Building a Flexible Radio Stack on Colosseum for Spectrum Sharing

Tan F Wong

UNIVERSITY of FLORIDA

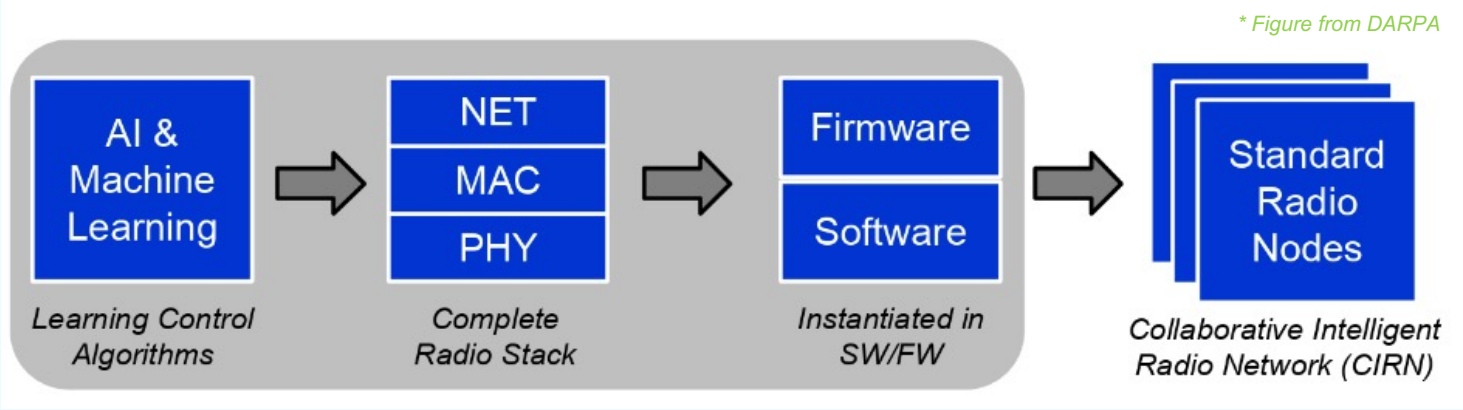
Outline

Designed and built radio stack from scratch for competition in DARPA Spectrum Collaboration Challenge (SC2)

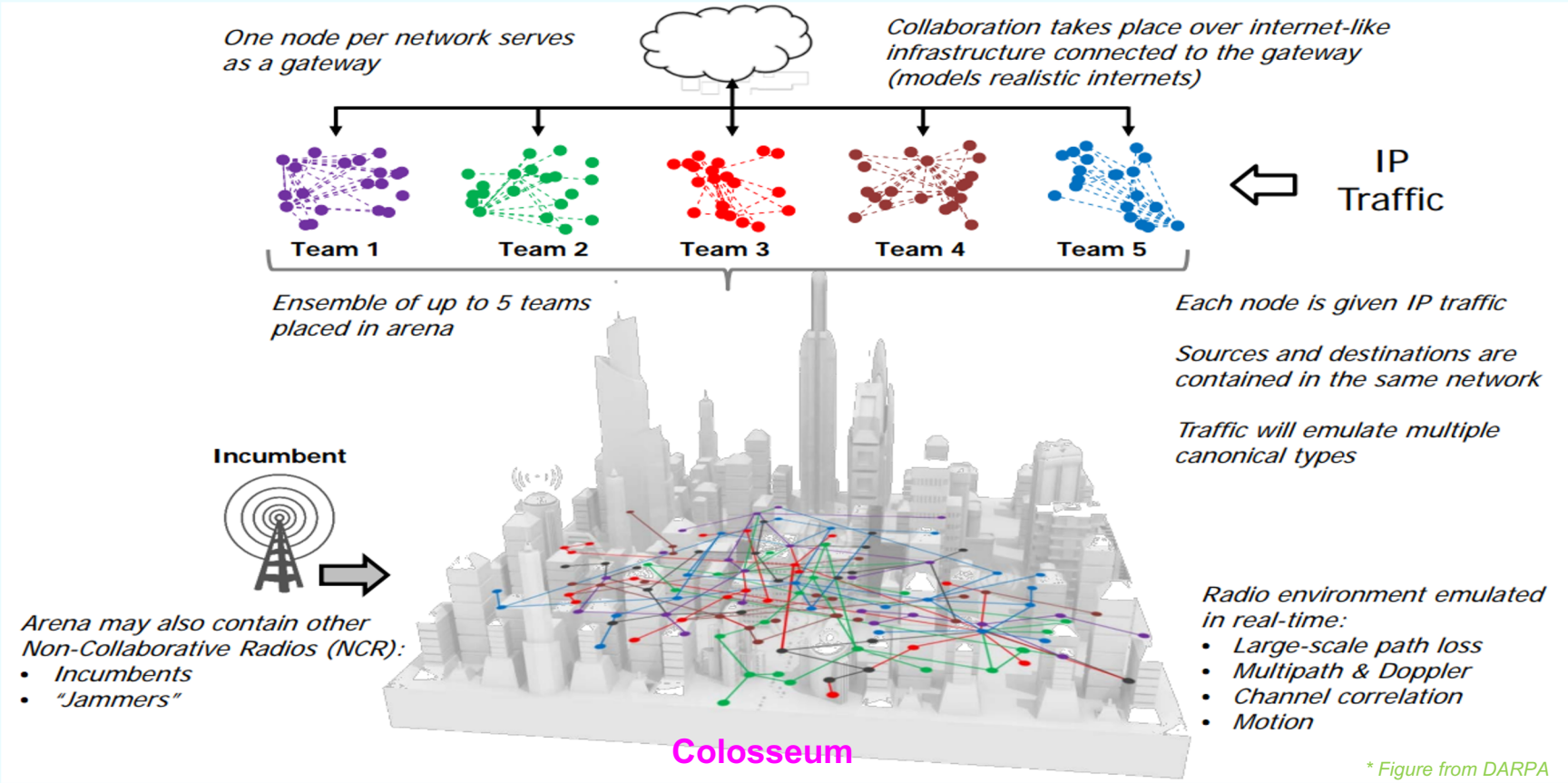
1. Motivation – SC2 design objectives
2. Radio stack details
3. Deep Q-learning experiment using radio stack
4. Our Colosseum workflow, tricks, and tools

DARPA Spectrum Collaboration Challenge (SC2)

- **Design Challenge:** Heterogeneous wireless networks *autonomously* and *collaboratively* manage and share radio spectrum in *real time* to support own communications tasks
- Competitor was to:
  - Design own radio stack and spectrum sharing algorithm to meet design challenge
  - Implement design on Colosseum to compete in SC2 matches



SC2 Match Scenario



* Figure from DARPA

SC2 Scoring Incentive

- Each team scores points by delivering IP traffic flows achieving certain QoS mandates (throughput, latency, hold time, etc.)
- Team's match score $= \begin{cases} \text{min score} & \text{if min score} \leq \text{scoring threshold} \\ \text{achieved score} & \text{if min score} > \text{scoring threshold} \end{cases}$

  where min score = minimum among all 5 teams' achieved scores
- A mixed cooperative/competitive game
- **Not just simply build most powerful radio and hog spectrum all time!**

