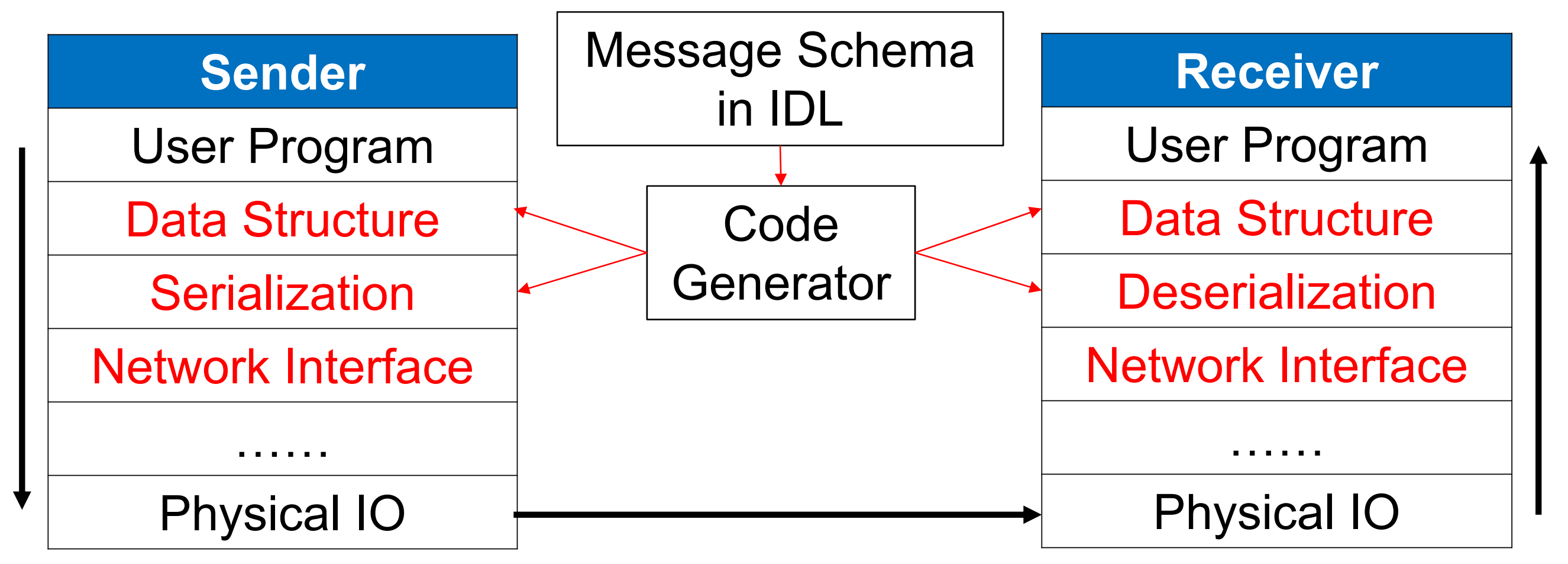
HGum: Messaging Framework for Hardware Accelerators

Sizhuo Zhang (szzhang@mit.edu MIT), Hari Angepat (Hari.Angepat@microsoft.com Microsoft),
Derek Chiou (dechiou@microsoft.com Microsoft)

