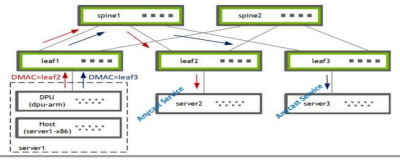
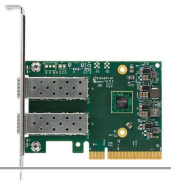
Evolving P4Runtime from Switch to DPU

Milind Chabbi*, Dmitry Kozlyuk, Alan Lo

NVIDIA

P4Runtime - Switch versus DPU

- P4Runtime is the defacto API for controlling P4 programmable data planes
- Use cases and requirements have evolved
- Need for a fresh look at the scale and performance of P4Runtime APIs

Feature 	**Switch** 	**DPU** 
Data center topology	Spine, Tor	Edge (hypervisor)
scale - table size	MBs	GBs
performance - entry update rate	~100K / sec	~10M / sec
throughput	up to 50 Tbps	up to 400 Gpbs

How do we evolve P4Runtime performance?

Where we are now

- Table update rate ranges from 100K - 400K entries / sec
- P4Runtime PI library (1.3 spec)

Where we need to go

- > 10 million insertions / sec (e.g. during VM live migration, local controller)
- PI API compatibility (preserve what we already have)

Proposals

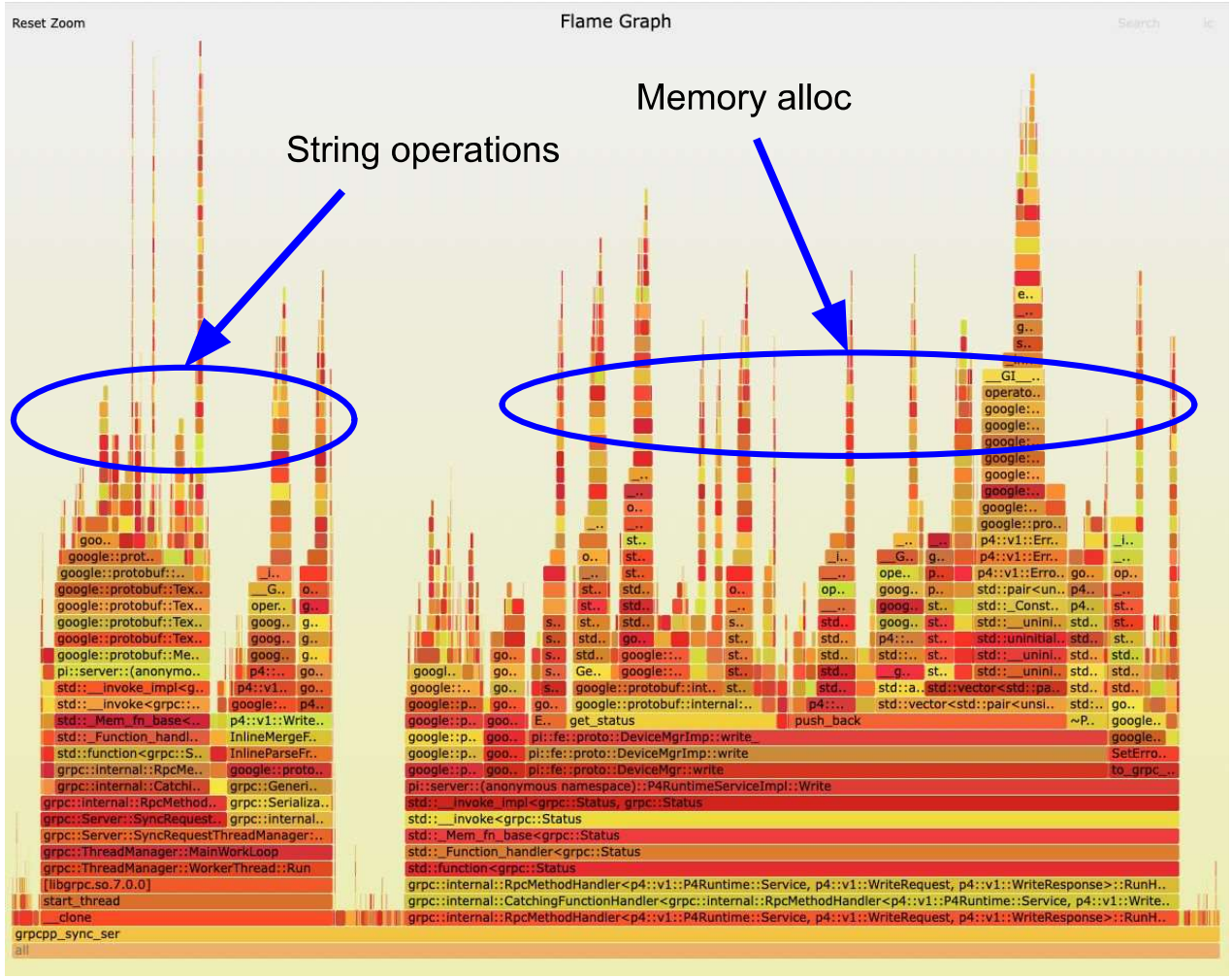
- Local controller uses shared memory
- Remote SDN controller uses bulk table updates

⇒ Not vendor specific, benefits the entire community

Why is it slow?