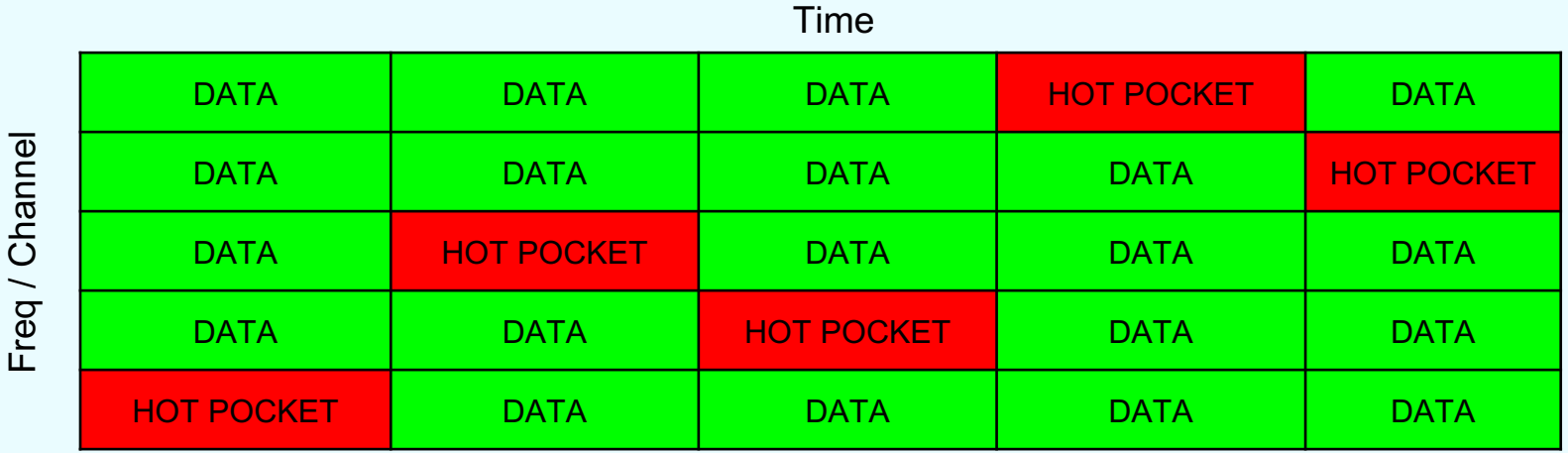
Radio Stack Design Objectives

- **Operation topology:** Small-scale ad-hoc network with 1 to 2 hops and diverse and dispersive data traffic patterns
- **Cognitive:** Real-time awareness of spectrum and spatial info, own and others' network status, *etc.*
- **Flexible/Agile:** Able to exploit opportunities in time, frequency & space
- **Robust:** Capable of operating in presence of strong interference
- **Everything adaptive:**
  - PHY: Acquisition, Modulation, Coding, TX Power, RX Gain
  - LL: Channels and Time Slots/Channel, Mapping of SRCs to Time Slots
  - NET: Supported flows, admission control granularity down to individual files/bursts
  - Other: Channels to jam

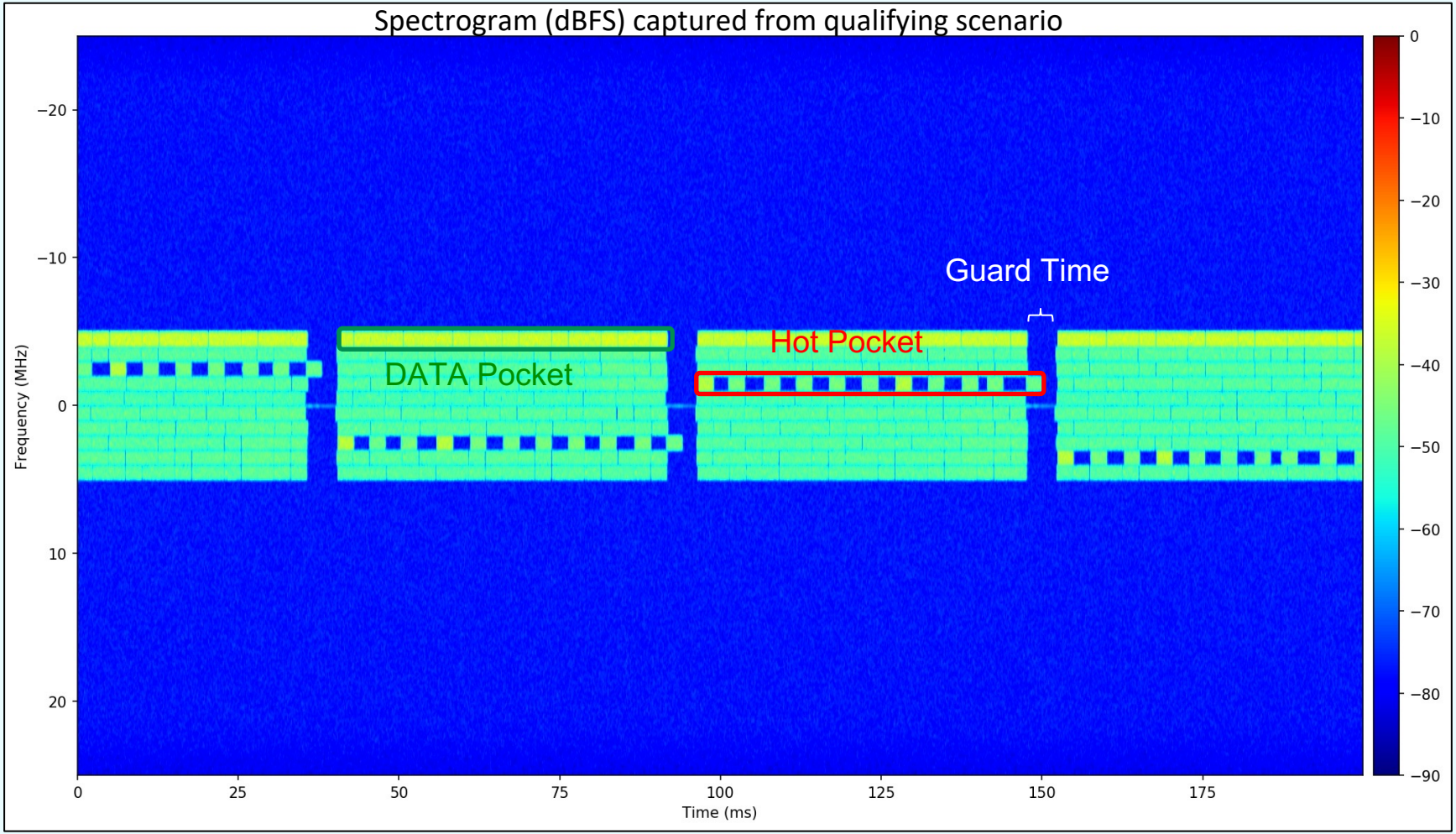
Channel Access Structure

- Radio stack design and spectrum sharing algorithm are both based on channel access structure
- Design choices:
  - Symmetric vs asymmetric
  - OFDM vs single carrier vs multicarrier
  - Control and data plane implementation
  - Synchronous vs event driven
  - Bit rate vs robustness
- Colosseum implementation constraints:
  - 2 TX & 2 RX chains
  - Compute resources: FPGA/GPU/CPU
  - SDR architecture
  - Time synchronization infrastructure