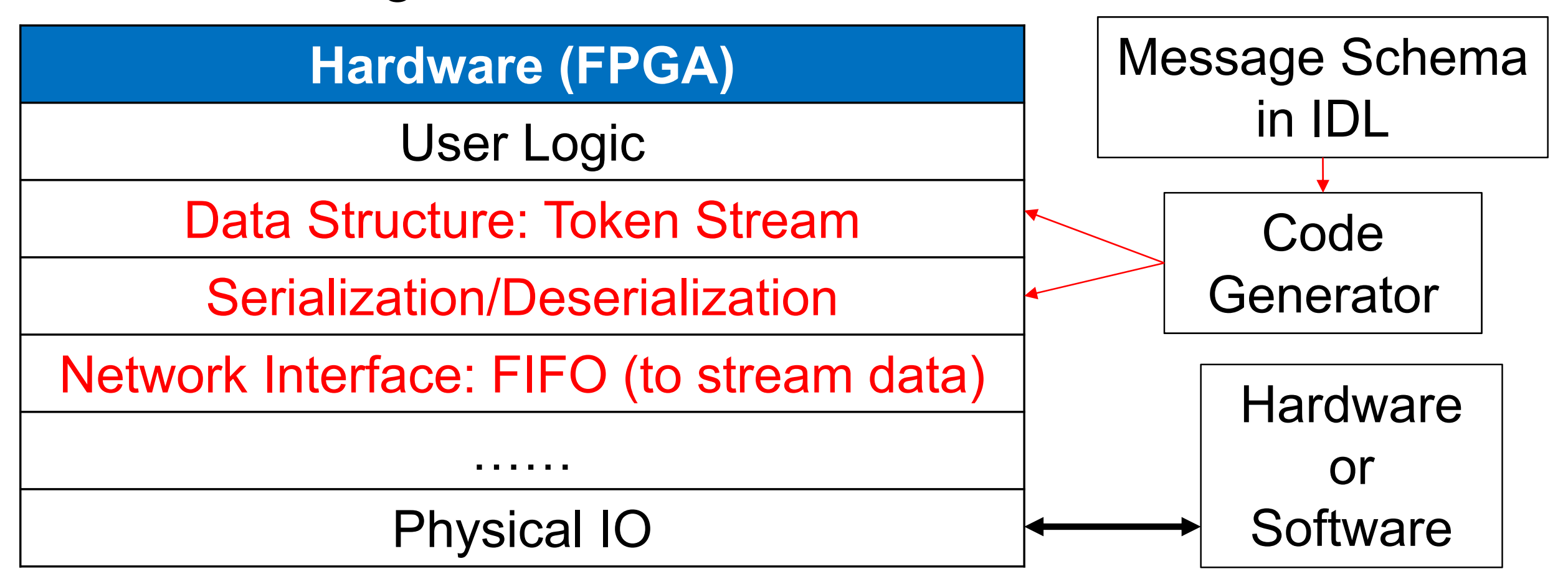
Background

- Messaging between two SW programs on two machines
  - Network transfer (e.g. Ethernet) is handled by libraries
  - Still need to encode data structure into binary format
  - Writing Ser/Des functions is tedious and error-prone
- Software messaging framework
  - Describe message schema (format) in an Interface Definition Language (IDL)
  - Automatically generate Ser/Des functions



Motivation

- Hardware accelerators: talk to SW host and other HW
  - Microsoft Catapult: accelerate Bing Search with 7 FPGAs
  - Also need a framework for generating Ser/Des functions
- Unique challenges for HW messaging frameworks
  - What is the data structure for HW?
  - Limited on-chip buffer: must stream data
  - Ser/Des logic must be fast and small: never be bottleneck