Protobuf and gRPC

- Hierarchical, dynamic data structures ⇒ Low memory locality
- Lot of memory management (alloc)
- Data parsing and string operations



Shared-memory for local controller

Shared-memory for table insertion/deletion/updates

No API change for protobuf creation

Substitute

```
p4::v1::p4runtime::Stub->Write(ClientContext,
p4::v1::WriteRequest, p4::v1::WriteResponse)
```

With

```
pi::fe::local::Write(p4::v1::WriteRequest)
```

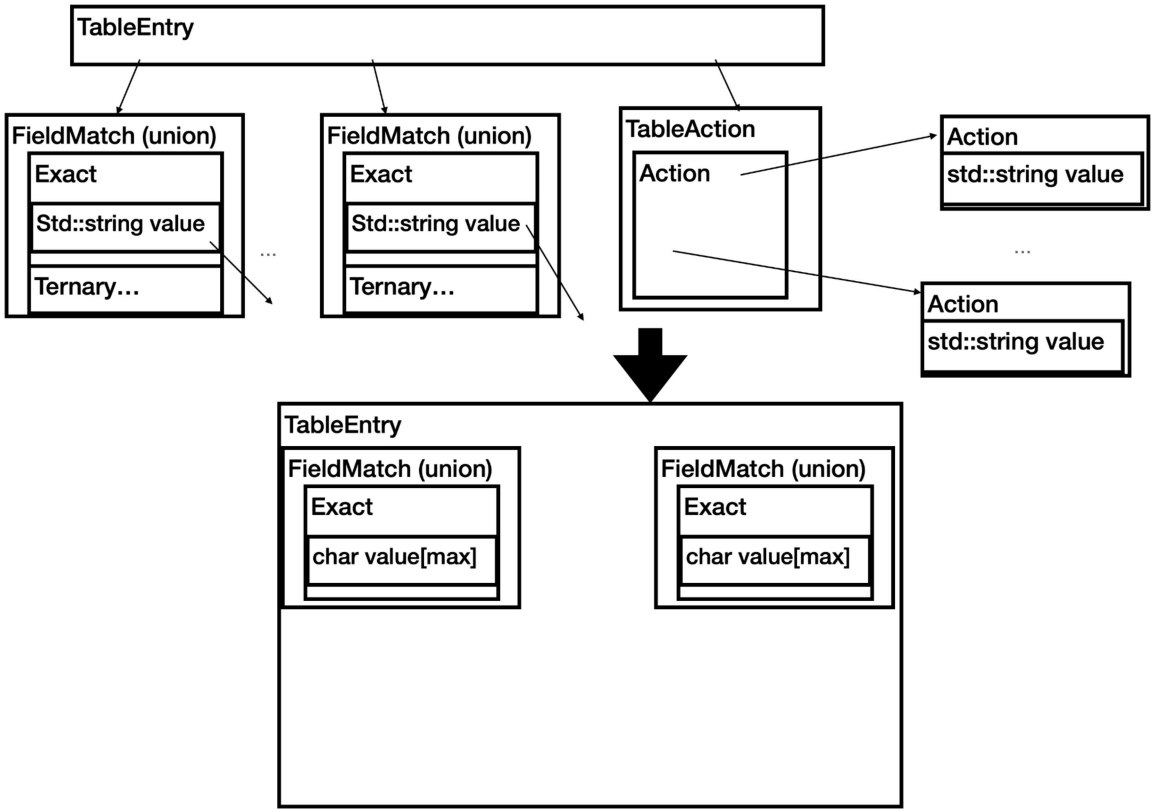
Example

```
for (...) {
      TableEntry *t = new TableEntry();
      FieldMatch *f = t->mutable_match();
      FieldMatch_Exact * exact = f->mutable_exact();
      std::string* value = exact->mutable_value();
      value->append("some value");
}
…
p4::v1::WriteRequest request;
request.set_device_id(dev_id);
auto update = request.add_updates();
update->set_type(p4::v1::Update_Type_INSERT);
auto entity = update->mutable_entity();
entity->set_allocated_table_entry(match_action_entry);
p4::v1::WriteResponse rep;
ClientContext context;
Status status = pi_stub_->Write(&context, request, &rep);
```