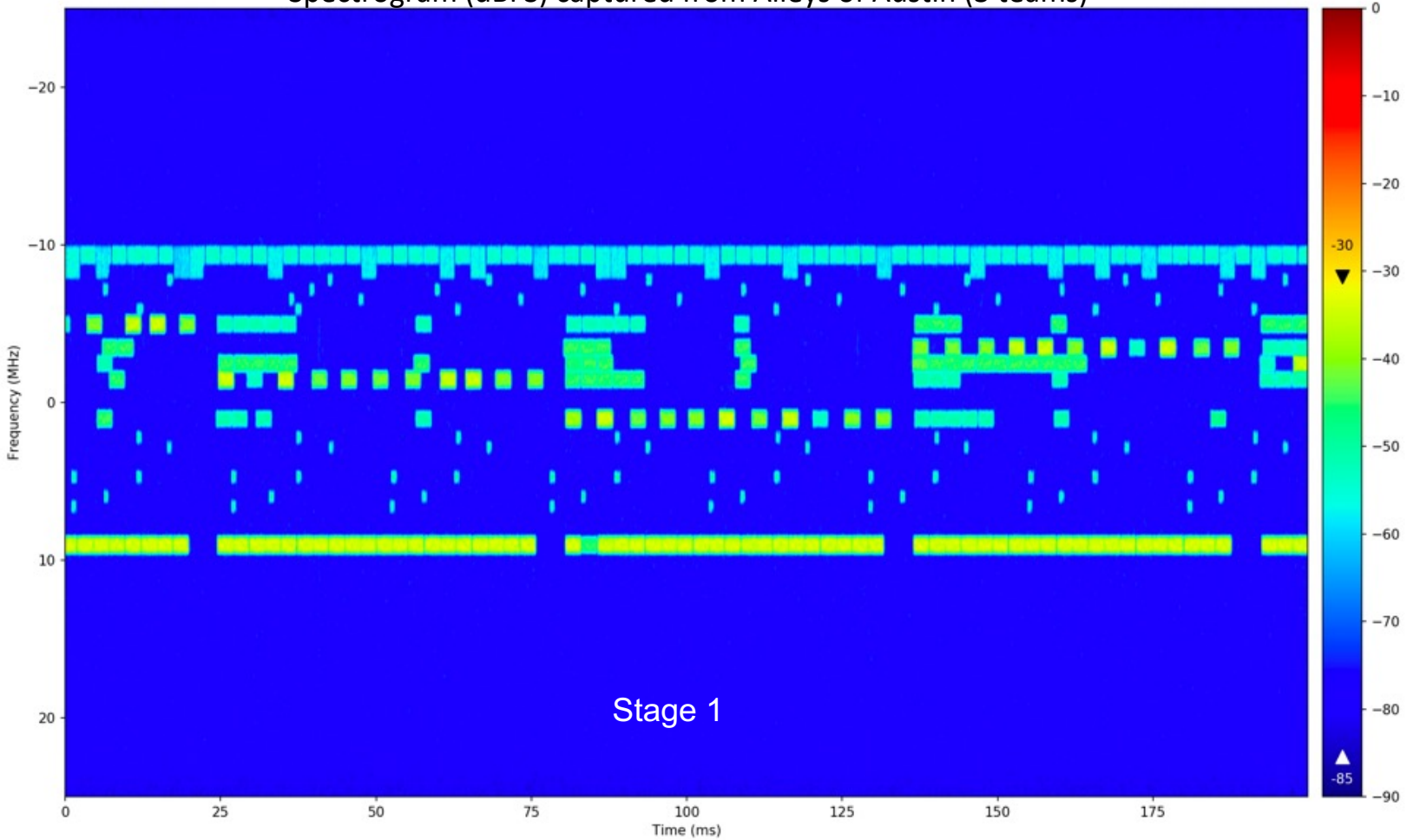
MAC: Time-Freq Pocket Structure

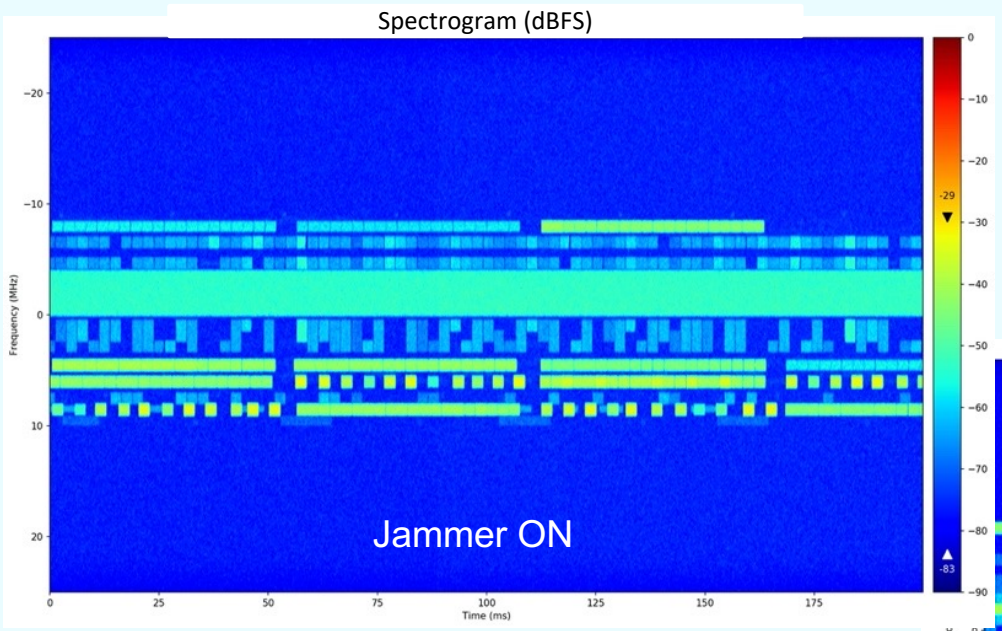


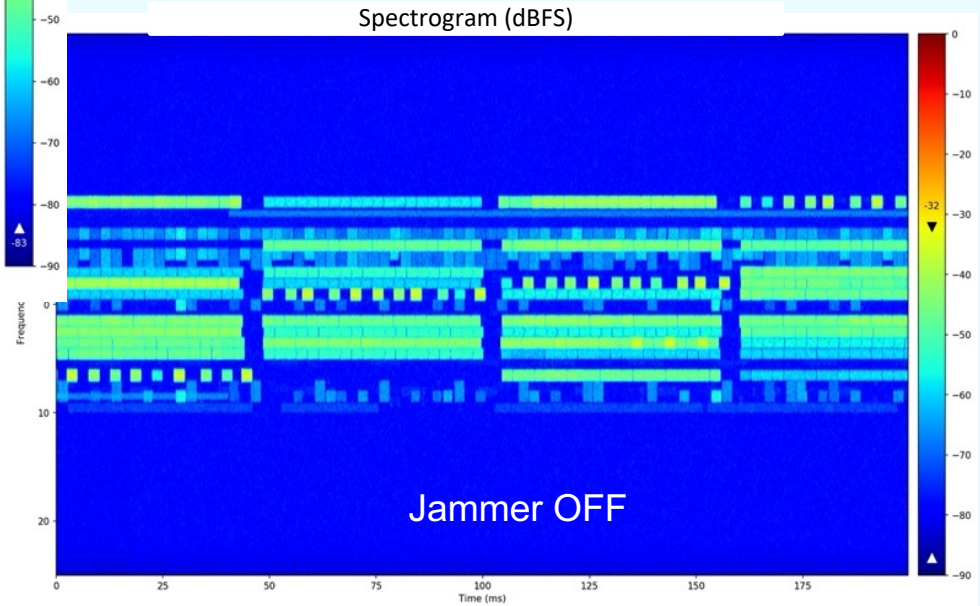
- 40MHz bandwidth channelized into 79 1MHz channels separated by 0.5MHz
- Dynamically choose a subset of non-overlapping channels to use
- Repeating frames of time slots with adjustable duration
- Each pocket (time-freq slot) can deliver **ACKed data** from one SRC to **multiple** DSTs
- Randomized "**hot pockets**" for broadcasting network management info **and ACKs**
- Packetized PHY transmission
- Each radio can simultaneously **transmit and receive on multiple channels (4 TX and 8 RX)**


Spectrogram (dBFS) captured from qualifying scenario
DATA Pocket
Hot Pocket
Guard Time
Frequency (MHz)
−20
−10
0
10
20
Time (ms)
0
25
50
75
100
125
150
175
0
−10
−20
−30
−40
−50
−60
−70
−80
−90

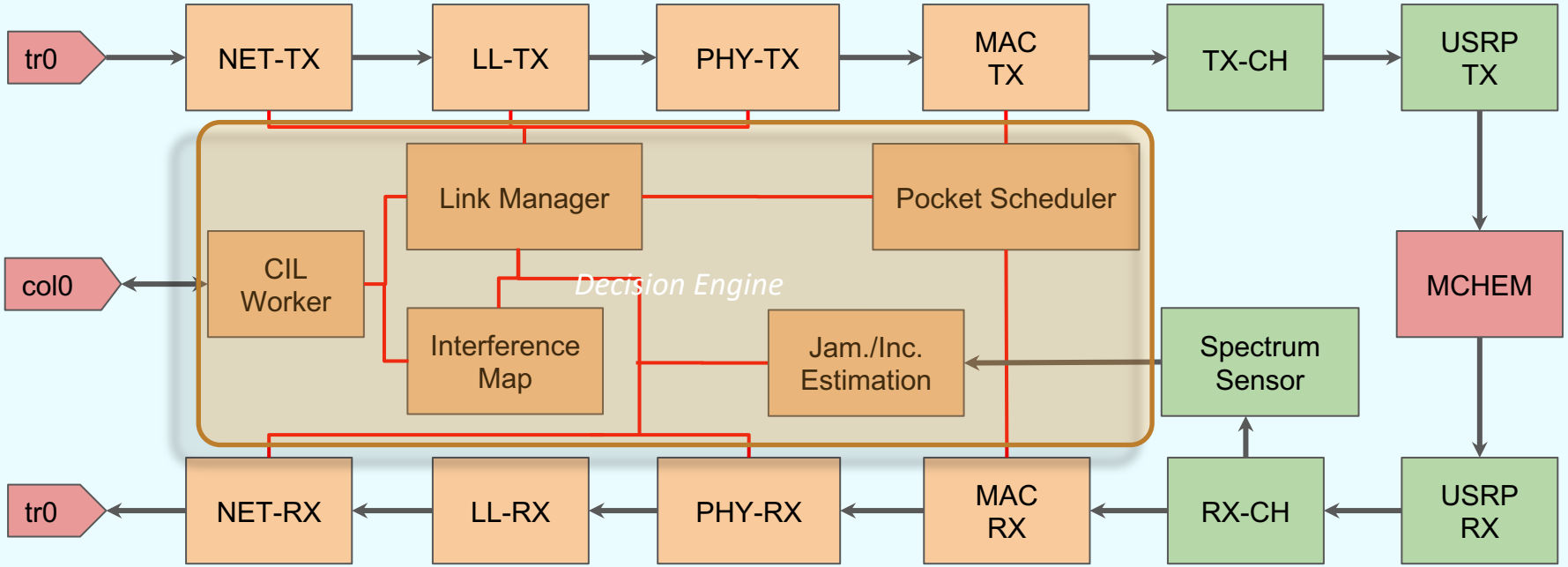
Spectrogram (dBFS) captured from Alleys of Austin (3 teams)