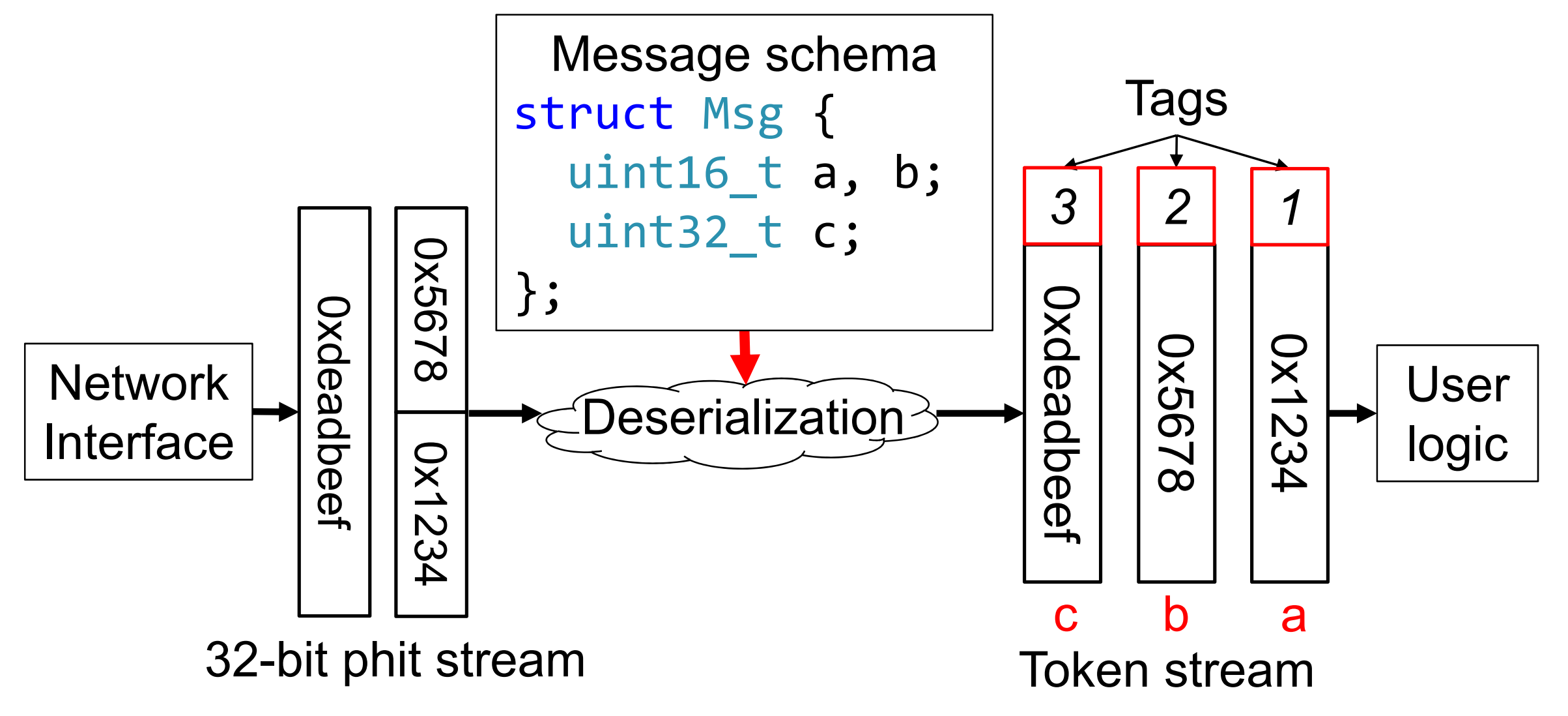

Hardware (FPGA)
User Logic
Data Structure: Token Stream
Serialization/Deserialization
Network Interface: FIFO (to stream data)
……
Physical IO
Message Schema in IDL
Code Generator
Hardware or Software

Contribution

- HGum: Messaging framework for HW accelerators
  - User specifies message schema in an IDL (JSON)
  - Support complex data types
    - -- Fixed-length data
    - -- Structure
    - -- Array (length known before generating elements)
    - -- List (length unknown until all elements are generated)
    - -- Arbitrary nesting of above types
  - High throughput, high clock frequency, small area cost

Framework Overview

- Deserialization: fixed-size data (phit) stream to token stream
  - Token: lowest-level field of the message structure
  - Output tokens are accompanied with user-specified tags



Framework Overview (cont.)