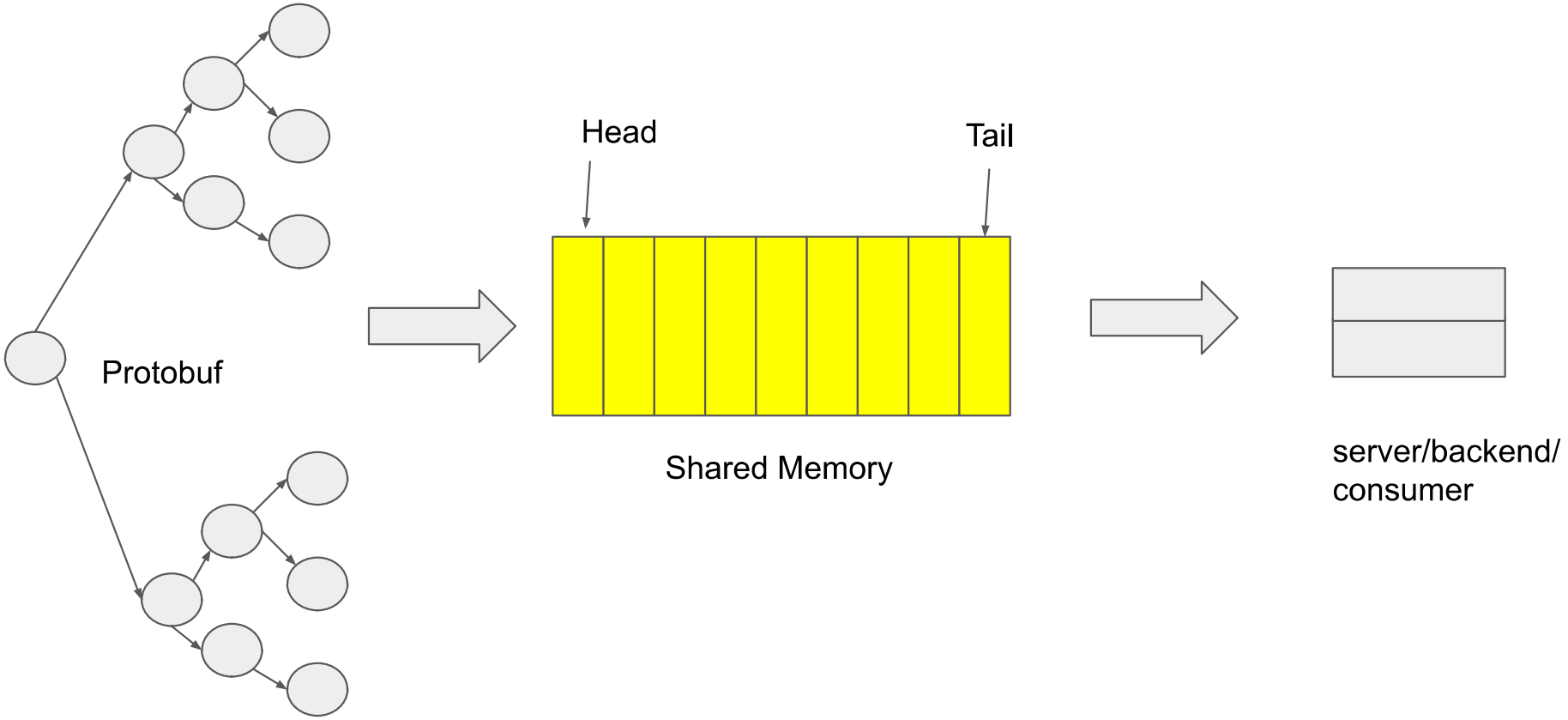
Proposed:

```
Status status = pi::fe::local::Write(&request);
```

