Network-Level Design of IoT and Edge Computing Systems

Andreas Gerstlauer

System-Level Architecture and Modeling (SLAM) Group

Electrical and Computer Engineering

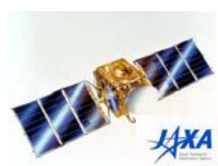
University of Texas at Austin

`http://www.slam.utexas.edu`

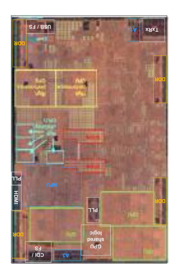
The University of Texas at Austin
Electrical and Computer Engineering
Cockrell School of Engineering

Traditional Embedded Systems

- **System-in-a-system**
  - Application-specific computing
    - Not general purpose
    - Known a priori
  - Tightly constrained
    - Guaranteed, not best effort
    - Real time/performance, power, cost, reliability, security, …
  - ➢ Opportunity & need to optimize

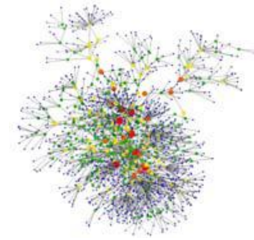


- **Ubiquitous & complex**
  - Far bigger than general-purpose computing
    - 98% of all processors sold [Turley02, embedded.com]
  - Growing complexities
    - Application demands & technological advances
    - Multi-Processor Systems-on-Chip (MPSoCs)
  - ➢ Design automation



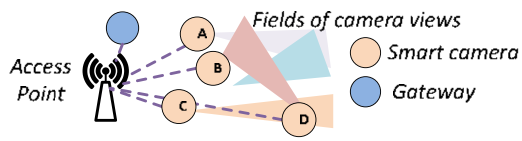
Embedded System Trends

- **Increasingly networked**
  - Application-specific
  - Resource-constrained
  - Heterogeneous
  - Distributed



➢ **Cyber-physical systems (CPS)**
  - Real-time sensing & acting
  - Interact with physical world

➢ **Internet-of-things (IoT)**
  - Cloud computing
  - Edge computing at/near sink/source
  - Open public networks



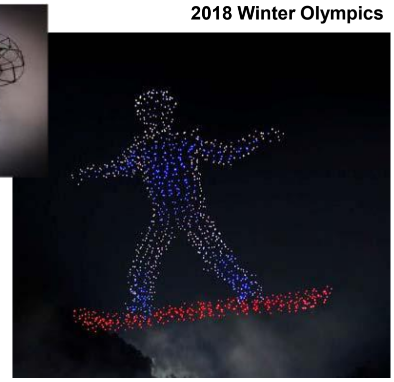
Cyber-Physical Systems (CPS)

- **Drone swarms**
  - Networked drone swarms
  - Flocking or swarming behaviors, producing images

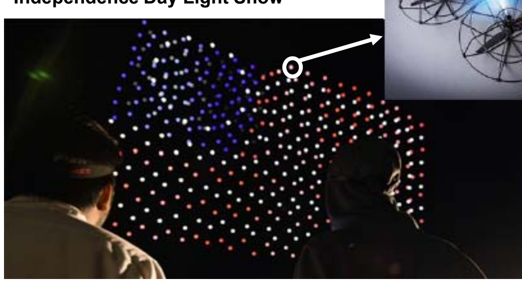
  ➢ Collaborative motion control and path planning

Intel Shooting Star Drone

2018 Winter Olympics



Independence Day Light Show



Internet of Things (IoT)

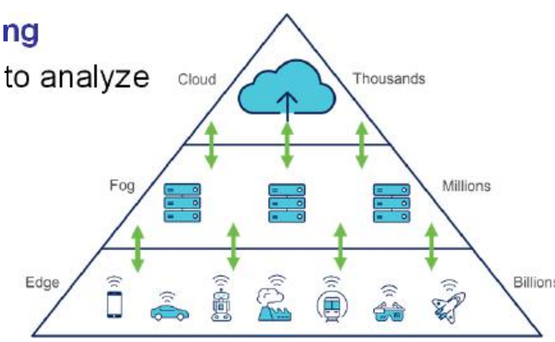
- **Smart city / smart camera networks**
  - Continuous, distributed sensor data collection and analysis
  - Traffic flow forecasting, automatic surveillance …
  - ➢ Large-scale parallel heterogeneous computing

City Brain @ Alibaba

Resource-constrained edge devices

Edge Computing

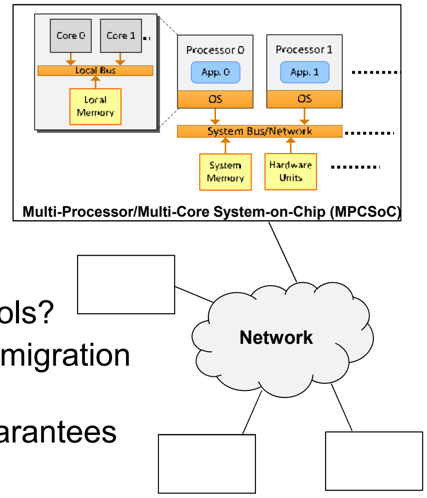
- **Traditional cloud offloading**
  - Send raw data to cloud to analyze
  - ➢ Latency/bandwidth
  - ➢ Availability
  - ➢ Privacy
  - ➢ Scalability



- **Fog and edge computing**
  - Process data (in real time) at or near the source
  - Resource-constrained platforms
  - ➢ Mobile systems
  - ➢ Sensor processing and data analytics
  - ➢ Self-driving cars and autonomous systems

New Design Challenges

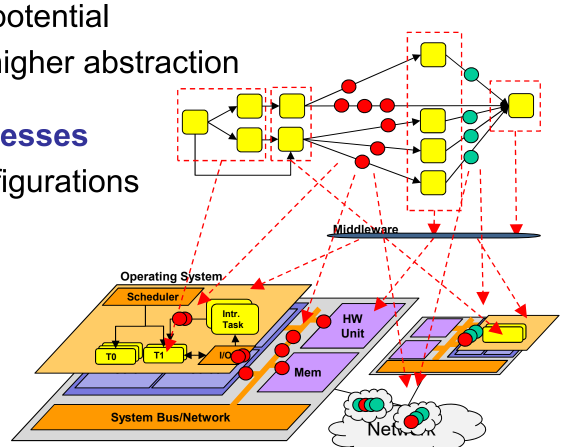
- **Networks-of-Systems (NoS)**
  - Network & system interactions
  - Computation & communication
- **Network/system co-design**
  - System device architectures?
  - Network architecture & protocols?
  - Task mapping, offloading and migration to accelerators, fog, cloud?
  - Real-time and correctness guarantees under network uncertainties?
  - Middleware?
  - Programming model?



NoS Design

- **Traditional network and system design in isolation**
  - Large co-optimization potential
  - ➢ Joint consideration at higher abstraction
- **Ad-hoc NoS design processes**
  - System & network configurations
  - Application mapping
  - Middleware & OS
  - ➢ Complex application, system and network interactions

➢ **Network-level design of NoS**
  - ➢ Systematic and automated
  - ➢ From applications to NoS implementations

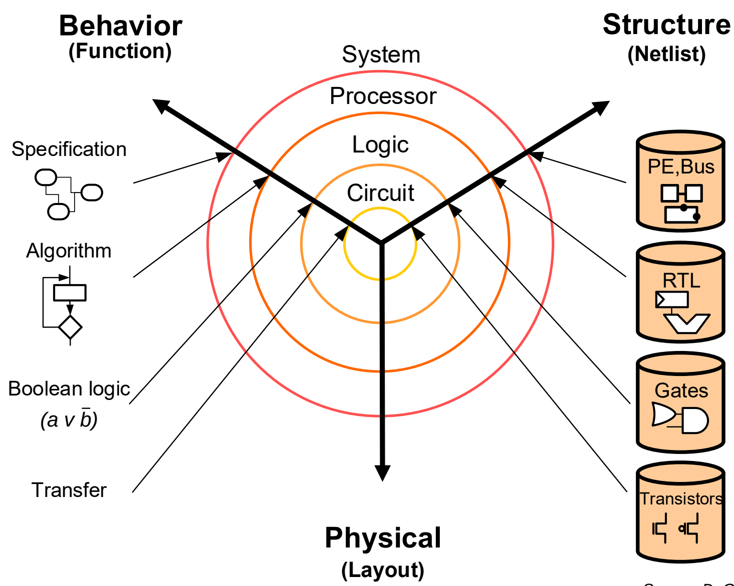


Outline

✓ **Introduction**
  ✓ Motivation, background
- **NoS Design Flow**
  - From system-level design to network-level design
- **Application Case Study: Deep Learning on the Edge**
  - Application partitioning & middleware design
- **NoS Specification and Analysis**
  - Programming model & real-time scheduling
- **NoS Simulation and Exploration**
  - Network/system co-simulation & design space exploration
- **Summary & Conclusions**