- Serialization: token (without tag) stream to phit stream
- Each message corresponds to a fixed token stream
  - Example: List has 1 element and Array has 2 elements

Schema in C++	Token Stream
`struct Msg {`	`(1) a.list_begin`
` struct Tuple {`	`(2) a[0].array_length`
`  uint32_t x;`	`(3) a[0][0].x  (4) a[0][0].y`
`  uint64_t y;`	`(5) a[0][1].x  (6) a[0][1].y`
` };`	`(7) a[0].array_end`
` List<Array<Tuple>> a;`	`(8) a.list_end`
` uint8_t b;`	`(9) b`
`};`	

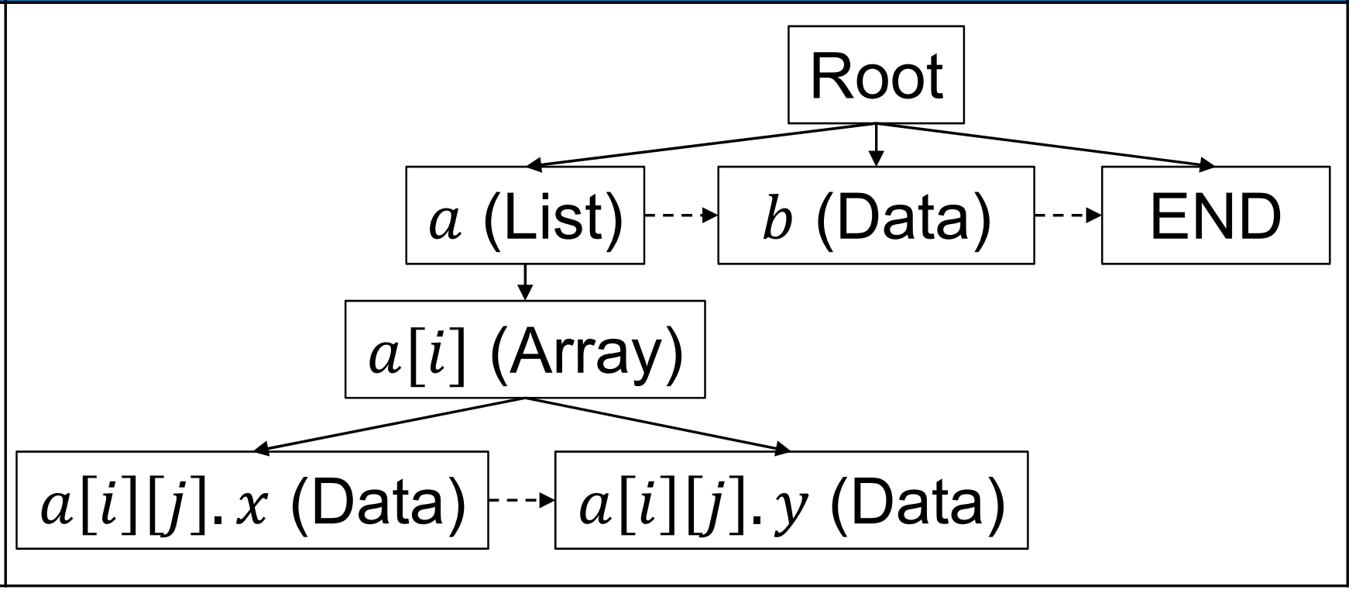
- Specify message schema and token tags in JSON

Schema in C++	Schema in JSON	Token Tags in JSON
`struct Msg {`	`{"Msg":[`	`{"/a/start":"1",`
` struct Tuple {`	`  ["a",["List",["Array",`	` "/a/elem/start":"2",`
`  uint32_t x;`	`   ["Struct","Tuple"]]]],`	` "/a/elem/elem/x":"3",`
`  uint64_t y;`	`  ["b",["Bytes",1]]],`	` "/a/elem/elem/y":"4",`
` };`	` "Tuple":[`	` "/a/elem/end":"5",`
` List<Array<Tuple>> a;`	`   ["x",["Bytes",4]],`	` "/a/end":"6",`
` uint8_t b;`	`   ["y",["Bytes",8]]] }`	` "/b":"7"`
`};`		`}`