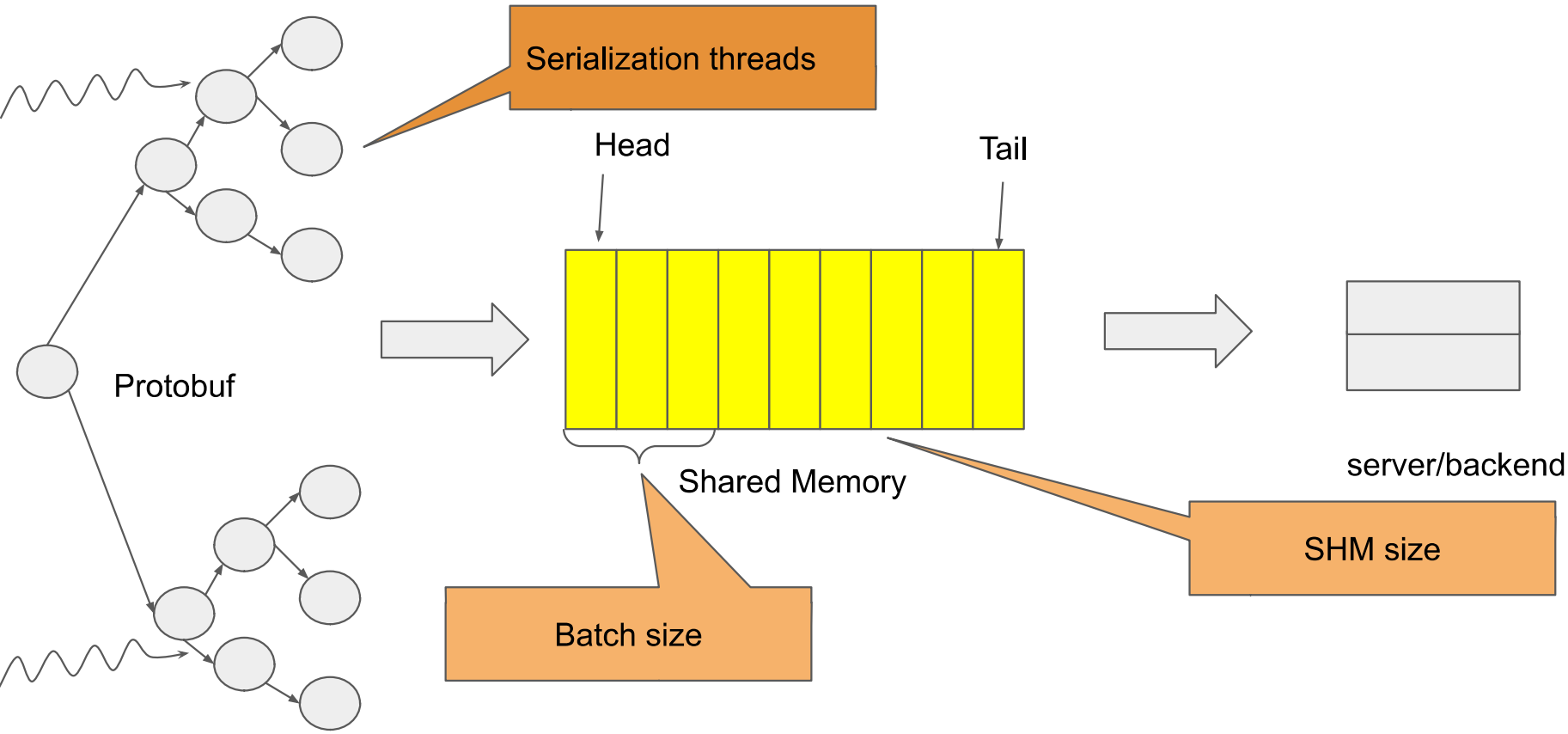
Serialize Protobuf to C-style Struct



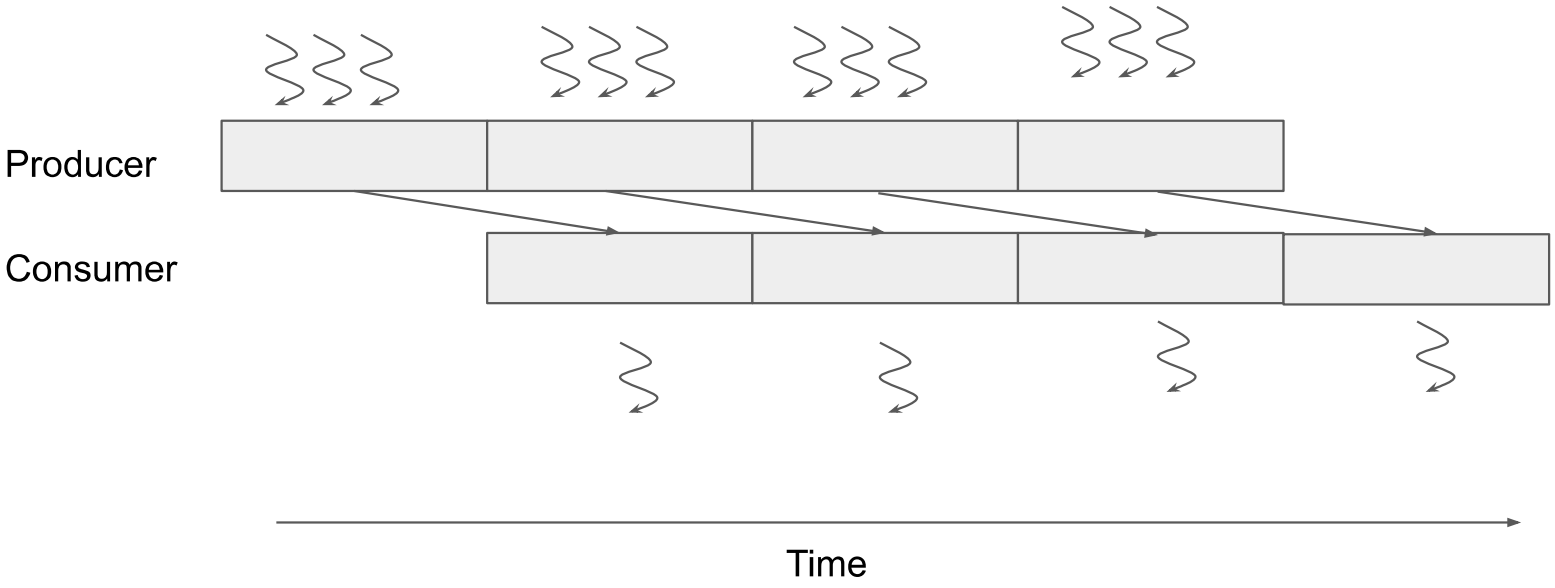
Implementation



Tunables


Serialization threads
Head
Tail
Protobuf
Shared Memory
server/backend
SHM size
Batch size

Pipeline parallelism

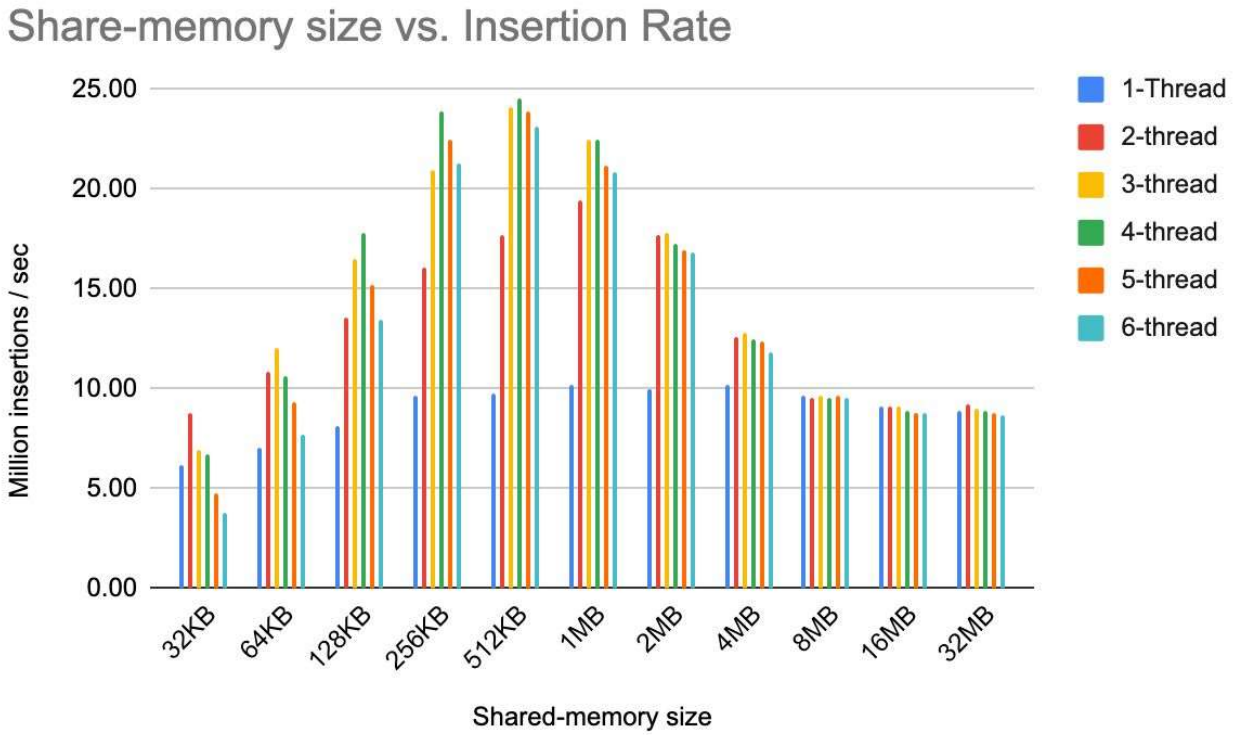


Experimental setup

1. BlueField2 (8 ARM cores)
2. 50M table entries
3. One consumer thread
    a. Consumer copies the entire available batch of data via memcpy of at most 10K entries
4. All numbers are average of 10 runs
    a. MAX reaches > 25M entries/sec
5. 380 bytes/SHM struct
    a. MAX_FIELD_MATCHES 5
    b. MAX_PARAMS 8
    c. MAX_MASK_STR 16 bytes
    d. MAX_VALUE_STR 16 bytes