Spectrogram (dBFS)
Jammer ON
Frequency (MHz)
Time (ms)


Spectrogram (dBFS)
Jammer OFF
Time (ms)

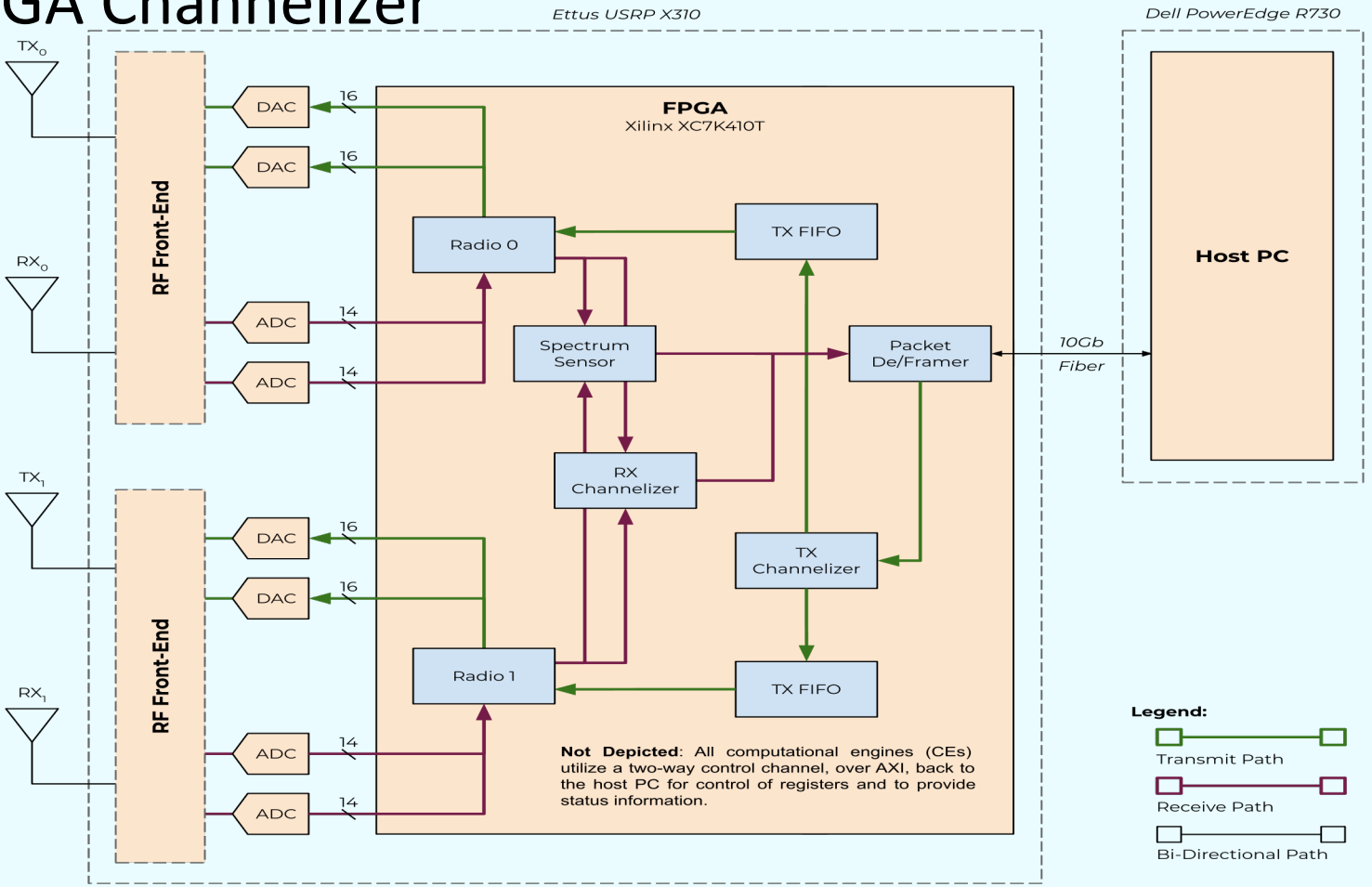
Overall Radio System



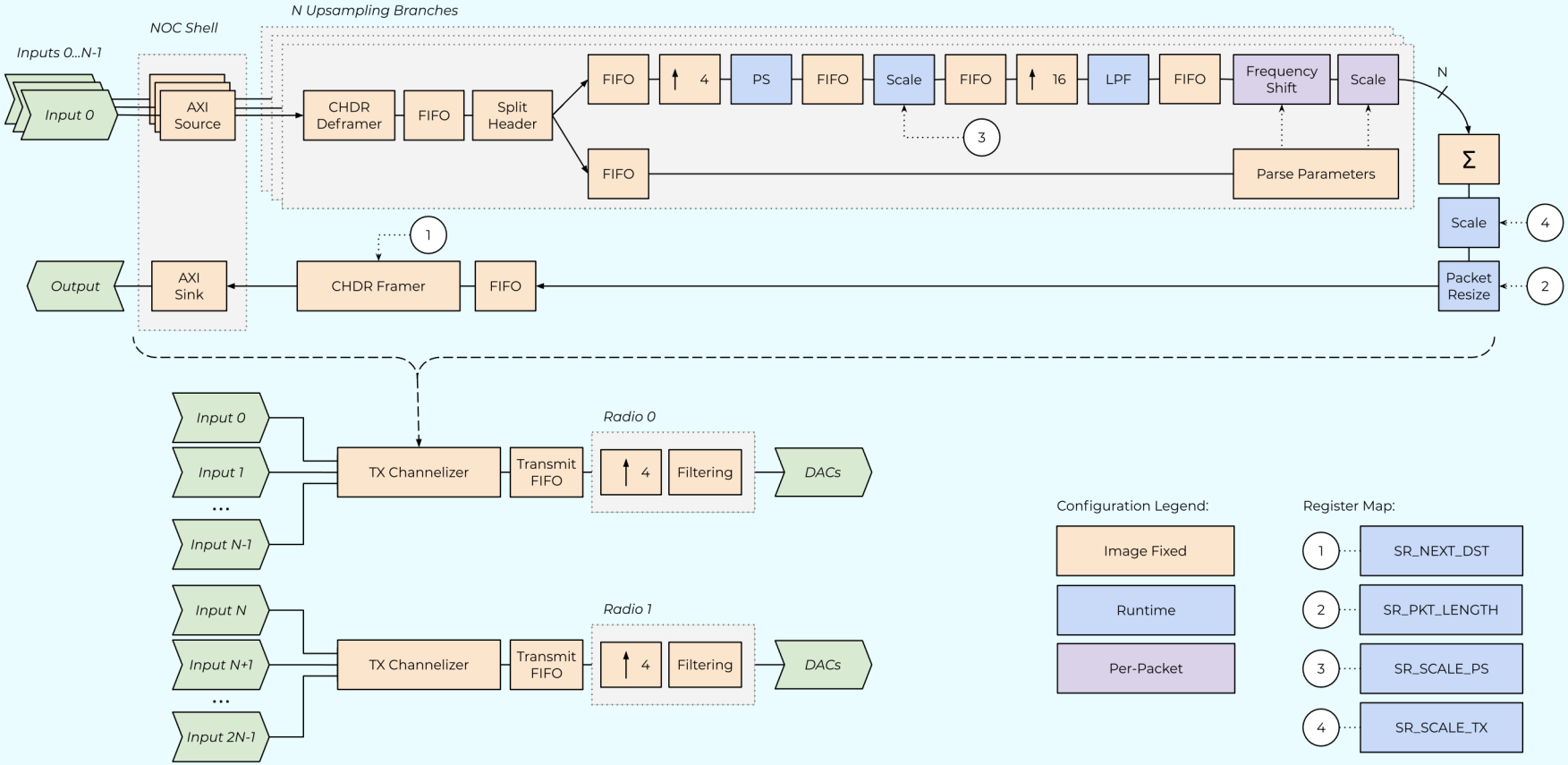
FPGA Channelizer

- Implemented multi-channel/carrier digital IF transceiver in FPGA
- Use 2 TX chains to support multi-channel/carrier transmission
  - 2 channels per TX chain → 4 simultaneous TX channels (max data rate vs PAPR)
- Use 2 RX chains to support RX beamforming
  - 8 simultaneous RX channels by digital IF
  - 1 RX channel for detecting active incumbent signals
  - Full-band FFT for spectrum sensing
  - RX signal amplitude histogram for RX gain control
- Real-time, high-rate, “fixed” TX/RX sample processing implemented in FPGA to free up CPU compute resource for higher-level radio stack functions

