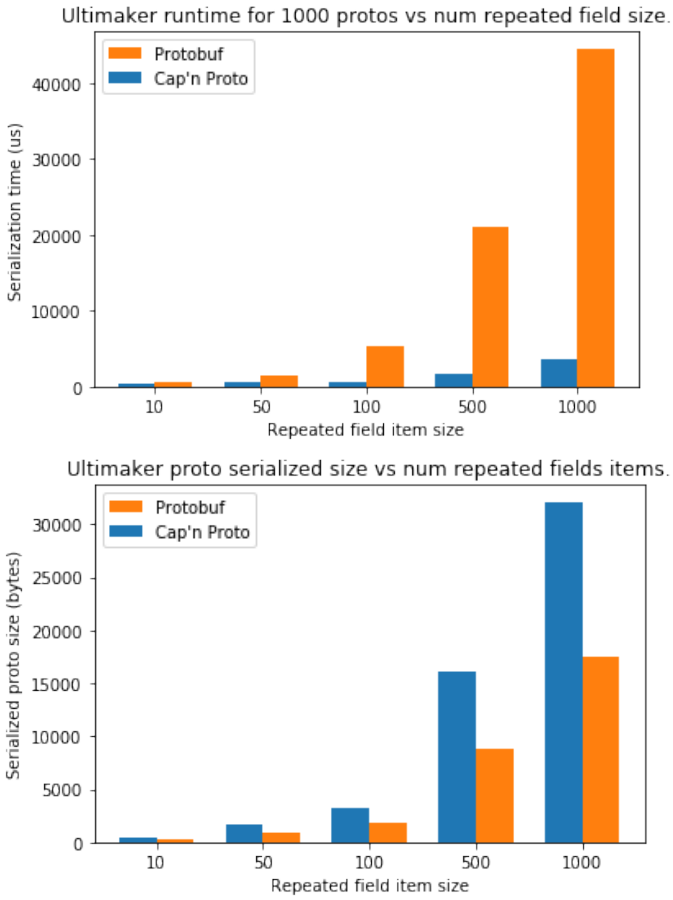

Fig. 7. Runtime and serialized bytestream size results when serializing messages across the Ultimaker proto.

B. Accelerator Design

Our accelerator will be a RoCC-directed coprocessor (Fig. 11), primarily targeted at accelerating the encoding processes of Protobuf serialization. It will be programmed via RISC-V custom instructions through the RoCC instruction interface of a Rocketchip SoC. The BOOM core on our SoC will decode our custom serialization instructions and send the decoded address of the object to be serialized to the accelerator.

We observed that the majority of serialization time was spent in serializing repeated fields. In these fields, each element was sequentially serialized to the destination buffer, with significant overhead being spent on encoding, buffer management, and elementwise memory accesses. While these operations are abstractly elementwise, they are being executed with no parallelism or pipelining. Thus we observe a natural opportunity for capturing the untapped parallelism in hardware.

Our proposed hardware will read, encode, and store sequential chunks of a repeated element field in parallel to the destination array. Additionally, the read, encode, and write stages will be pipelined, such that as one chunk of serialized data is being written, another is being encoded, and another is being read.

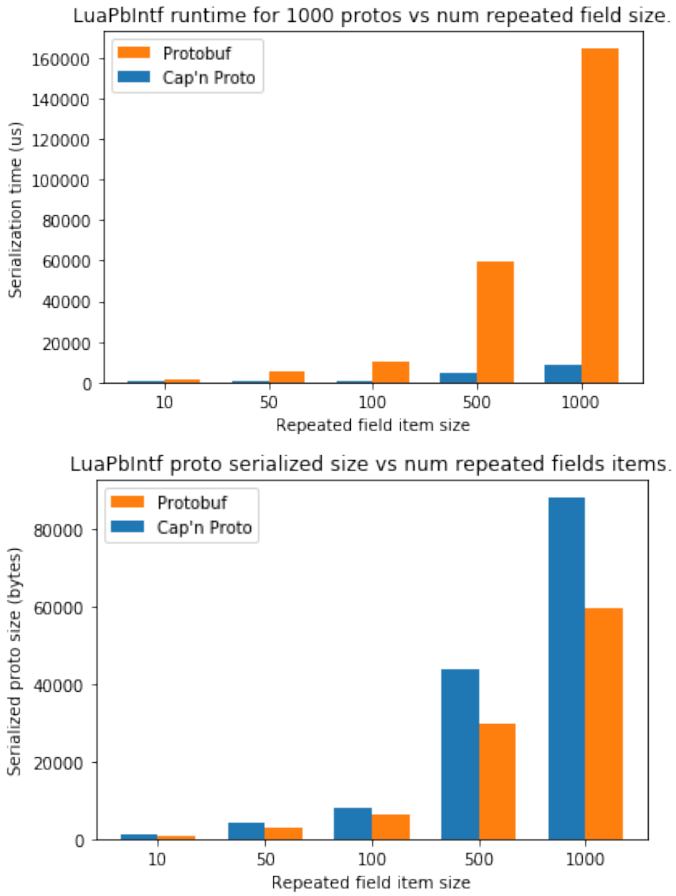

Fig. 8. Runtime and serialized bytestream size results when serializing messages across the LuaPbIntf proto.