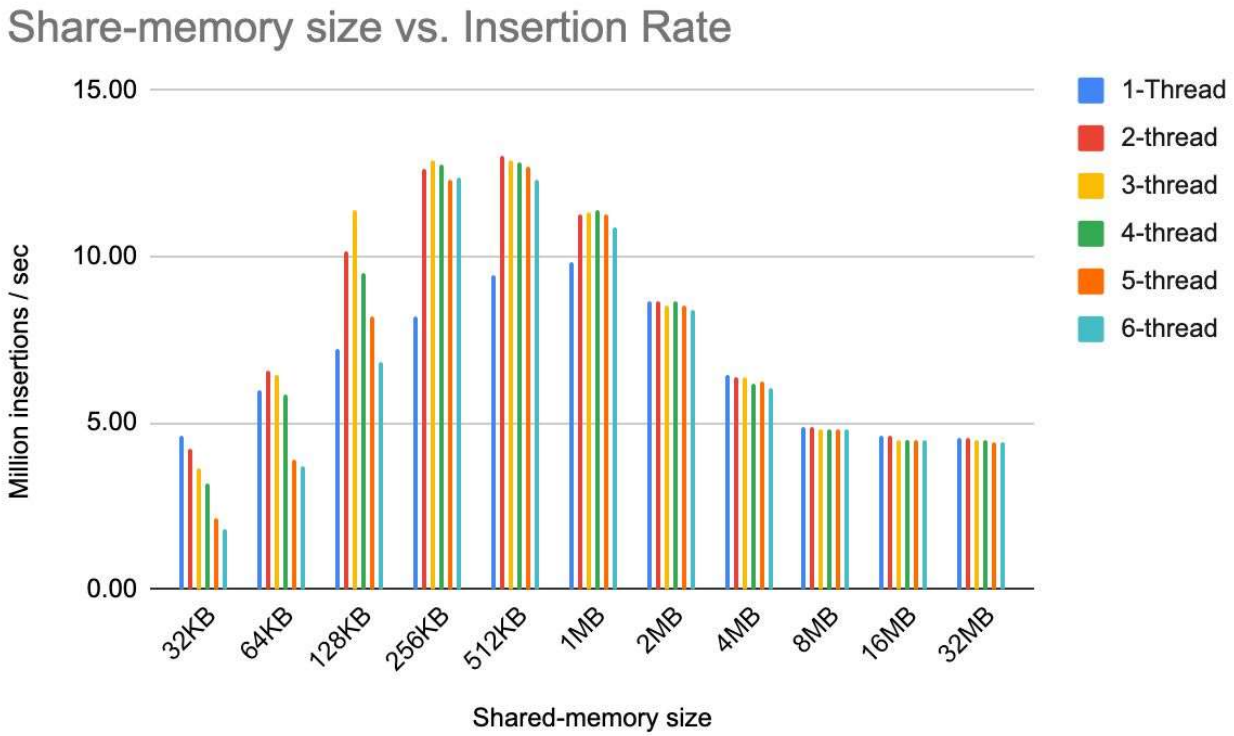
Configurable during compile time

Use case 1: 5-tuple and 0 action param



Batch size = SHM/4

Use case 2: extreme 16 tuple and 8 action params



Batch size = SHM/4
Struct size = 820 bytes

Things to know

1. Operation must be one of
   a. p4::v1::Update_Type_MODIFY
   b. p4::v1::Update_Type_INSERT
   c. p4::v1::Update_Type_DELETE
   If anything else ⇒ runtime error and partial/corrupted table
2. Additional cost of upfront update type validation is not evaluated
3. Updates exceeding MAX_xxx struct values ⇒ runtime error and partial updates
4. Can optimize the 380 byte large struct and MAX_xx limits by creating different SHM pools and having pointers/offsets from one to another
5. Need to introduce this P4 API
   a. pi::fe::local::Write(...)