Abstraction Levels

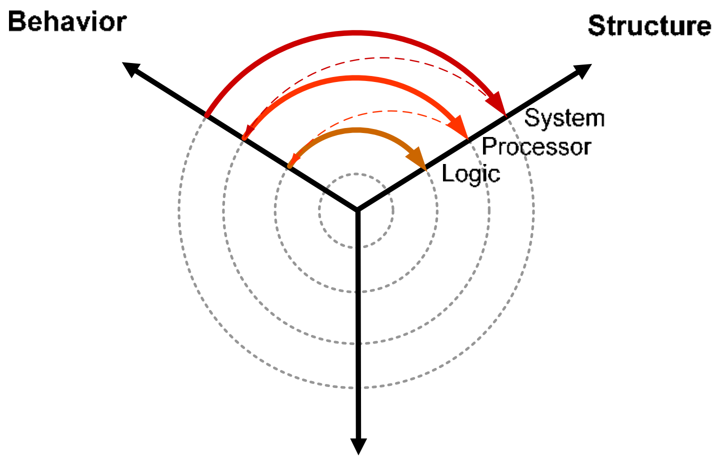
- **Gajski's Y-Chart**



Source: D. Gajski, UC Irvine

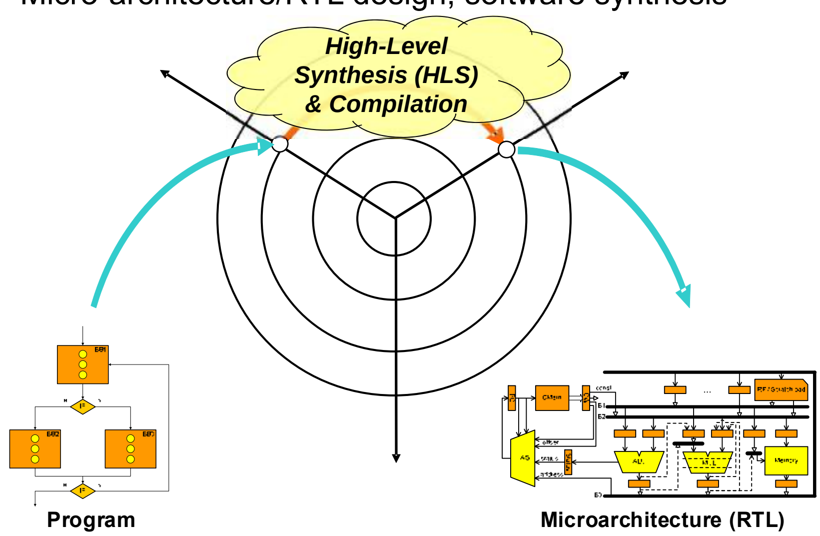
Design and Synthesis Flow

- **Gajski's Y-Chart**



Processor-Level Design

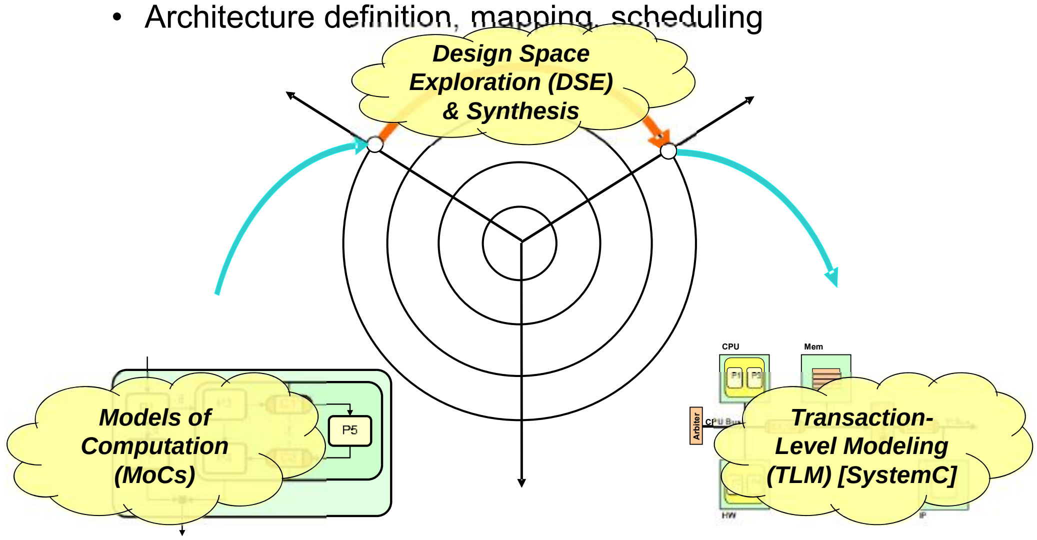
- **Hardware/software processors**
  - Micro-architecture/RTL design, software synthesis



Source: D. Gajski, UC Irvine

System-Level Design

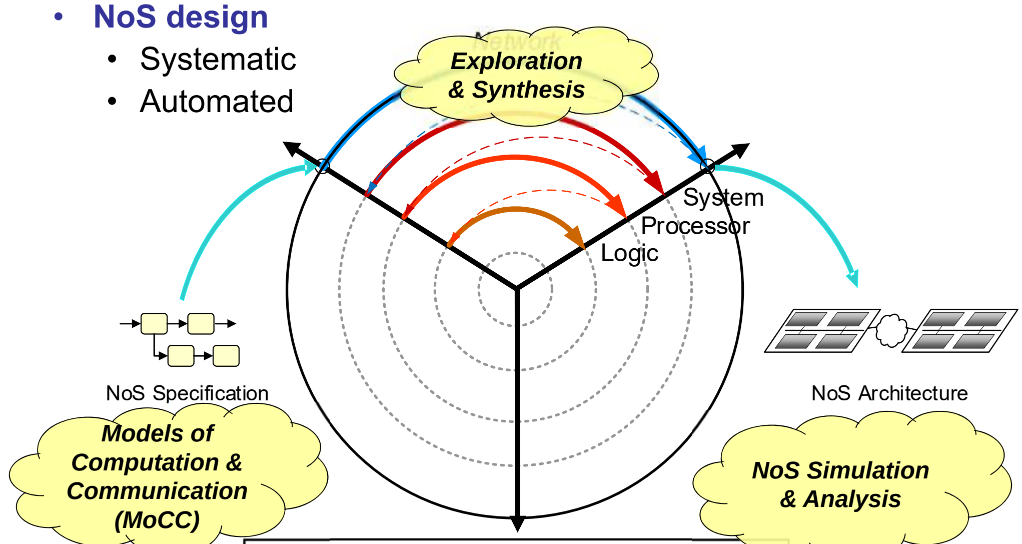
- **MPSoCs**
  - Architecture definition, mapping, scheduling



Source: D. Gajski, UC Irvine

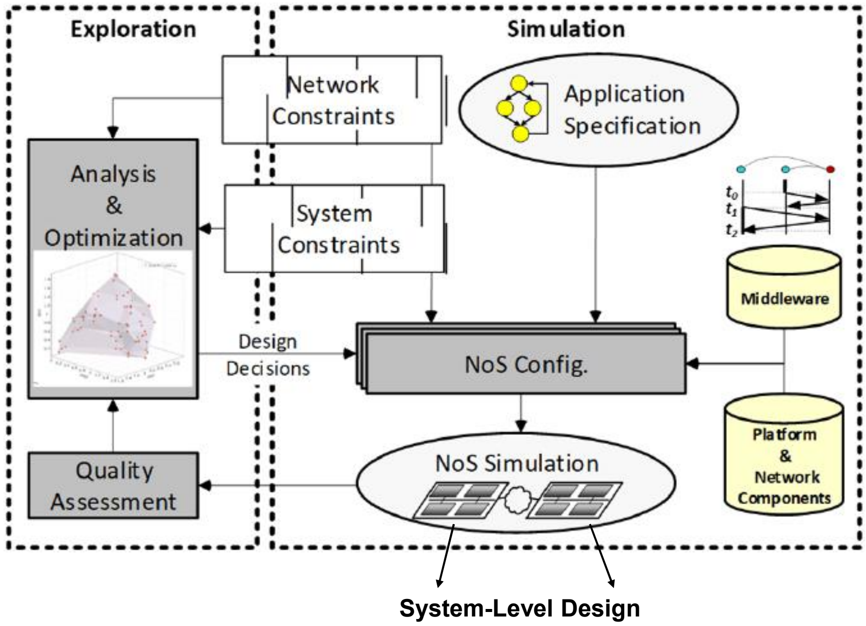
Network-Level Design

- **NoS design**
  - Systematic
  - Automated



➢ **A new level of abstraction**
  ➢ Beyond system-level design

Network-Level Design Flow



Outline

- ✓ **Introduction**
  - ✓ Motivation, background
- ✓ **NoS Design Flow**
  - ✓ From system-level design to network-level design
- **Application Case Study: Deep Learning on the Edge**
  - Application partitioning & middleware design
- **NoS Specification and Analysis**
  - Programming model & real-time scheduling
- **NoS Simulation and Exploration**
  - Network/system co-simulation & design space exploration
- **Summary & Conclusions**

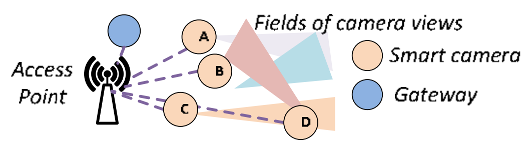
Background

- **Internet-of-Things (IoT)**
  - Complicated and noisy sensing scenarios
  - Large scale data processing & analytics
  - ➢ Deep learning (DL) techniques for IoT applications
    - – Computationally and memory-intensive
- **Edge vs. cloud computing**
  - Privacy
  - Unpredictable remote server and communication latency
  - Computational resources near the sources
    - – Edge and gateway devices

➢ **Deep learning inference in IoT edge clusters**
  - ➢ Efficient deployment on resource-constrained IoT devices