Fig. 9. Int64 Serialization Code

```
1.93 :      mov     %eax,%edx
0.48 :      shr     $0x7,%rax
1.54 :      or      $0xffffff80,%edx
2.07 :      cmp     $0x7f,%rax
0.53 :      mov     %dl,0x1(%rbx)
1.97 :      lea     0x2(%rbx),%rdx
0.00 :      jbe     40ef40
6.73 :      mov     %eax,%ecx
7.21 :      shr     $0x7,%rax
5.58 :      add     $0x1,%rdx
8.47 :      or      $0xffffff80,%ecx
4.82 :      mov     %cl,-0x1(%rdx)
     :        if (value < 0x80) {
     :          ptr[1] = value;
     :          return ptr + 2;
     :        }
     :        ptr++;
     :        do {
23.44:      cmp     $0x7f,%rax
0.00 :      ja      40eee6
```

We also observe that since the protobuf specification allows ”holes” in the encoding to represent unknown fields. This

Fig. 10. Double Serialization Code

```
5.07 :      mov     0x18(%r13),%rax
6.67 :      add     $0x9,%rbx
3.62 :      mov     (%rax,%r12,8),%rax
38.26:      add     $0x1,%r12
6.81 :      movb    $0x9,-0x9(%rbx)
4.20 :      mov     %rax,-0x8(%rbx)
7.39 :      cmp     %r12d,%r15d
```

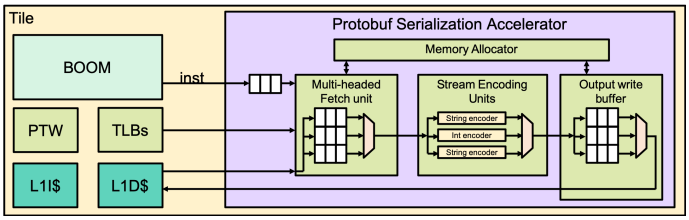

Fig. 11. Block diagram of accelerator structure.

enables parallelism between the accelerator and the control core, as the general purpose CPU does not need to know how much memory the accelerator has written before serializing the next object. Rather, the general purpose CPU just needs to know the upper bound for the size of the serialized object the accelerator will handle.

C. Interface and ISA

The accelerator will be attached via the RocketChip RoCC interface to a BOOM general-purpose core. For accelerating serialization, each custom RoCC instruction will correspond to a serialization command for one type of field (int, float, string), and will indicate the start address of the data-to-be serialized, and whether or not that field is repeated. For deserialization (parsing), we observe that since the first byte of any serialized field contains information on the type of the field, we can use the accelerator to quickly scan the byte stream, and the control core to decode the types of fields. Thus our instructions are:

Instruction	Behavior
`seroutput`	rs1 contains address of output buffer rs2 contains size of output buffer
`serfield`	Begin serializing this field to output buffer. rs1 contains address of field. opcode contains type.
`parfield`	Begin parsing this serialized field rs1 contains source address. rs2 contains output address. opcode contains type. rd indicates source address of next field
`chksuccess`	Check success of accelerator rd indicates success of prior operations

1) Serialization: For serialization, the general-purpose core will allocate a worst-case estimate of the size of the output serialized data. This can be conservative, since if the accelerator

decides it has not enough memory left for the serialized data, it can safely fall back to the standard CPU serialiation routine.

For every field in the proto, the core will emit a serfield instruction that contains the source address of the field, as well as the type. The accelerator's memory-allocation manager will assign some subset of the total allocated output buffer to this field, and perform serialization and encoding of that type. After issue of instructions for each field, the core can use the `chksuccess` command to determine if the accelerator failed at any previous tasks. If the accelerator failed, the code will fall back to the standard serialization routines.

2) Parsing: For parsing, the general-purpose core will decode the type bits in the serialized format, and determine which field in the proto is currently being parsed. Upon reading the type bits, the `parfield` command will instruct the accelerator to parse a field of a specific type. Unlike serialization, we do not know the address of the next field after the current one until we have read the serialized stream. Thus we use the return value of this instruction for the accelerator to pass back to the general-purpose core the address of the next field in the serialized format.