Bulk table updates for remote controller

P4 Runtime Protobuf - table entry add

Observations

- Hierarchy is flexible. Any table with any valid entry
- But not efficient in certain common DPU use cases

Goal

- Support more efficient and performant P4 Runtime API
- Can be used by remote P4 Runtime controllers (also local)
- Targets the common use case of single table update, many entries
- Share with community (PR to be submitted to p4lang)

P4 Runtime - Entry Insertion Performance

PI Proto

```
message FieldMatch {

  uint32 field_id = 1;

  message Exact {

  bytes value = 1;

  }


      . . . . .


}
```

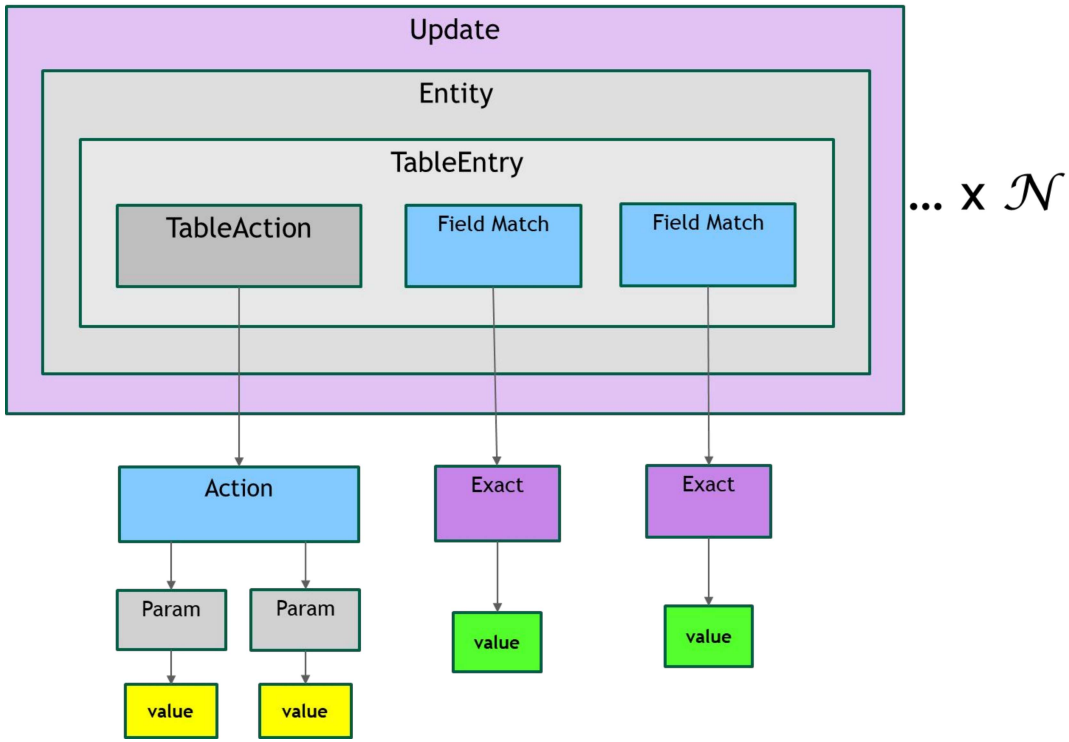
Bulk Proto

```
message FieldMatch {
      uint32 field_id = 1;
      bytes indirect_data = 2;

      message Exact {
            bytes direct_values = 1;
            uint32 direct_value_size = 2;
            repeated uint64 indirect_offsets = 3;
            repeated uint32 indirect_sizes = 4;
      }

      oneof type {
            Exact exact = 3;
      }
}
```

P4 Runtime - Entry insert write request

P4 runtime protobuf is defined in a way that each value is wrapped in multiple layers of messages.
Protobuf is parsed sequentially, depth-first; parsing cannot be done in parallel.

- Parsing is slow, it is O(N) where N = number of updated entities.
- Processing is slow, because each entry must be checked for validity.
- Message overhead is large, e.g. 4+4+2+2 bytes match data & 4+2 bytes action data = 18 bytes of data take >80 bytes for Update message (x4).



P4 Runtime - Entry insert Bulk write request

UpdateSet: the operation.