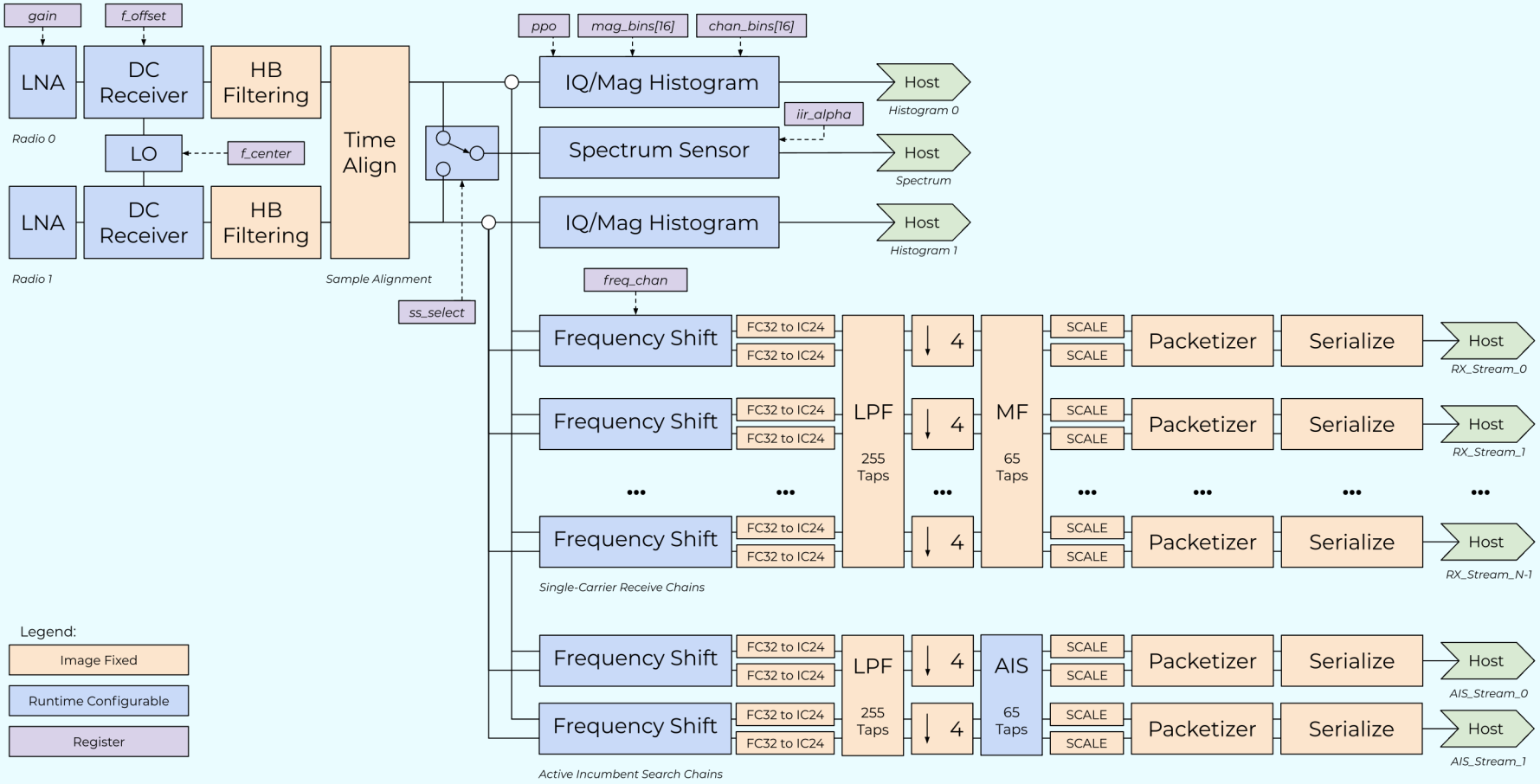
FPGA Channelizer



FPGA TX Channelizer



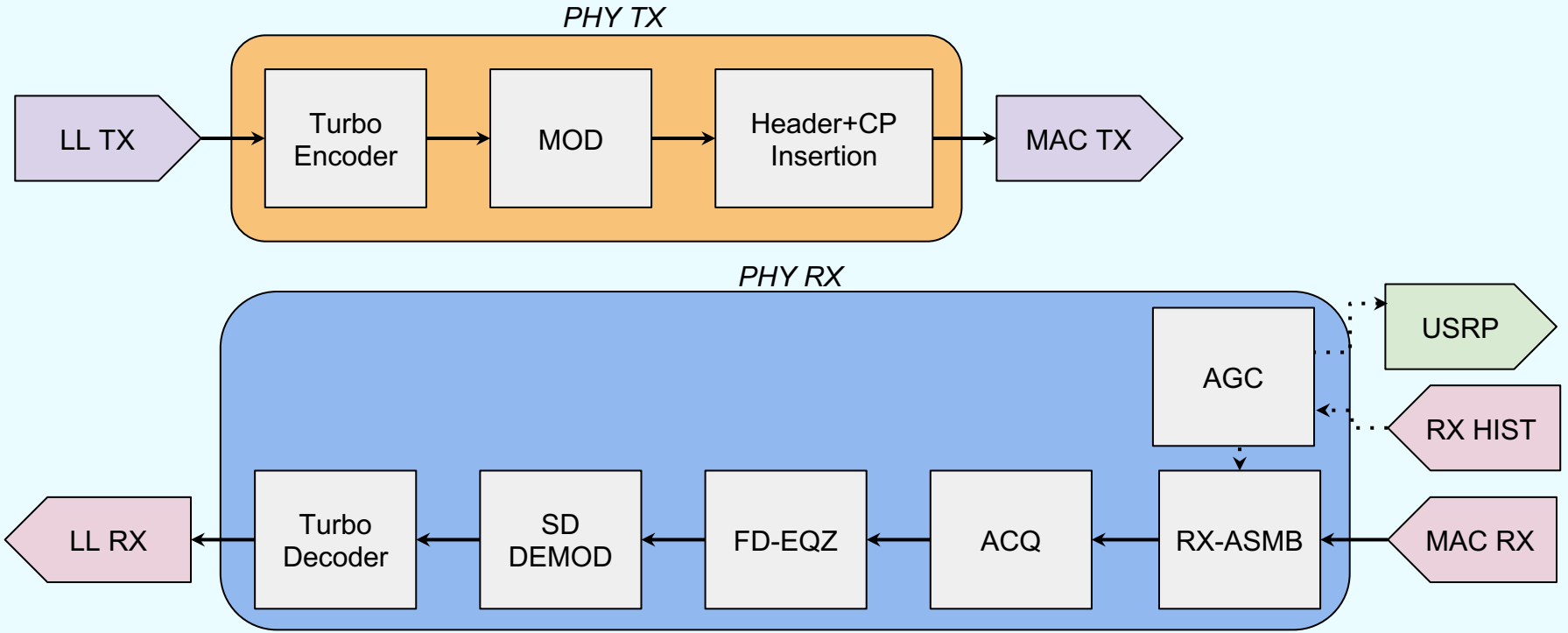
FPGA RX Channelizer



FPGA: Post-synthesis resource utilization

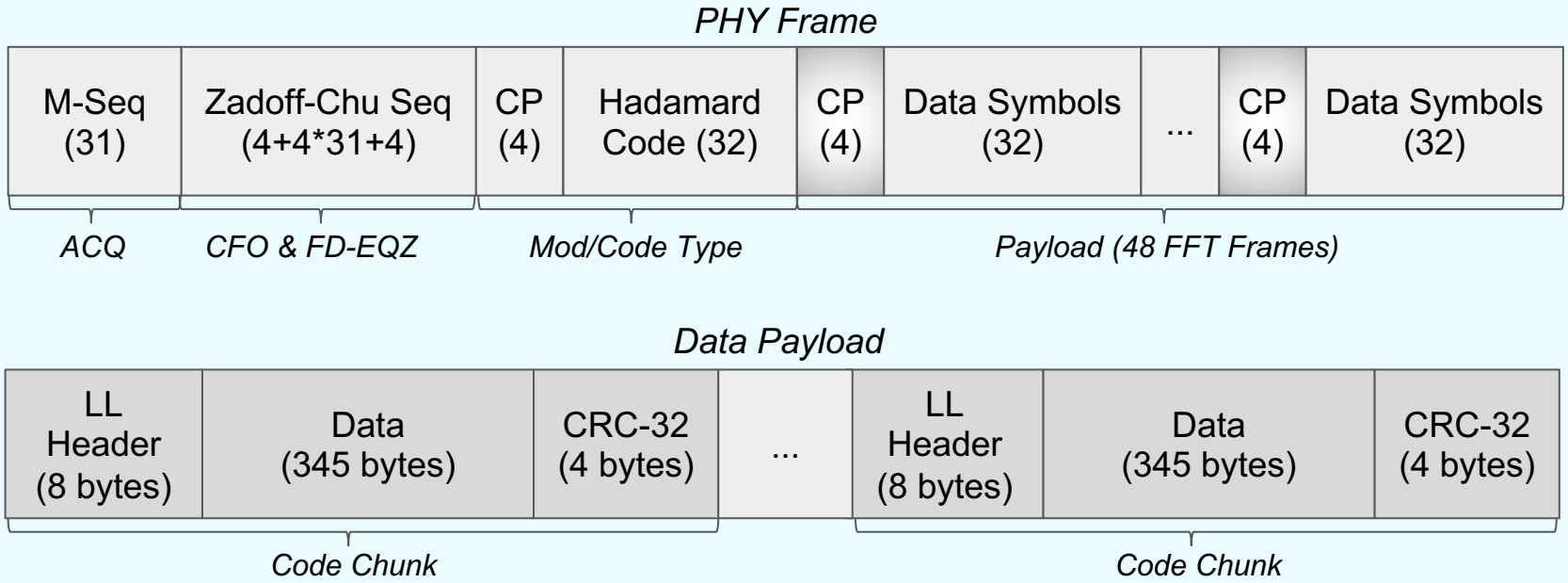
Component	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	Total RAMBs	DSP48 Blocks
Base Radio	30,204	23,340	1,680	5,184	32,930	70	116
RX Channelizer	58,751	38,653	1,712	18,386	73,052	174	483
TX Channelizer	29,372	21,396	1,712	6,264	37,096	104	224
Spectrum Sensor	9,040	6,654	464	1,922	13,354	27	76
Misc. Logic	43,369	37,717	1,198	4,454	43,099	351	3
Total Usage	170,736	127,760	6,766	36,210	199,531	726	902
Available	254,200	254,200	90,600	508,400	508,400	2,385	1,540
Remaining	83,464	126,440	83,834	472,190	308,869	1,659	638

PHY: TX & RX per channel



- Turbo code: Incremental redundancy with base rate-⅚ code, rate-½ code for robust TX
- Adaptive modulation: QPSK, 16QAM, 64QAM
- FD-EQZ: CP insertion can be turned off; FD-EQZ does CFO correction, phase sync, RX beamforming

PHY: Frame Format



- BPSK M-seq and Hadamard symbols; QPSK, 16QAM, or 64QAM data symbols
- Data payload cyclic prefix (CP) can be replaced by data symbols
- 1, 2, or 3 code chunks per data payload depending on data modulation

LL: Packet Scheduling and ARQ

- At TX:
  - Stores data packets in table and schedules according to tokens pushed to priority queue based on traffic type (TCP/UDP) and ToS
  - Creates expiration times based on QoS and removes expired packets
  - Pushes chunks of packets to fill out PHY frame based on modulation type
  - Retransmits chunks or sends incremental redundancy based on feedback from RX (or timeouts for lost packets/ACKS)
  - Generate PHY frame from control packet and push to outbound control frame queue
- At RX:
  - Tracks chunks received for each data packet
  - Tracks error probabilities for use in adaptive modulation/coding and channel selection
  - Push ARQ feedback info about which packets were completely received and which chunks are received/missing for those not complete to Link manager to be included in control packets

NET: IP Packet Handler and Link Manager

- IP Packet Handler
  - **TX:** queues incoming IP packets for LL to pull and collects incoming traffic flow info and statistics for decision engine
  - **RX:** validates blocks delivered by LL and assembles to IP packets; perform simple routing function by pushing packet back to LL for forwarding if final DST is not current node
- Link Manager
  - Implements control plane functions
  - Performs network discovery before data transmission
  - Generate control packets to push to LL and process control packets from LL