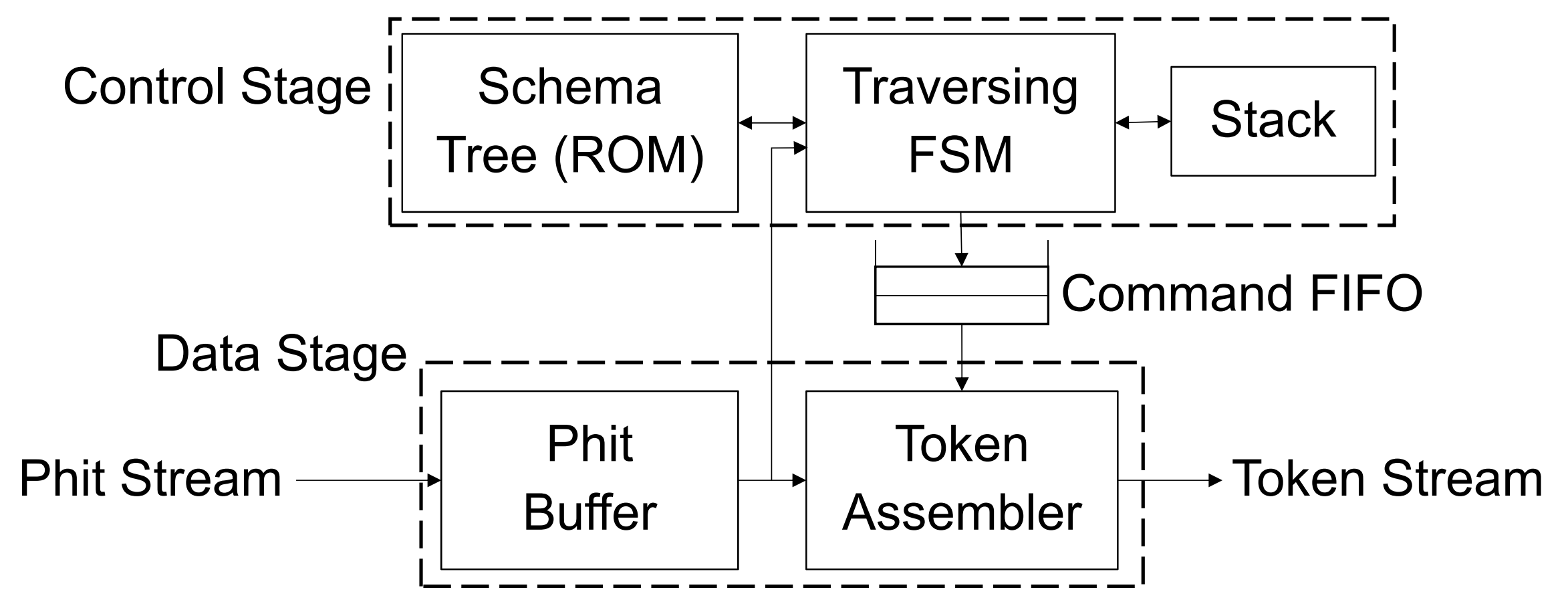
SW-to-HW Messaging

- SW serialization: the whole message is buffered
  - Fixed-length data: directly encode it to binary
  - Structure: encode each field in order
  - Array/List: first encode the number of elements, then encode all elements in order
- HW deserialization: stream processing
  - X Option 1: generate an FSM based on schema
    - -- Number of FSM states = number of fields in message
  - ✓ Option 2: algorithmically traverse schema to generate tokens
    - -- The number of FSM states is fixed, irrelevant to schema
- HW deserialization by algorithmically traversing schema
  - Schema can be represented by a tree (Structures are inlined)
    - -- Fixed length data: leaf nodes
    - -- Array/List: internal nodes
  - Outputting tokens is similar to a pre-order traversal of the tree
    - -- Subtree of an internal node needs to be traversed multiple times (= the number of elements in the Array/List)
  - Tree traversal can be done using a stack and a fixed FSM

Schema in C++	Schema Tree

`struct Msg {`
` struct Tuple {`
`  uint32_t x;`
`  uint64_t y;`
` };`
` List<Array<Tuple>> a;`
` uint8_t b;`
`};`
	