IoT Design Example

- **Smart camera network**
  - Smart city
  - Smart homes
  - …



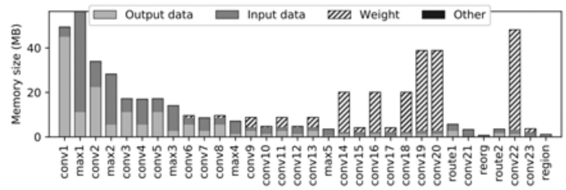
➢ **Edge computing**
  - Detect, locate and classify objects in video stream
    - – Bounding boxes
    - – Types of objects
  - Directly on cameras
    - – Privacy
    - – Real-time

Motivation

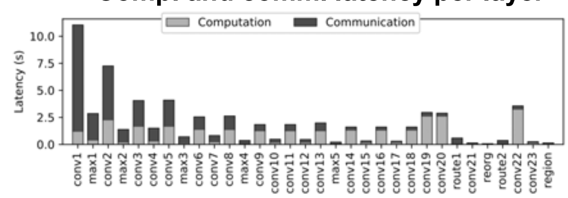
- **Deep neural network (DNN) inference resource demands**
  - Memory footprint
    - Early layers: data dominant
    - Later layers: weight dominant
  - Comm. overhead
    - Decreasing with number of layers

➢ **Distribute among edge and gateway**
  - Early layers on edge devices
  - Later layers on gateway

Memory footprint per layer*



Comp. and comm. latency per layer*



***Profiling data is collected based on the single-core performance of a Raspberry Pi 3 running YOLOv2.**

Related Work

- **Cloud-assisted inference [Kang’17, Teerapittayanon’17]**
  - ➢ Unpredictable cloud status and communication latency
  - ➢ Privacy issues and scalability
- **Model simplification**
  - Sparsification and pruning [Bhattacharya’16, Yao’17]
  - Compression [Iandola’16, Howard’17, Zhang’17]
  - ➢ Loss of accuracy and application-/scenario-dependent
- **MoDNN [Mao’17]**
  - Static partition and local distribution on mobile devices
  - MapReduce-like programming model in mobile cluster
    - Bulk-synchronous and lock-step fashion
    - Layer-by-layer synchronization
  - Limitation in scalability

DeepThings [TCAD'18]

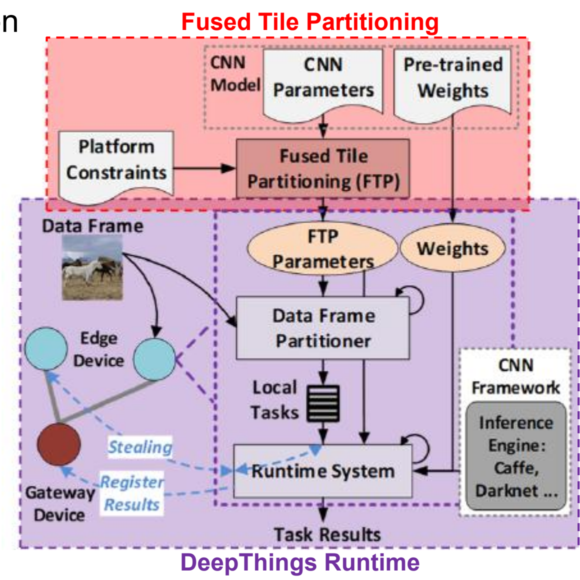
- **Scalable and flexible partitioning method**
  - Lightweight data synchronization
  - Independently distributable tasks
- **Efficient workload distribution/balancing**
  - Adaptive load balancing under dynamic IoT scenarios
  - Task distribution with minimum communication overhead

➢ **DeepThings: distributed adaptive deep learning inference on resource-constrained IoT edge clusters**
  - Fused Tile Partitioning (FTP)
  - Lightweight work stealing middleware

Source: Z. Zhao, K. Mirzazad, A. Gerstlauer, "DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters," IEEE TCAD, 2018

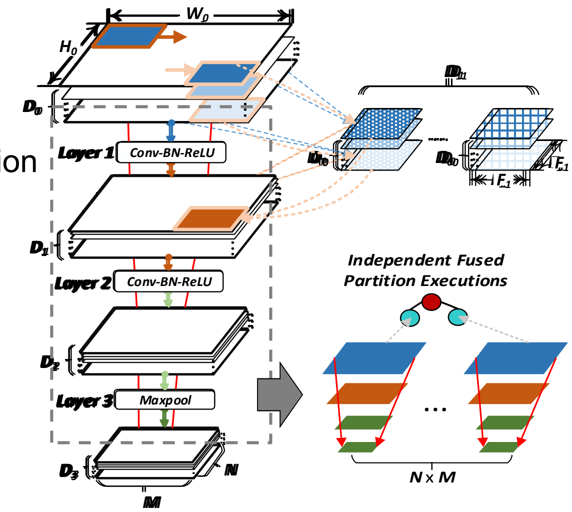
DeepThings Overview

- **Fused Tile Partitioning (FTP)**
  - Layer tiling & fusion
  - Memory & comm. constraints
  - ➢ Independently distributable tasks
- **Runtime system**
  - Adaptive task distribution and load balancing
  - Peer-to-peer work stealing
  - ➢ Collaborative inference



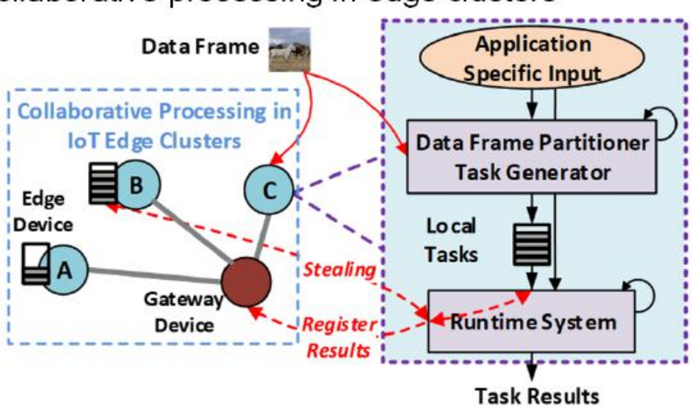
Fused Tile Partitioning (FTP)

- **Convolutional operation**
  - Local connectivity between neurons of consecutive layers
  - ➢ Grid partitioning with boundary consideration
- **Chain of multiple convolutional layers**
  - Large amount of intermediate data
  - Boundary synchronization overhead per layer
  - ➢ Layer fusion
- ➢ **Independent execution stacks**



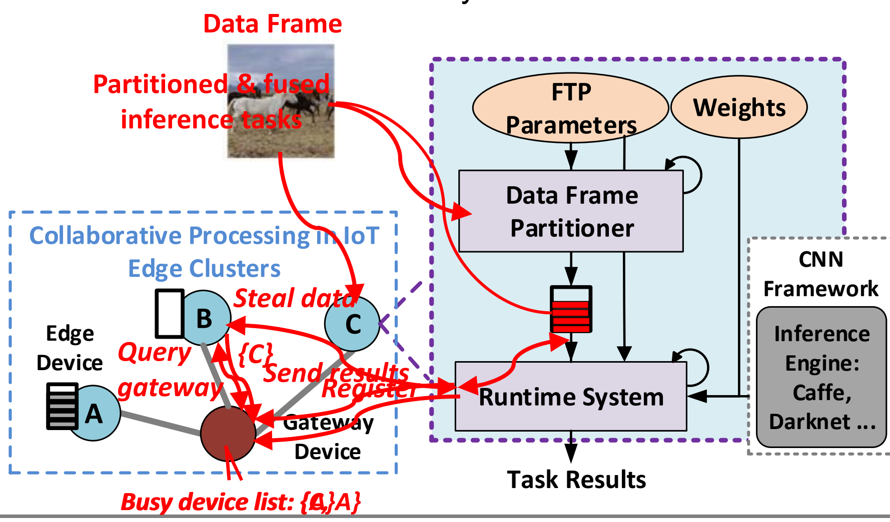
Adaptive Work Stealing Middleware

- **Distributed work stealing in local area network**
  - Lightweight data synchronization
    - Peer-to-peer data & task migration
  - Dynamic data streams and device availability
    - Gateway-managed load balancing
  - ➢ Collaborative processing in edge clusters



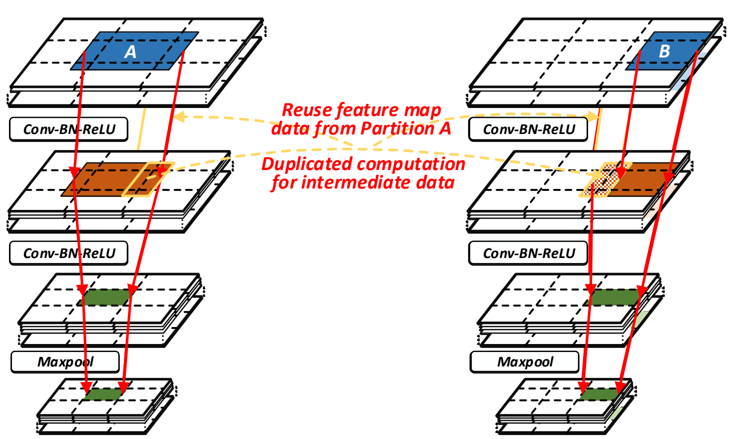
Work Stealing Approach

- **Message flow and data movement**
  - Peer-to-peer input data migration
  - Idle device steals from busy device