EntitySet: the type of entities updated

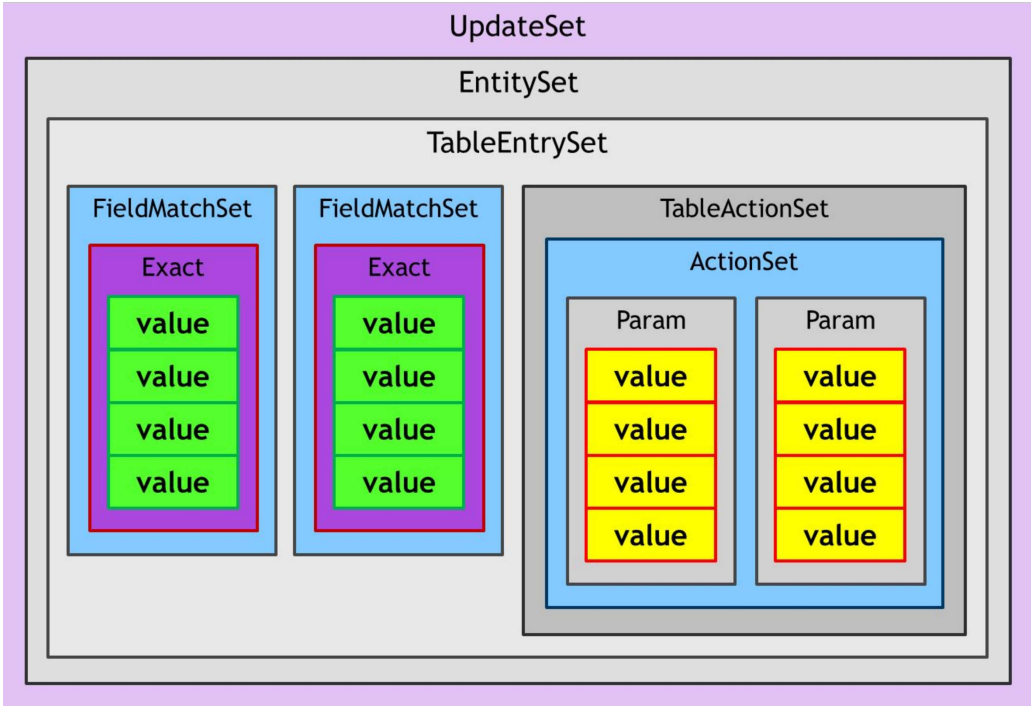
TableEntrySet: the table to update with N entries

FieldMatchSet: values for one field of all entries

TableActionSet, *ActionSet*: containers

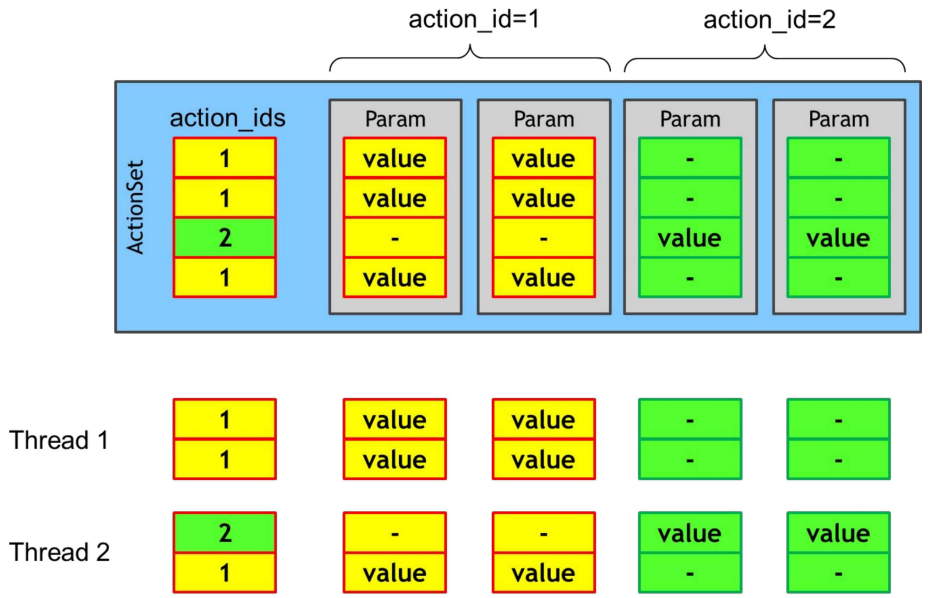
Param: values for one parameter of one action for all entries

- Number of messages does not depend on the number of entries
- Arrays of values can be represented as bytes, i.e single TLV in protobuf for fast parsing
- Optimized for bulk insertion into one table
- Simplest use for TableEntrySet with only entries for one action



P4 Runtime - Table entry set with different actions

- split the set into shards for processing scalability.
- tradeoff between size and speed
- in each value array one index corresponds to one and the same entry;
- the entry uses values from parameters of its action, values in other arrays are unused.
- Alternative would be to scan action ID array and calculate offsets.



P4 Runtime - some bulk results

Layer	Operations / sec	Comments
gRPC	945,000	C, deserialize with arena
gRPC bulk	9,289,000	C, deserialize with arena
libPI	**598,000**	**no gRPC, already deserialized**
bulk PI	**5,996,000**	**no gRPC, already deserialized**

* Broadwell - Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz

P4 Runtime - Next steps

P4Runtime local

- Move beyond POC to full implementation
- Get community feedback

P4Runtime bulk

- Submit PR to P4-API WG for comments
- Further refinements and use cases to be explored

Thank you!