- HW deserialization microarchitecture: two stage pipeline
  - Control stage: traverse schema to determine token type
  - Data stage: consume phit stream
  - Schema tree is encoded into a ROM (with user-tag info)



Other Types of Messaging

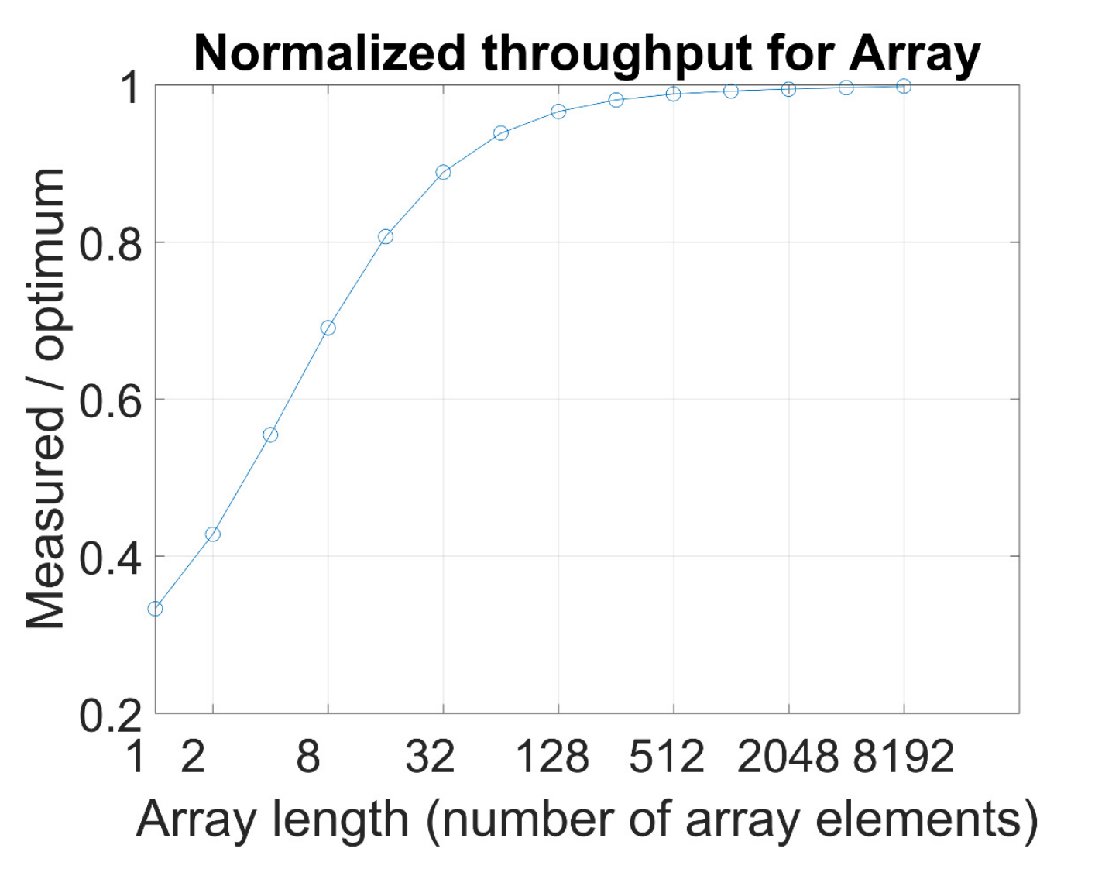
- HW-to-SW messaging: similar to SW-to-HW
  - HW serializes Array/List length after elements
  - SW deserializes from the end of the message
- HW-to-HW messaging
  - Problem: List cannot be buffered on either side
    - -- Hard to know the number of elements in a List
  - Solution: cut List into frames (frame size is bounded)
  - HW serialization
    - -- Allocate on-chip buffer to store at least one frame
    - -- Each frame contains a header (frame size, …)
    - -- Each List is ended with an empty frame
    - -- Frame header contains the level of nested Lists
  - HW deserialization: rely on frame header to determine the next step in schema traversal