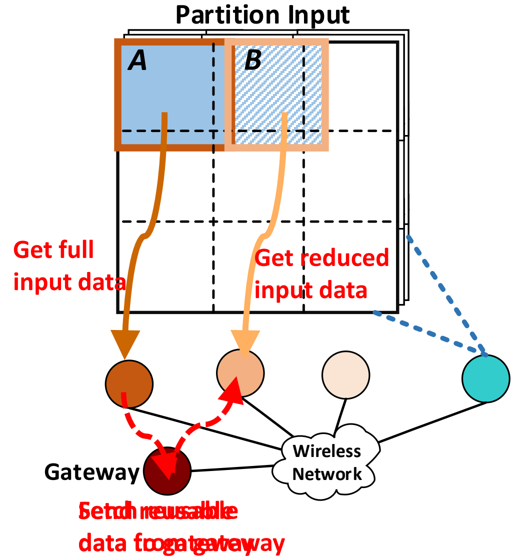
Data Reuse Opportunities

- **Redundancy in Fused Tile Partitioning**
  - Duplicate overlapped data for independent sub-tasks
  - Overlapped data amplified through many fused layers
  - ➢ Possible data reuse to reduce computation



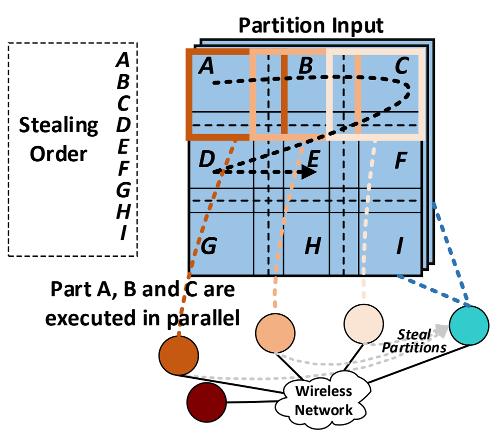
Soft Data Dependencies in DeepThings

- **Dynamically managed**
  - Processing task without reusable data
    - Get the original input region locally/from peer device
    - Send intermediate data to gateway for future reuse
  - Processing task with intermediate data reuse
    - Fetch reusable intermediate data from gateway
    - Get reduced input data locally/from peer device
- **Dependency not satisfied**
  - Fall-back execution
    - Recompute if intermediate data is not available



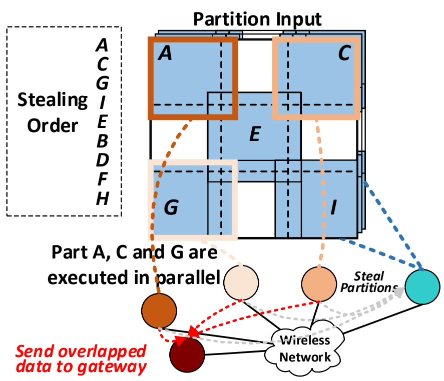
Dependency-Aware Work Scheduling

- **Soft execution dependency between adjacent tasks**
  - Data reuse depends on scheduling order
- ➢ **Data reuse-aware FTP partition scheduling**
  - Stealing task reordering and scheduling



Dependency-Aware Work Scheduling

- **FTP partition scheduling**
  - Minimize the partition dependency
    - Scheduling tasks to be stolen in dependency order
    - Caching overlapped reuse data in gateway



Experimental Setup (1)

- **DeepThings framework**
  - Retargetable implementation in C
  - TCP/IP socket APIs
  - Released in open-source form
- **Experiment platform**
  - Raspberry Pi 3 Model B
  - Up to 6 nodes in WLAN over WiFi
- **Deep leaning application**
  - You Only Look Once (YOLO) object detector on Darknet
    - First 16 layers (12 convolutional and 4 maxpool layers)
    - More than 49% of computation and 86.6% of memory footprint
  - Multiple data sources
    - Emulate dynamic application scenarios

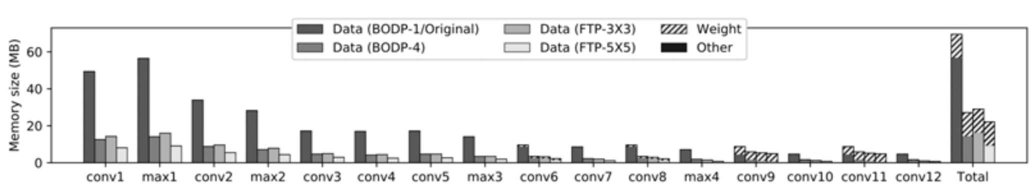
Experimental Setup (2)

- **DeepThings vs. MoDNN**
  - Work Sharing (WSH): Central data collection/coordination
  - Work Stealing (WST): Peer-to-peer data transmission
  - Data partitioning & synchronization
    - DeepThings (FTP): Overlapped data is duplicated/transmitted at input
    - MoDNN (BODP): Overlapped data is synchronized after every layer.

	DeepThings	**MoDNN**
Partition Method	Fused Tile Partitioning (FTP)	Biased One-Dimensional Partition (BODP)
Partition Dimensions	3x3 ~ 5x5	1x1 ~ 1x6
Distribution Method	Work Stealing (WST) Work Sharing (WSH)	Work Sharing (WSH)
Edge Node Number	1 ~ 6	

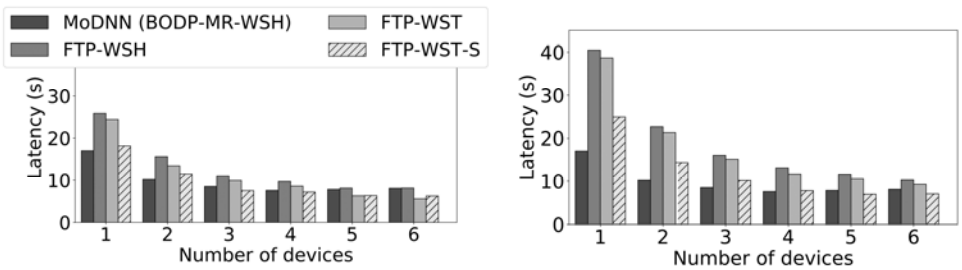
Memory Footprint

- **Per device memory footprints of each layer**
  - Memory reduction
    - Input/output feature map data is partitioned to save memory
    - Weight data is not partitioned and remains the same
  - Maximum memory usage reduction
    - 61% in 4-way BODP, 58% and 68% for FTP 3x3 and 5x5
  - Average memory footprint reduction per layer
    - 67% in 4-way BODP, 69% and 79% for FTP 3x3 and 5x5



Latency (1)

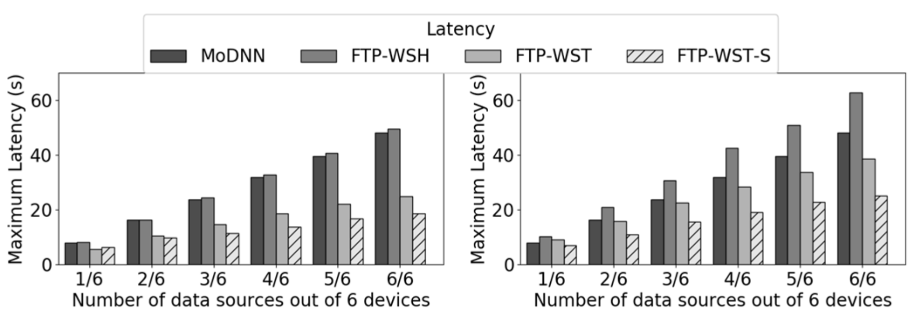
- **Single data source**
  - 6.8s with 3.5x speedup in FTP-WST-S, 6-device network
  - 8.1s with 2.1x speedup MoDNN, 6-device network
  - Scalability benefits in DeepThings
    - FTP: Avoid intensive intermediate data exchange
    - WST: Adaptively use communication bandwidth and exploit communication overhead
  - Data-reuse aware scheduling reduces 27% latency



FTP-3x3 **FTP-5x5**

Latency (2)

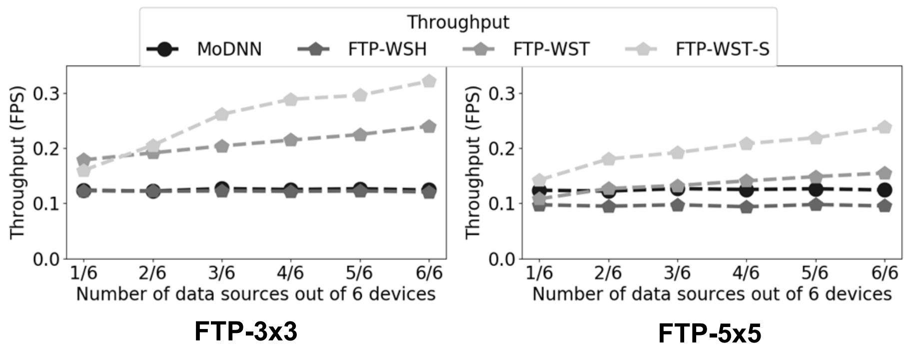
- **Multiple data sources**
  - Maximum latency
    - MoDNN: proportional with number of sources
    - FTP-WST-S: 3.1x with data source(s) increasing from 1 to 6



FTP-3x3 **FTP-5x5**

Throughput

- **Multiple data sources**
  - Throughput
    - MoDNN: 0.12 FPS with 6 data sources
    - FTP-WST-S: 0.29 FPS with 6 data sources



Case Study Summary

- **Distributed deep learning on the edge**
  - Leverage idle devices and idle times in local edge cluster
  - Partition to meet resource constraints (memory, comm.)
  - Speedup through parallelization (computation)
  - Orthogonal to cloud offloading and model simplification
    - Partition model between edge and cloud statically or dynamically
    - Generic or edge-specific model compression and pruning approaches
- **DeepThings**
  - Fused Tile Partitioning (FTP)
  - Lightweight work-stealing middleware w/ work scheduling
  - ➢ Efficient, flexible and adaptive inference task distribution

Open-source release at https://github.com/SLAM-Lab/DeepThings

Outline

- ✓ Introduction
  - ✓ Motivation, background
- ✓ NoS Design Flow
  - ✓ From system-level design to network-level design
- ✓ Application Case Study: Deep Learning on the Edge
  - ✓ Application partitioning & middleware design
- **NoS Specification and Analysis**
  - Programming model & real-time scheduling
- NoS Simulation and Exploration
  - Network/system co-simulation & design space exploration
- Summary & Conclusions

Specification & Analysis

- **Formal approaches to provide static guarantees**
  - Specification/programming model: determinism, correctness
  - Static analysis: latency, throughput, … guarantees
  - Synthesis: scheduling and mapping
- **For networks-of-systems with**
  - Distributed, heterogeneous computation and communication
  - Unreliable, unpredictable/unbounded open networks (IoT)
  - ➢ How to provide real-time, determinism, … guarantees?