  - Control packet includes LM info such as discovery info, ACKs, link state (PER and other link quality metrics), pocket assignments, data flow info, dropped flows, SRN locations, the status of flow mandates, routing table, *etc*.
  - Different types of LM info from various parts of radio stack are queued with different priorities for insertion into control packets

Spectrum Sharing Decision Engine

- Decision engine fuses all info to adapt pocket schedule to maximize our team's match score :
  - **Which data flows are transmitted**
  - **Which channels are used and by which radios**
  - **Which flows are sent in which pockets (time-frequency resource unit)**
- Action space is huge!
  - 40 channels x 10 time slots = 400 pockets
  - As many as 100+ flows
  - 100^{400} possible pocket schedules!

Decision Engine Design

- **No ML black box that can solve spectrum sharing problem**
- Decompose problem into smaller pieces:

1. **Channel selection**
   - Choose number of channels $|C|$ to be used by our network
   - Determines particular set of channels C to use
   - RL and expert system/control/optimization approaches
2. **Admission control**
   - $|C|$ determines number of pockets available
   - Estimates number of pockets needed to support each flow
   - Iterative process to determine set of flows to admit in order to maximize points scored
3. **Pocket schedule assignment**
   - Linear program to allocate number of pockets to satisfy latency requirements of all admitted flows
   - Greedy algorithm to assign pockets in each frame to satisfy mandates of all admitted flows
   - Maps to channels in C based on worst-case SINR over links of SRC-DST pairs in above assignments

Deep Q-Learning Experiment using Radio Stack

- Trained a Deep Q-Learning agent to decide $|C|$
- Input features (28-dim) obtained from this team's own and other teams' info:

This team's metrics

$$
\begin{cases}
s_0 = (\text{our current score})/r_{\max} \\
s_1 = (\text{our current mandate threshold})/r_{\max} \\
s_2 = (\text{our current load})/r_{\max} \\
s_3 = (\text{our current bandwidth})/b_{\max}
\end{cases}
$$

Metrics for each of other 4 teams

$$
\begin{cases}
s_{4+6(i-1)} = (\text{peer } i\text{'s current score})/r_{\max} \\
s_{5+6(i-1)} = (\text{peer } i\text{'s current mandate threshold})/r_{\max} \\
s_{6+6(i-1)} = (\text{peer } i\text{'s current load})/r_{\max} \\
s_{7+6(i-1)} = (\text{peer } i\text{'s current bandwidth})/b_{\max} \\
s_{8+6(i-1)} = (\text{overlap bandwidth b/w peer } i \text{ \& us})/b_{\max} \\
s_{9+6(i-1)} = (\text{distance b/w centroids of our and peer } i\text{'s network})/d_{\max}
\end{cases}
$$