D. Load/Store Stream Units

We observe that we are guaranteed that the source and destination buffers will never overlap. In other words, serialization is never done in-place. Thus load-to-store ordering, and load-to-load ordering do not need to be maintained in the accelerator's DMA unit. However, we observe that for serialization, the addresses of the source fields are not necessarily contiguous, and so it is beneficial to perform loads out-of-order. Thus our load memory unit will allow loads from multiple "heads", or streams to be inflight at once.

For the same reason, the store unit can execute stores out-of-order, since the stores will not conflict with each other, or with the loads. For serialization, the memory allocation manager will allocate segments of the large originally-allocated buffer for each field, and assign these to the store units.

E. Encoding/Decoding Units

Since we observed that a substantial performance bottleneck in serialization and parsing is the encoding scheme for some types, the encoding units will be designed to perform the complex encoding tasks in hardware. For example, serializing a 64 bit integer can be performed in a pipelined functional unit, instead of iteratively as it is performed on a general purpose core. The iterative method eats up many of the functional units on the core for computing shifts and branches, but we can hardcode a pipelined implementation of this algorithm in our accelerator instead.

F. Evaluation Plan

In order to collect baseline numbers for the performance of the protocol buffers library on Rocket Chip, we simulated a Rocket core with full memory hierarchy using FireSim [17] and developed a Buildroot-based workload to cross-compile and run the protobuf micro-benchmarks on the core. This workload is fully compatible with BOOM, and will be used to evaluate the performance of the accelerator once implemented in the Chisel DSL.

VIII. Conclusion

A detailed analysis of Google's implementation of the protocol buffers library reveals several computation inefficiencies that should lend themselves well to specialized hardware acceleration. Speed and throughput are heavily bottlenecked by the encoding schemes used in the protobuf standard, as well as the memory access patterns characteristic of the Google implementation.

While there are software-based mitigation strategies possible, as seen in the implementation choices of Cap'n Proto, they sacrifice some level of flexibility in order to gain efficiency. Specialized hardware provides a path to a solution that is far more efficient in time and power while being truly transparent to a software developer, and allowing existing systems dependent on protobufs to be ported with minimal effort.

The availability of a performant and extensible superscalar, out-of-order core in BOOM provides a useful analog for modern warehouse-scale processors, and provides a development-friendly platform (along with the RoCC interface) on which to develop the accelerator in the future.

Acknowledgment

The authors would like to thank Professor Kubiatowicz for his continued guidance and support throughout this project. Additionally, the support of several collaborators within the ADEPT lab (Sagar Karandikar and Nathan Pemberton) was instrumental in helping develop our simulation and benchmarking infrastructure. We would also like to extend our thanks to the developers of the Rocket Chip, BOOM, and FireSim projects, which were directly necessary for our ongoing work on this project.

References

[1] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA 15, 2015

[2] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the Killer Microseconds," Communications of the ACM, vol. 60, no. 4, pp. 48–54, 2017.

[3] Z. Dai, N. Ni, and J. Zhu, "A 1 cycle-per-byte XML parsing accelerator," Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays - FPGA 10, 2010.

[4] Ayers, Grant, et al. "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers." Proceedings of the 46th International Symposium on Computer Architecture. ACM, 2019.

[5] Fowers, Jeremy, et al. "A scalable high-bandwidth architecture for lossless compression on fpgas." 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, 2015.

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

[7] Jun, Sang Woo, et al. "Zip-io: Architecture for application-specific compression of big data." 2012 International Conference on Field-Programmable Technology. IEEE, 2012.

[8] Kovacs, Kyle. "A Hardware Implementation of the Snappy Compression Algorithm." UC Berkeley EECS Masters Thesis (2019).

[9] Maas, Martin, Krste Asanović, and John Kubiatowicz. "A hardware accelerator for tracing garbage collection." Proceedings of the 45th Annual International Symposium on Computer Architecture. IEEE Press, 2018.
[10] Maeda, Kazuaki. "Performance evaluation of object serialization libraries in XML, JSON and binary formats." 2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP). IEEE, 2012.
[11] Opyrchal, Lukasz, and Atul Prakash. "Efficient object serialization in Java." Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware. IEEE, 1999.
[12] Philippsen, Michael, and Bernhard Haumacher. "More efficient object serialization." International Parallel Processing Symposium. Springer, Berlin, Heidelberg, 1999.
[13] Shahrad, Mohammad, Jonathan Balkind, and David Wentzlaff. "Architectural Implications of Function-as-a-Service Computing." Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2019.
[14] Tsai, Po-An, and Daniel Sanchez. "Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy." Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2019.
[15] Tsai, Po-An, Nathan Beckmann, and Daniel Sanchez. "Jenga: Software-defined cache hierarchies." 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.
[16] "Introduction," Cap'n Proto: Introduction. [Online]. Available: https://capnproto.org/. [Accessed: 15-Dec-2019].
[17] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), 2018.