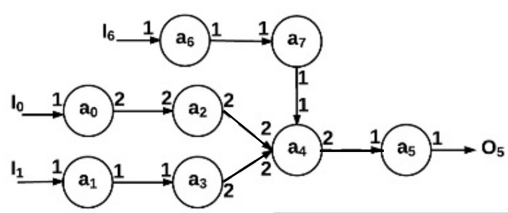
➢ **Models of Computation and Communication (MoCC)**
  - ➢ Model-based design

Traditional Approaches

- **Models of Computation (MoCs) in system-level design**
  - On-chip [SDF, KPN, …] or distributed [PTIDES]
  - ➢ Reliable & bounded communication
- **Distributed embedded & real-time computing**
  - Middleware [RT-CORBA, RMI/RTSJ, Storm, …]
  - ➢ Computation focus, limited network analyzability
- **Network protocols and network analysis**
  - Real-time audio/video streaming [RTP]
  - Queueing theory, network calculus
  - Trade-offs between quality, buffer size and latency
  - ➢ Deterministic or end-to-end communication only, no distributed computation

Reactive and Adaptive Data Flow (RADF) [ESL'17]

- **Traditional dataflow basis**
  - Embedded applications of streaming nature
- **Extended by two channel types**
  - Lossless (solid) and lossy (dashed) channels



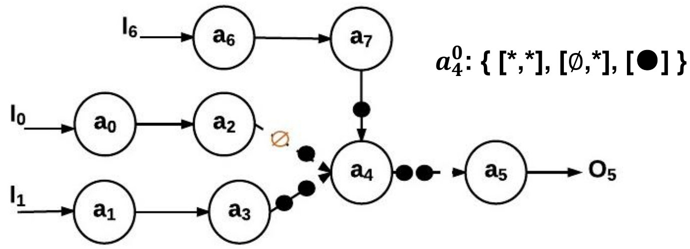
- **Lossless channels**
  - Map stream of tokens to ide
- **Lossy channels**
  - May replace tokens with empty ones: $[* \cdots *] \rightarrow [* \cdots \emptyset \cdots *]$

Empty tokens
- **Tokens that do not carry useful data**
- **Model network losses**
- **Preserve token order determinism**

Source: S. Francis, A. Gerstlauer, "A Reactive and Adaptive Data Flow Model For Network-of-System Specification," IEEE ESL, 9(4), 2017.

Adaptivity

- **Actor variants**
  - Applications need to handle data losses and empty tokens
  - Different execution versions of each actor
  - Based on firing rules of input token patterns [...{*,∅,●}...]

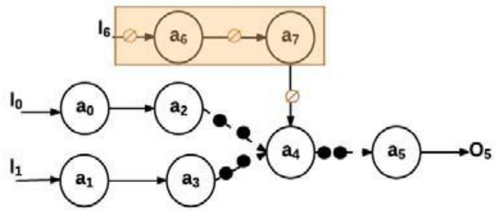


  - Default version fires if no other pattern matches
  - All variants produce only non-empty output tokens

➢ **Model dynamic behavior**

Reactivity

- **Idle actor variants**
  - Fires when all input tokens are empty ([∅,∅,...] patterns)
  - Does not perform any computation
  - Produces only empty tokens
- **Reactive Island**
  - Chain of connected actors with idle variants
  - No actor in island fires if input to chain is empty

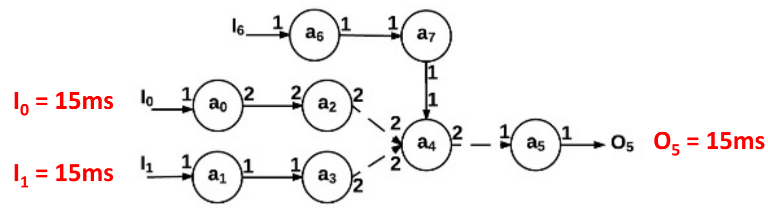


➢ **Empty tokens model absence of events and reactivity**

Timed RADF (T-RADF)

- **How to detect losses and inject empty tokens at runtime?**
  - Can't wait for token not to arrive → need timeouts
  - No distributed global time, decide based on local time only
  - ➢ Relative timeouts between firings

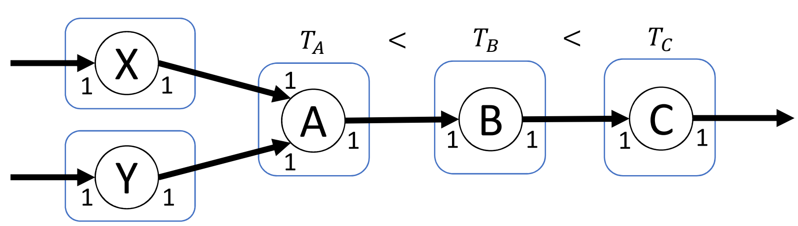
➢ **T-RADF extends RADF with rates on input and outputs**
  - Set (average) timeouts based on firing rates
  - Firing rates derived from external rates + repetition vector



Network-Level Analysis & Synthesis

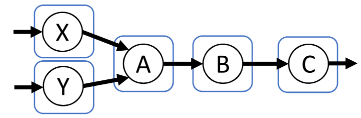
- **Key challenge**
  - Real-time guarantees over unpredictable networks?
- **Latency guarantees provided via timeouts**
  - Fundamental tradeoff between latency and losses (quality)
  - Per-actor timeouts

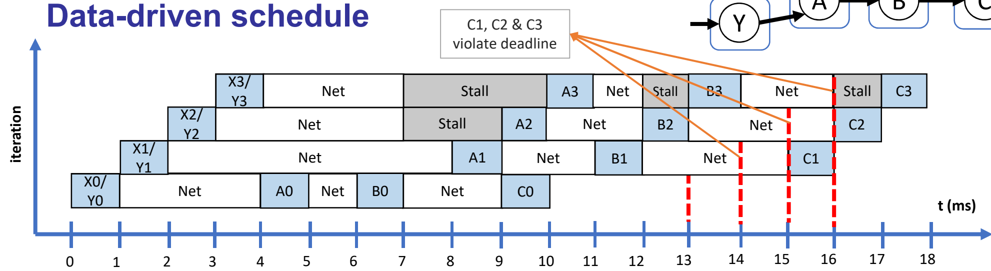
➢ **Timeout assignment for distributed real-time data flow**
  - Partition latency budget across nodes
  - Application/network-dependent tradeoff



Latency Budget Assignment

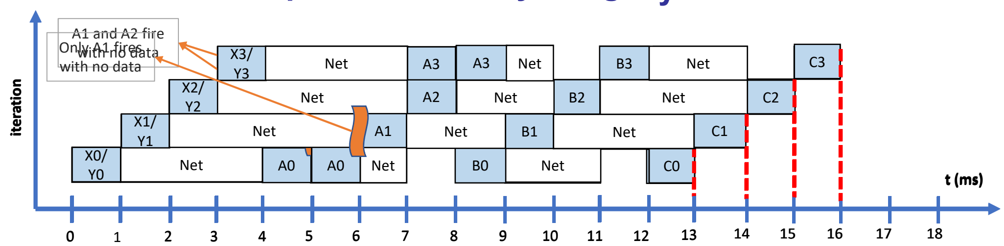
Data-driven schedule



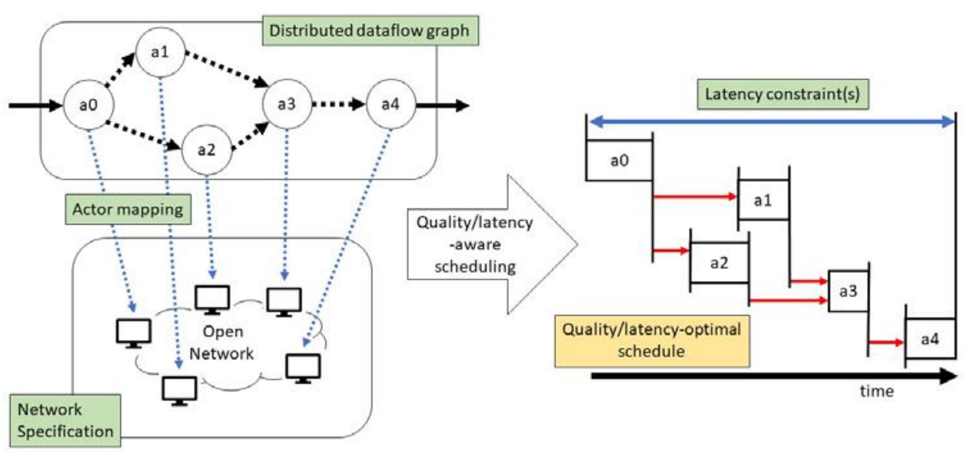


Schedule with optimized latency budget distribution

Schedule with timeouts & uniform latency distribution



Distributed Real-Time Scheduling [TECS'19]



- Derive a schedule for the given dataflow graph using
  - Mapping and worst-case execution times (WCETs)
  - Latency constraints
  - Network delay characteristics

Source: K. Mirzazad, Z.Zhao, A. Gerstlauer, "Quality/Latency-Aware Real-time Scheduling of Distributed Streaming IoT Applications," ACM TECS, 2019.