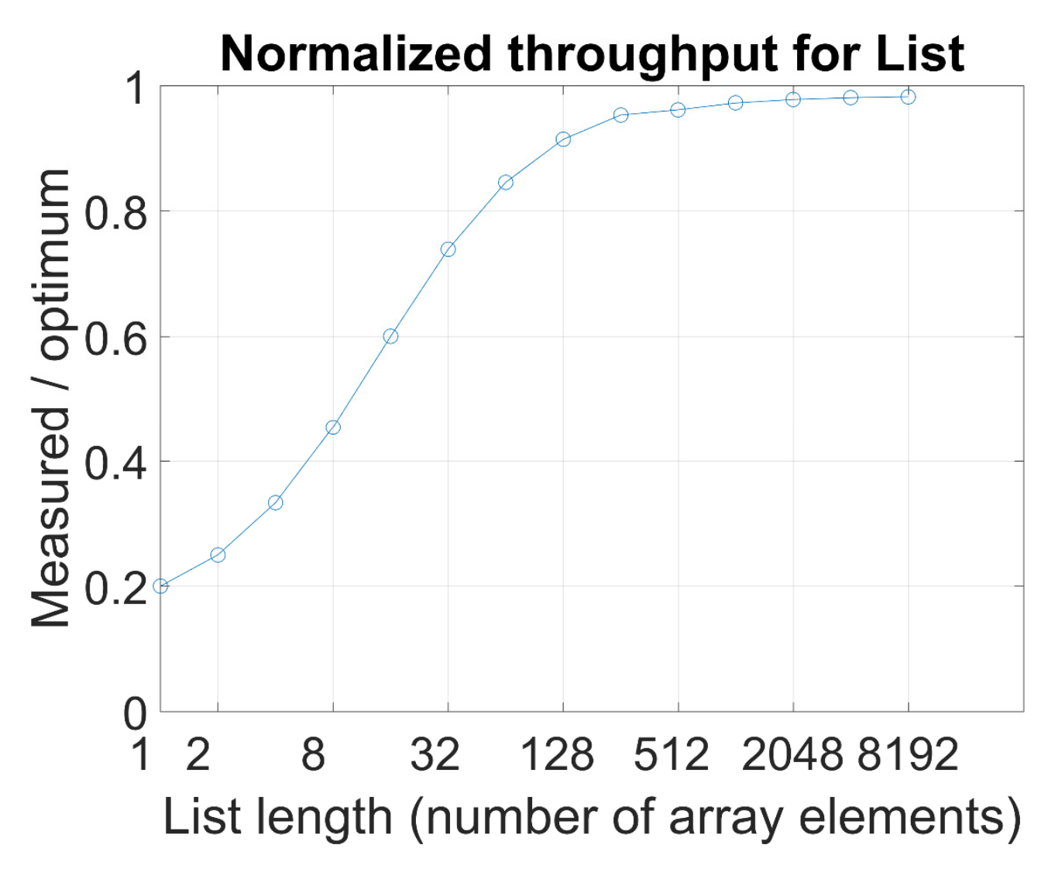
Evaluation: Clock Frequency and Area

- Complex schema with 3 levels of nested Arrays and Lists
- Combination of SW-to-HW deserialization, HW-to-SW serialization, and HW-to-HW serialization and deserialization
- Synthesized on Altera Stratix V D5 FPGA
- Clock frequency > 200MHz
- Area: 5.5% logic, and 0.2% BRAM