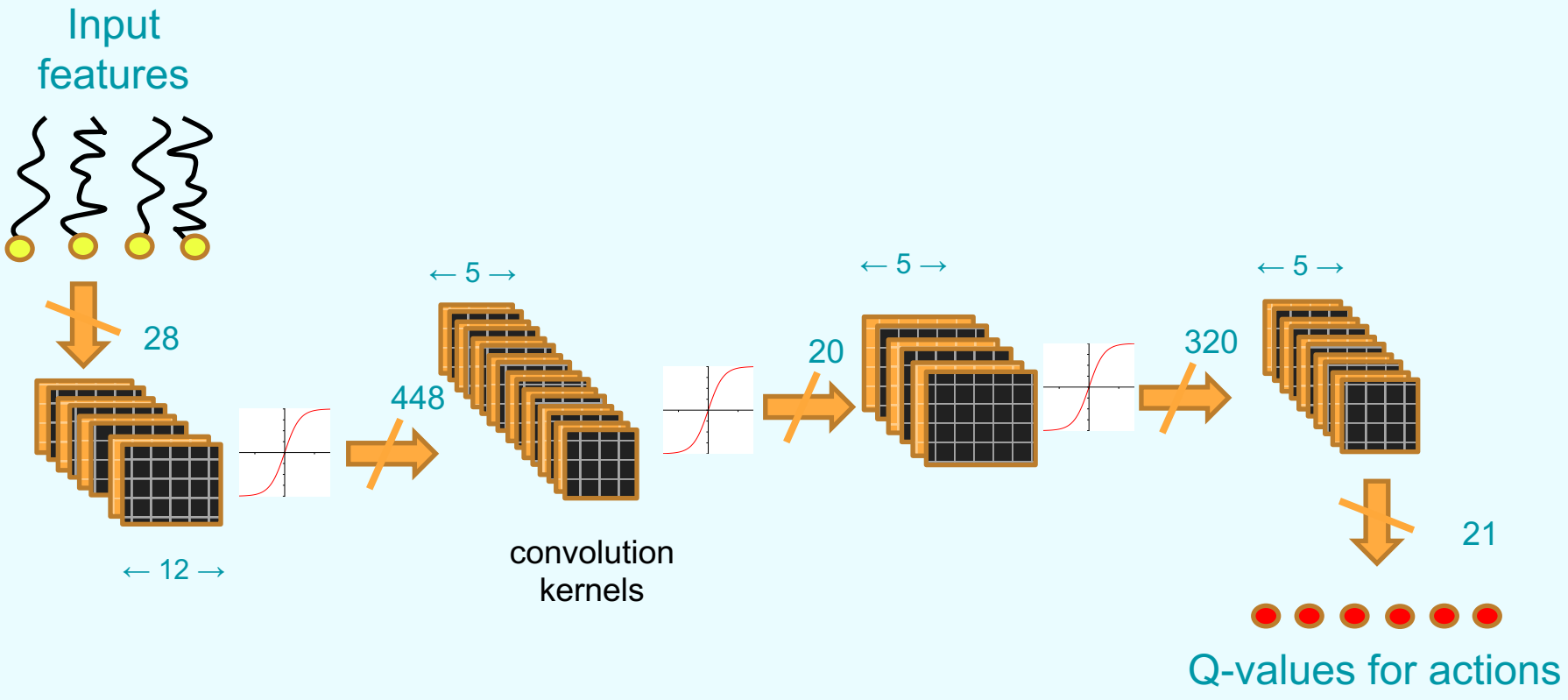
Deep Q-Learning Experiment

- Action space A is number of possible channels A = {0, 1, … , 20}
- Deep Q-network takes feature vector s to approximate Q-value Q(s, a) for each $a \in A$
- Use a convolutional neural network with three hidden layers
- Train offline with data collected from Colosseum batch jobs and use policy obtained from training for online evaluation
- Loss function accounts for QoS hold time of T :

$$\sum_{n \in T} \left(r[n+1] + \gamma \max_{a' \in A} Q(\mathbf{s}[n+1], a') - Q(\mathbf{s}[n], a[n]) \right)^2$$

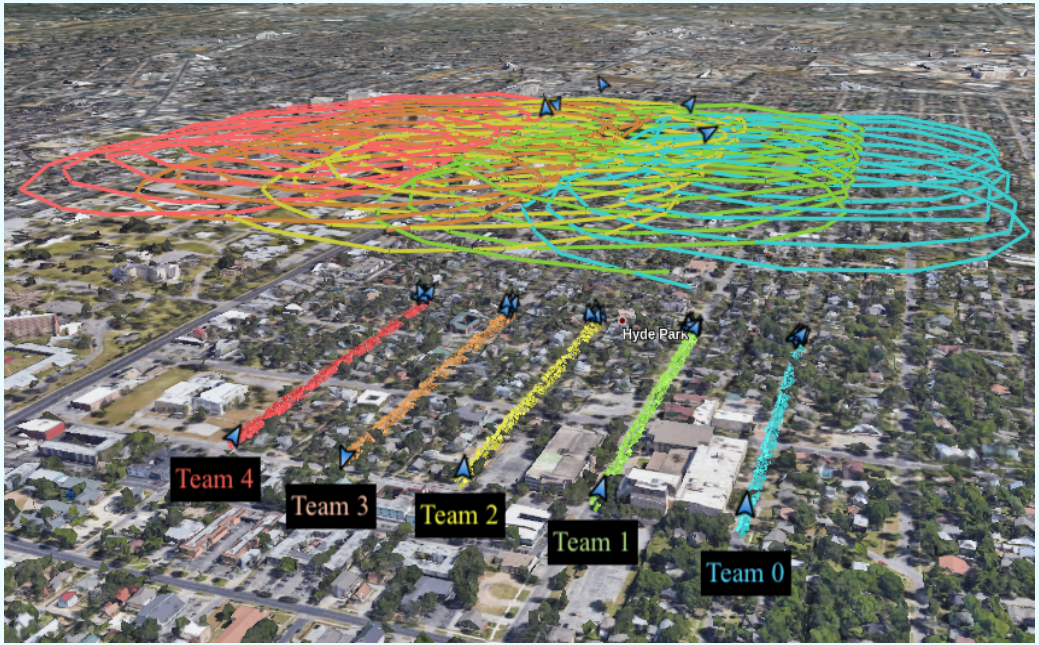
$$r[n] = \text{actual score achieved at time } n$$

Deep Q Network

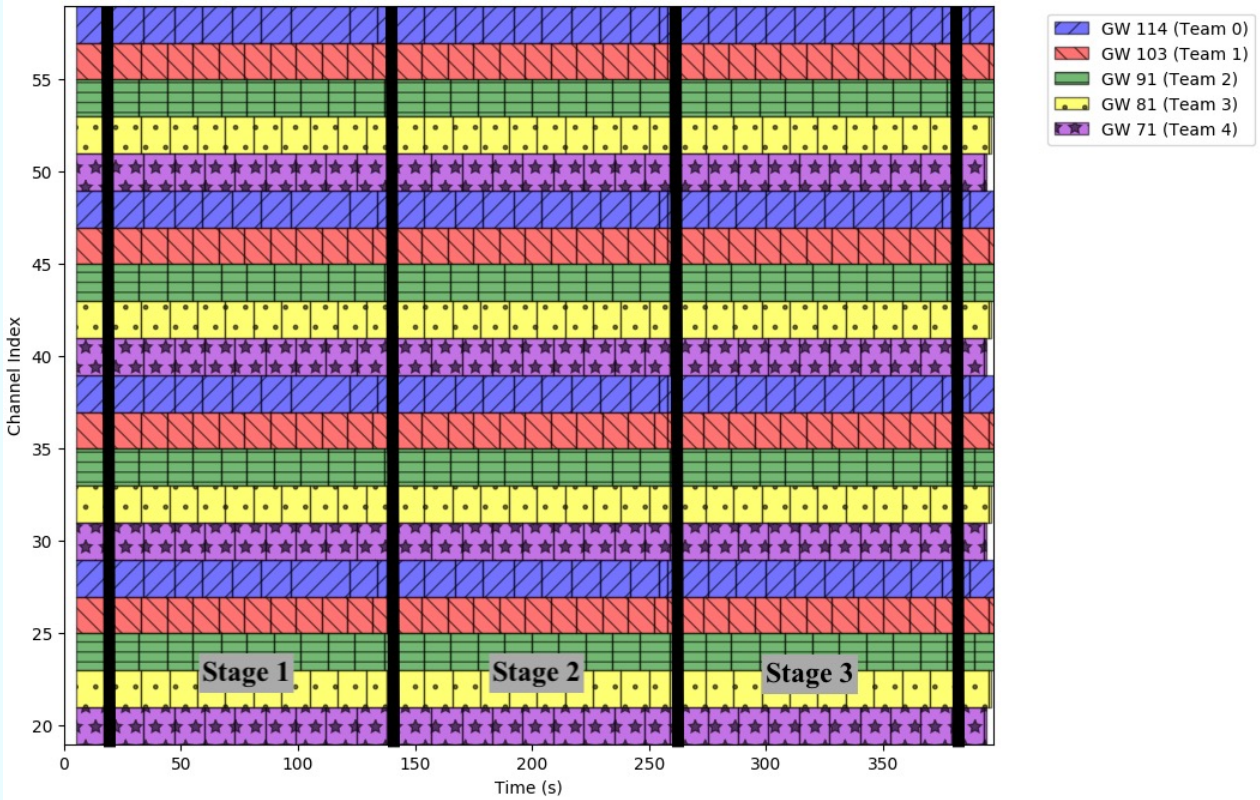


Colosseum Scenario: Alleys of Austin (10015)