Scheduling Formulation

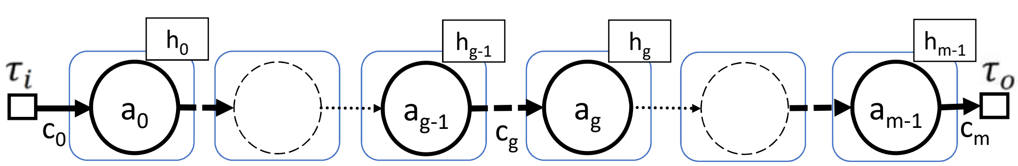
- **Assumptions**
  - Homogeneous dataflow (task graph)
    - Any graph can be made homogeneous albeit exponentially larger
  - One actor per host
    - Statically schedule actors mapped to the same host into a super-actor
- **Conservative analysis**
  - Fixed static schedule with specified periods
  - Can adjust schedule dynamically to optimize latency/quality
  - ➢ Derive relative start time offsets/phase shifts

Schedule Computation

- **Latency l between input-output pair**
  - Depends on actor execution times e_i and channel delays d_j

$$l = \sum_{i=0}^{m-1} e_i + \sum_{j=1}^{m-1} d_j \leq l'$$

- **Latency constraint l' requires bounding e_i and d_j**
  - Worst-case execution time bounds: $e_i \leq e'_i$
  - Goal: find bounds d'_j for d_j
- ➢ **Find d'_j to satisfy l' and maximize output quality Q**
  - d'_j affects token delivery probability p_j and therefore quality
  - ➢ Quality model to describe Q in terms of d'_j and statistical network delays



Schedule Computation

- **Optimization problem: find schedule that maximizes quality**

$$\underset{\mathbf{d}'}{\text{maximize}} \quad Q(\mathbf{d}', \tau)$$

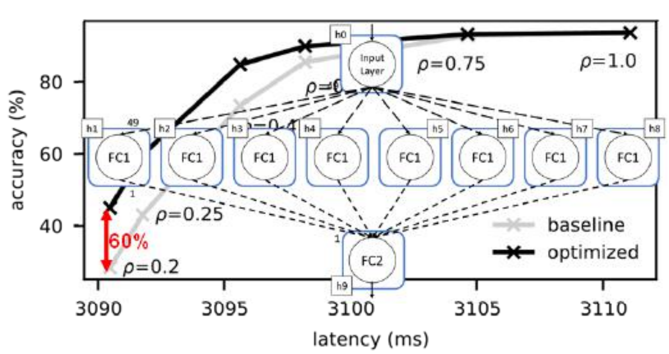
$$\text{subject to} \quad d'_j \geq 0, \ \forall j \in channels. \qquad \text{precedence constraints}$$

$$\sum_{i=0}^{m_k-1} e'_{(k,i)} + \sum_{j=1}^{m_k-1} d'_{(k,i)} \leq l'_k, \ \forall k \in paths. \qquad \text{latency constraints}$$

- **Solving the optimization problem**
  - $\boldsymbol{Q}$ is non-convex but has closed form & is differentiable
  - $\boldsymbol{d}'$ are continuous
  - ➢ Use numerical gradient-based iterative methods
    - ➢ E.g. Constrained Trust Region (CTR) solver

Distributed Neural Network Example

- **Two-layer network for MNIST digit recognition**
  - Simulated in OMNet++ using INET cloud model (gamma dist.)
  - Each token is 16 doubles, WCET of 1s for FC1/FC2
  - Account for network delay of 49 tokens in WCET of Input



- ➢ **Significant accuracy gains under tight latency constraints**
  - ➢ Up to 60%, on average 20%

Specification & Analysis Summary

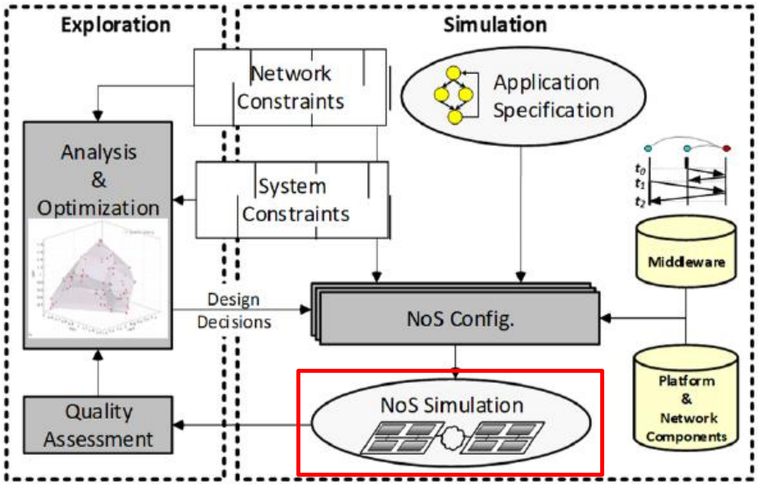
- **Real-time analysis and synthesis**
  - Unpredictable and unbounded network delays
  - Fundamental quality vs. latency tradeoffs
  - Scheduling and latency budgeting to optimize tradeoff
  - Can be extended to computation actors
    - Tighter deadlines than unknown or overly conservative WCET estimates
    - Drop/reduce task if deadline violation: approximate/imprecise computing
- **Quality/latency-aware real-time scheduling**
  - Reactive and Adaptive Dataflow (RADF)
  - Probabilistic analysis based on network delay distributions
  - Optimize end-to-end quality under real-time guarantees
  - ➢ Tools, graphs and simulation models available at:

Open-source release at https://github.com/SLAM-Lab/QLA-RTS

Outline

- ✓ **Introduction**
  - ✓ Motivation, background
- ✓ **NoS Design Flow**
  - ✓ From system-level design to network-level design
- ✓ **Application Case Study: Deep Learning on the Edge**
  - ✓ Application partitioning & middleware design
- ✓ **NoS Specification and Analysis**
  - ✓ Programming model & real-time scheduling
- **NoS Simulation and Exploration**
  - Network/system co-simulation & design space exploration
- **Summary & Conclusions**

Network-Level Design Flow



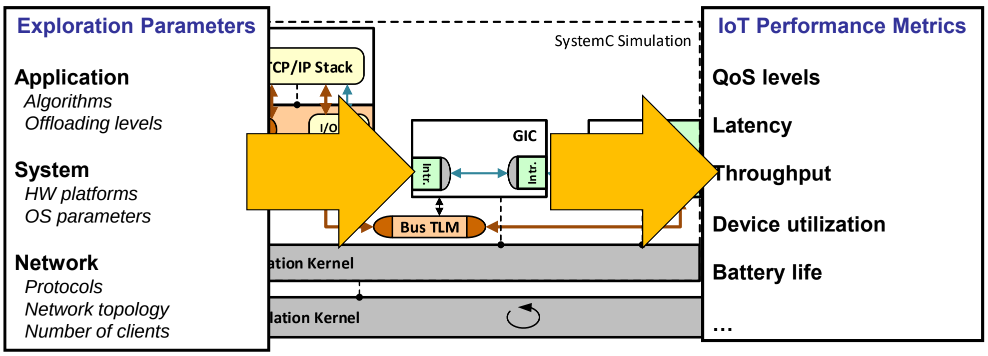
- ➢ **Flexible network/system co-simulation platform**
  - ➢ Instantiate various network/system configurations for exploration
  - ➢ Evaluate and explore dynamic effects and design decisions

Existing Approaches

- **IoT design space exploration**
  - Ad-hoc & application-specific with over-simplified models
    - Smart camera networks [Devarajan'06, Quaritsch'07]
    - Healthcare systems [Doukas'12, Catarinucci'15]
  - ➢ Limited design space exploration
- **Network and system simulation**
  - Existing network simulators
    - Network simulators w/o system device models [ns-3, OMNeT++]
    - WSN simulators w/ state-based system and over-simplified network models [Sommer'09, Bai'11, Du'11, Damm'10]
  - Existing system simulators
    - Detailed but slow full-system simulators [GEM5]
    - Host-compiled & transaction-level modeling (TLM) [Bringmann'15]
    - Simple network TLM extensions [Fummi'08, Banerjee'09]
  - ➢ No network/system co-simulation

NoSSim [SAMOS'17]

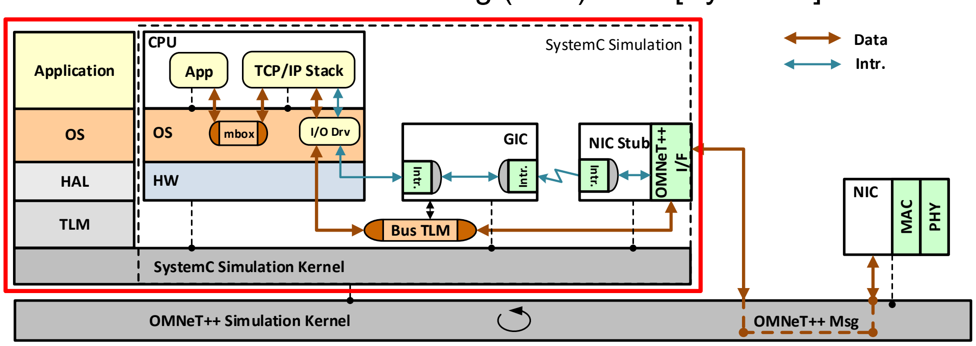
- **System simulation model**
  - SystemC-based host-compiled device model
  - Capture system-wide interactions between application, OS and underlying hardware components
- **Network simulation backplane**
  - OMNeT++/INET network simulation framework



Source: Z.Zhao, V. Tsoutsouras, D. Soudris, A. Gerstlauer, "Network/System Co-Simulation for Design Space Exploration of IoT Applications," SAMOS'17.