Evaluation: Throughput

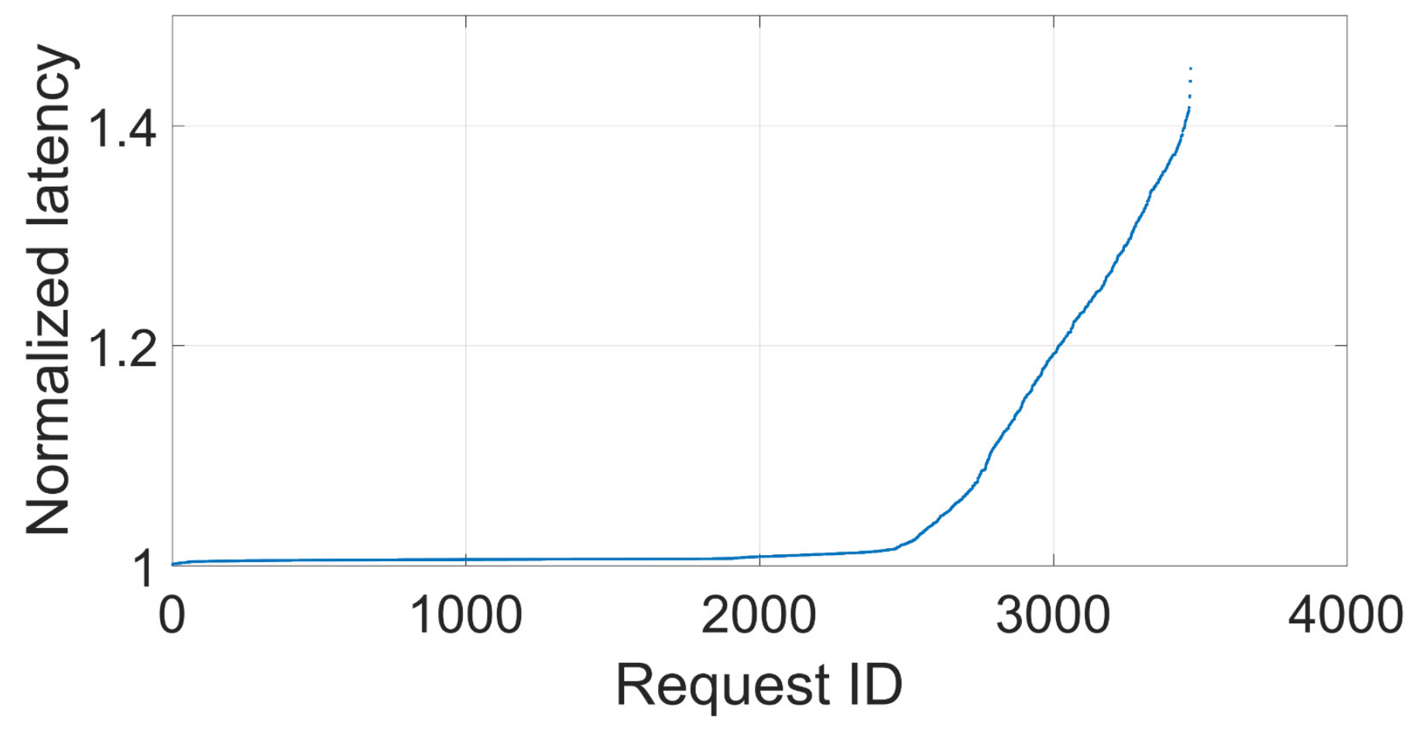
- Transfer Array and List of 128-bit element
  - SW → HW → HW → SW
  - 128-bit phit (network interface width)
  - Achieve near-optimal throughput when Array/List is long





Evaluation: Real Case Study

- Feature Extraction (FE): an important step of Bing Ranking
  - Being accelerated by Microsoft Catapult FPGA
  - Complex SW-to-HW message schema
- Ported FE to HGum framework ~ 1 man-week
- Reduce 73% hand-written code for HW deserialization
  - 27% code is for adapting tokens to FE kernel interface
- Same clock frequency, almost the same area
- Measured performance using 3468 real requests
  - FE is a blocking operation, so we measure latency
  - Geometric mean of normalized latency: 1.05


Normalized latency
Request ID