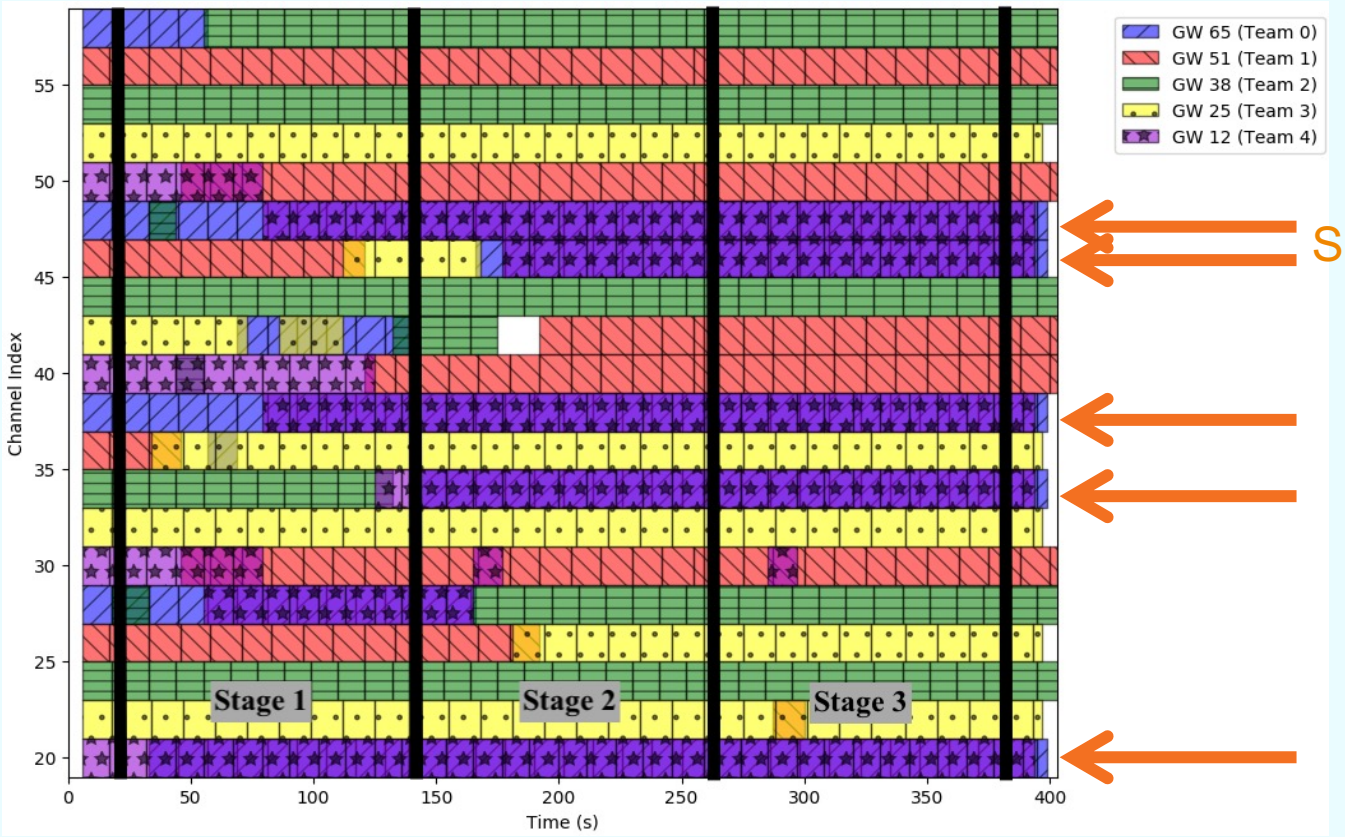
- 5 squads (teams) of 9 soldiers + UAV sharing 20 MHz spectrum
- 3 stages of increasing IP traffic demands
  - Stage 1: VOIP & C2 streams
  - Stage 2: Stage 1 + video stream & file
  - Stage 3: Stage 2 + more video streams & files



Spectrum Sharing Plan: Fair-share strategy



Spectrum Sharing Plan: Deep Q Network



Experimental Results

Fair Share Strategy

Job #	Cumulative score
1	11284
2	11428
3	10922
4	11594
5	11335

Deep Q-Network

Job #	Cumulative score
6	16833
7	17046
8	17315
9	15963
10	16599

Single Team

Job #	Cumulative score
11	26058

Experimental Results - Conclusions

- Deep Q-network able to take advantage of spatial reuse to improve scores (in comparison to fair sharing)
- 48% improvement in SC2 score
- Use of Deep RL still limited by many factors:
  - Availability of different scenarios & radio designs
  - Ability to generate training data: each Colosseum job takes around ½ hour if there is no queueing delay
  - Training time and sensitivity to parameter selection

Our Colosseum Development Workflow

1. Development and testing of basic radio functions using local SDR pair running on local LXCs:
   - Most of FPGA and PHY functions
   - Point-to-point MAC and LL functions
2. Development and testing of network functions using Colosseum batch jobs:
   - Testing “advanced” PHY functions requiring MCHEM channel emulation
   - MAC, LL, and NET functions
   - Interface to Colosseum: traffic flow handler and mandate info processing
   - Decision engine functions
3. Development and testing of multi-team functions using Colosseum batch/freeplay jobs:
   - CIL functions
   - Decision engine: interference map, jammer and active incumbent detection, jamming

Colosseum Development Experience

- More convenient to use local SDRs to develop low level radio functions
- Did almost all development and testing of higher level radio functions using Colosseum batch jobs:
  - Hard to manage 10s of nodes, MCHEM, MGEN, CIL server, *etc.* in interactive mode. All are automated in batch jobs!
  - Must use freeplay jobs to test with other teams' radios
  - **Batch jobs allow asynchronous development**
- Needed and thus developed tools to streamline development, testing, and analysis using Colosseum batch jobs

Batch Job Development Tricks

- No need to update LXC image for each code update:
  - Batch job scheduler loads **radio.conf** file from LZ and puts it in LXC when starting batch job
  - Seems no limit on file size of **radio.conf** (we tested 100MB+)
  - Create tar ball of executables, parameter files, FPGA images, ML models, *etc.* and name tar ball **radio.conf**
  - Create base LXC image with startup service to untar **radio.conf** and put all files in desired places
  - For every code change, update tar ball, scp new **radio.conf** into LZ, start batch job with base LXC image
- Log, log, and log:
  - Batch job upon termination copies all files in LXC log directory (/logs) to LZ
  - Use radio_api_scripts (batch job functionality) to move log data to LXC /logs
  - Use ramdisk if logging speed is of concern
  - Add gdb symbol table when building executable
  - Dump and save core (and executable) to /logs for offline debugging after batch job completes

Batch Job Tools

- Python script to automatically create **radio.conf** tar ball and batch job config file (**batch.json**), copy files to LZ, and submit batch job via Colosseum scheduler API
- Server scripts to keep track of batch job activities, rsync log data from LZ back to local server, and call processing scripts to perform analysis
- Python parsers and programs to plot captured spectrograms, calculate achieved throughputs and delays of traffic flows, calculate and plot match scores, plot spectrum sharing plan, … from batch job logs
- Web page to manage all above in one place: Batch Job Central

Closing Thought

- Colosseum is a wonderful experimentation tool for wireless researchers to implement, test, validate, and evaluate their ideas, from PHY to ML, with real radios and in scale
- Rather steep learning curve and very significant development effort
- But reward could be high!

Thank you!

Acknowledgements

John Shea

Team co-Lead
Link Layer
Decision Engine

David Greene

FPGA
Network Layer
IDE & Tools

Tyler Ward

Physical Layer
Link Layer
Control Plane

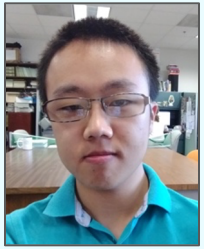
Marco Menendez[1]

CIL Development
Workflow
Optimization

Caleb Bowyer[1]

Machine
Learning

[1]Phase 2&3



Shiming Deng[2]

Analysis &
Visualization
Tools

[2]Phase 1&2

Quan Pham[2]

Channel
Emulation

Josh Agarth[3]

CIL
Development

[3]Phase 1

- DARPA SC2 Prizes and NSF EAGER Grant 1738065 for supporting our team's efforts