System Simulation Model

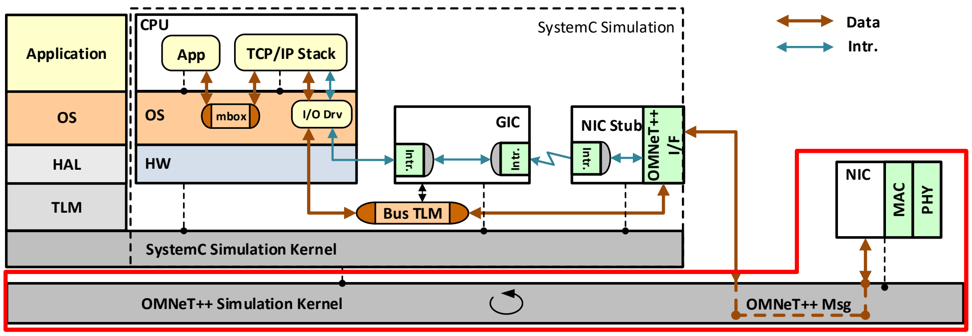
- **Host-compiled (HC) system simulator**
  - Source-level simulation w/ back-annotation [TCAD'16]
  - Abstract multi-core OS and processor models [TECS'14]
  - Network stack model ported onto simulator [lwIP]
  - Network interface card (NIC) stub model
  - Transaction-level modeling (TLM) base [SystemC]



Source: Z. Zhao, L. John, A. Gerstlauer. "Source-Level Performance, Energy, Reliability, Power and Thermal (PERPT) Simulation," IEEE TCAD, 2016.
Source: P. Razaghi, A. Gerstlauer. "Host-Compiled Multi-Core System Simulation for Early Real-Time Performance Evaluation," ACM TECS '14.

Network Simulation Backplane

- **Detailed NIC model in OMNeT++**
  - Media access (MAC) and physical (PHY) layer simulation
  - Interfacing with SystemC stub through OMNeT++ messages
- **SystemC/OMNeT++ integration**
  - Scheduled and synchronized globally by OMNeT++
  - Multiple device instances in given network topology

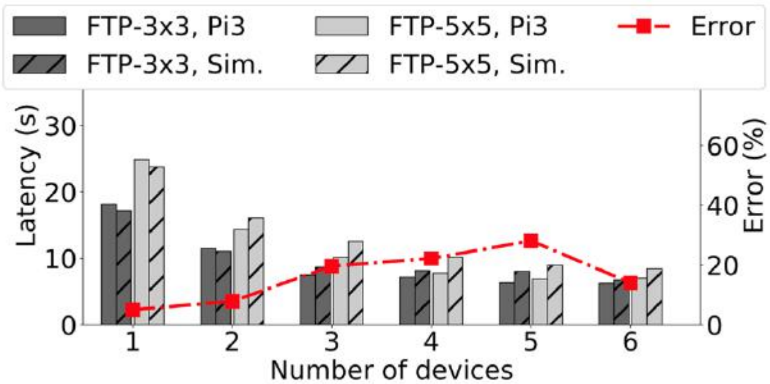


NoSSim Evaluation

- **Platform setup**
  - Compare NoSSim against Raspberry Pi 3 device cluster
  - Raspberry Pi 3:
    - Raspbian
    - Linux TCP/IP stack
    - IEEE 802.11n
  - NoSSim:
    - Host-compiled OS Model, function-level back-annotation
    - lwIP TCP/IP stack
    - IEEE 802.11n (OMNET++/INET)
- **Application setup**
  - DeepThings
    - FTP partition dimensions: 3x3~5x5
    - Edge cluster size: 1~8
    - Data sources: 1~6
    - Distributed work stealing and work scheduling

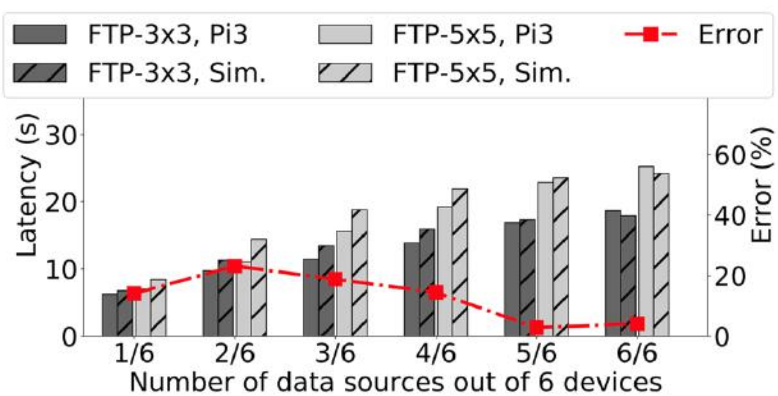
NoSSim Accuracy Validation

- **Simulation accuracy with single data source**
  - Number of edge devices: 1~6
  - FTP 3x3, 5x5



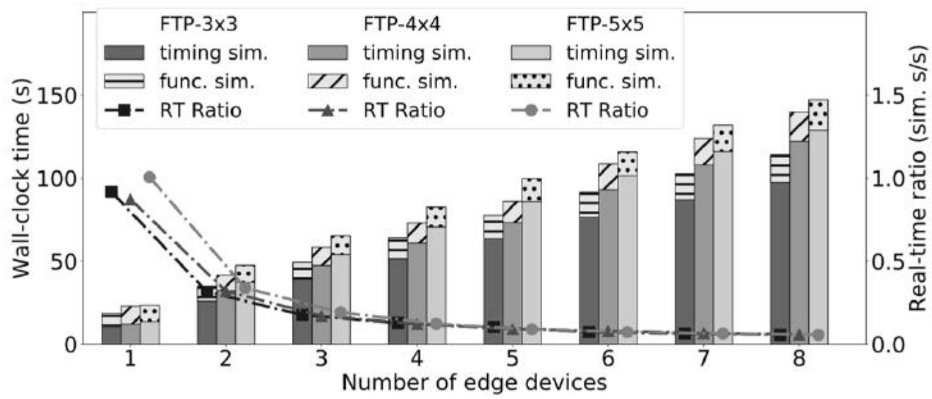
NoSSim Accuracy Validation

- **Simulation accuracy with multiple data source**
  - Number of edge devices: 1~6
  - FTP 3x3, 5x5

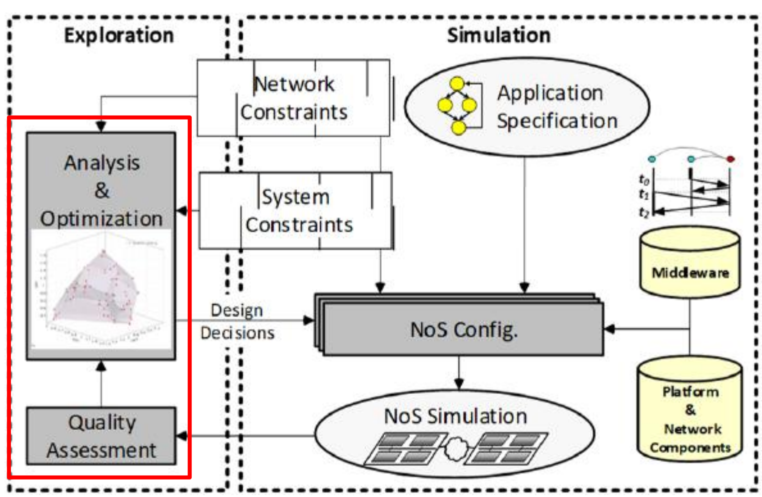


NoSSim Runtime

- **Simulation wall-clock time**
  - Timing simulation: task models without functionality
  - Functional simulation: full CNN inference computation



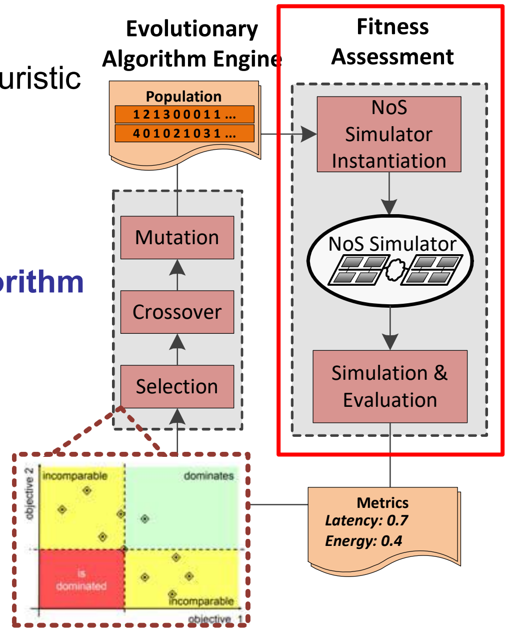
Network-Level Design Flow



- **Automated design space exploration (DSE)**
  - Explore Pareto front in multi-objective design space
  - Meta-heuristics for exploration

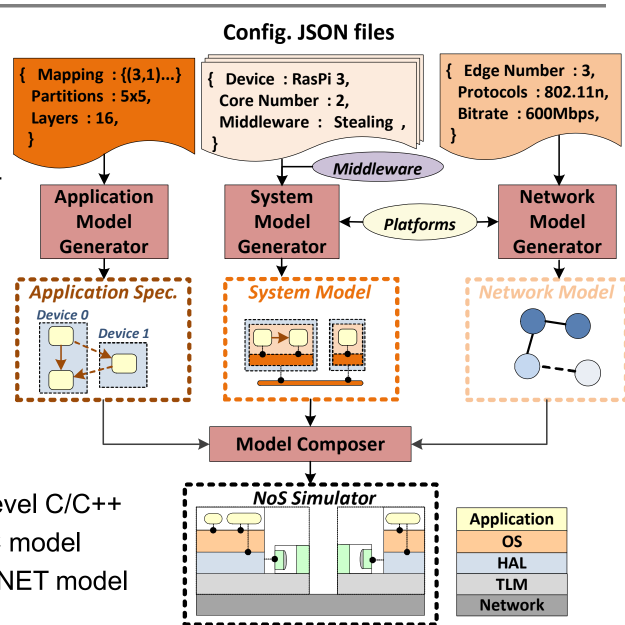
NoS Design Space Exploration (DSE)

- **Evolutionary algorithm**
  - Population-based metaheuristic optimization algorithm
  - Mechanisms inspired by biological evolution
- **Multi-objective genetic algorithm**
  - Non-Dominated Sorting Genetic Algorithm-II (NSGA-II)
  - Strength Pareto Evolutionary Algorithm (SPEA2)
  - ➢ Pareto optimal solutions

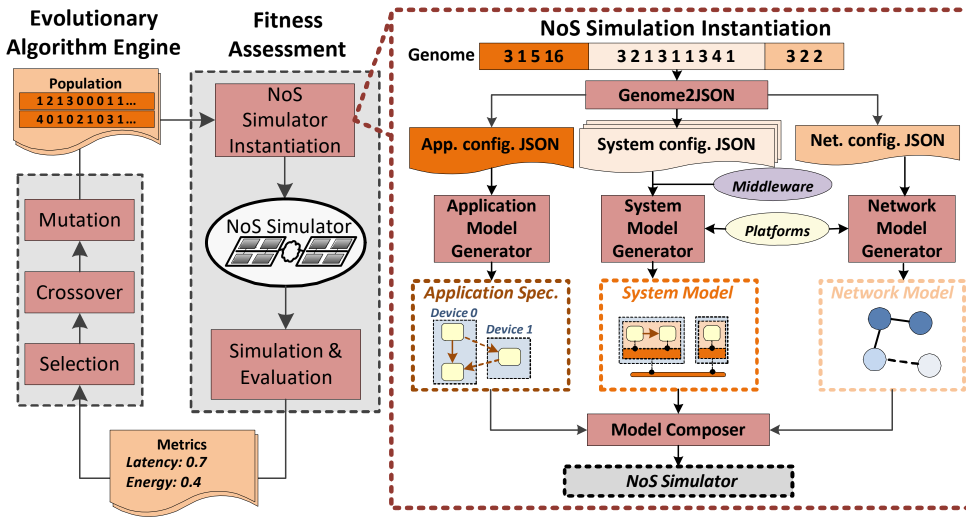


NoS Model Generation

- **Tunable parameters**
  - Application level
    - Static task mapping
    - App. specific config.
  - System level
    - Number or cores
    - Device type
  - Network level
    - Protocol
    - Number of nodes
- **Model generation**
  - Application: Source-level C/C++
  - System: SystemC HC model
  - Network: OMNeT++/INET model



NoSDSE Framework [TECS'20]

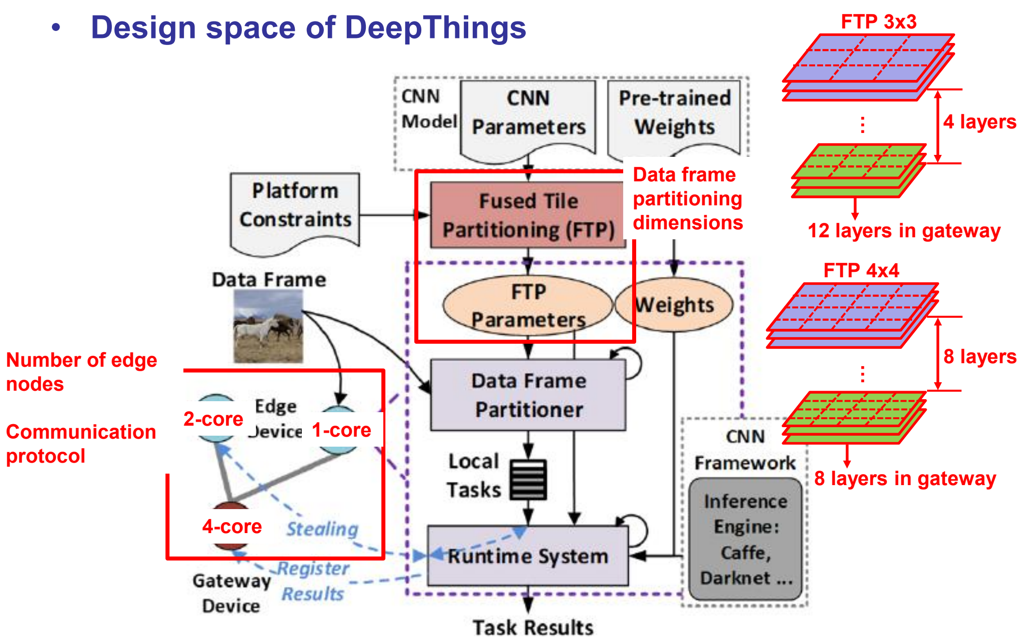


➢ **Automated NoS design space exploration**

Source: Z. Zhao, K. Mirzazad, A. Gerstlauer, "Network-level Design Space Exploration for Resource-Constrained Networks-of-Systems," ACM TECS, revised, 2019.

NoS DSE Case Study

- **Design space of DeepThings**



Exploration Setup

- **Design space exploration parameters**

Application & NoS Parameters

FTP Grid Dimensions	3x3, 4x4, 5x5
Fusing Layers	4, 8, 12, 16
Edge Core Number	1, 2, 4
Gateway Core Number	4
Communication Protocol	802.11b@(5.5, 11Mbps) 802.11g@(6, 18, 48, 54Mbps) 802.11n@600Mbps
Edge Node Num.	1~12
Data Source Num.	1

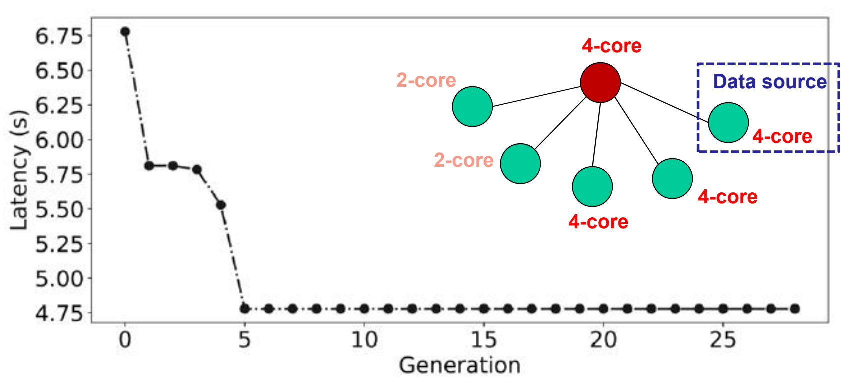
NSGA-II Parameters

Generations	30
Population	80
Mutation Rate	0.8
Crossover Rate	0.8

- **Optimization targets**
  - Average device energy consumption
  - Frame inference latency
  - Memory requirement

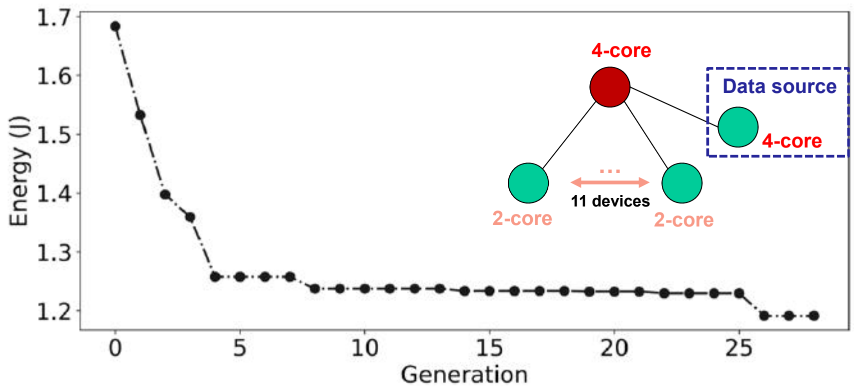
Latency Optimization

- **Optimization progress**
  - FTP 3x3 partitioning dimension
  - Heterogenous cluster with 5 devices, FTP 3x3, 12 layers
  - IEEE 802.11n communication protocol



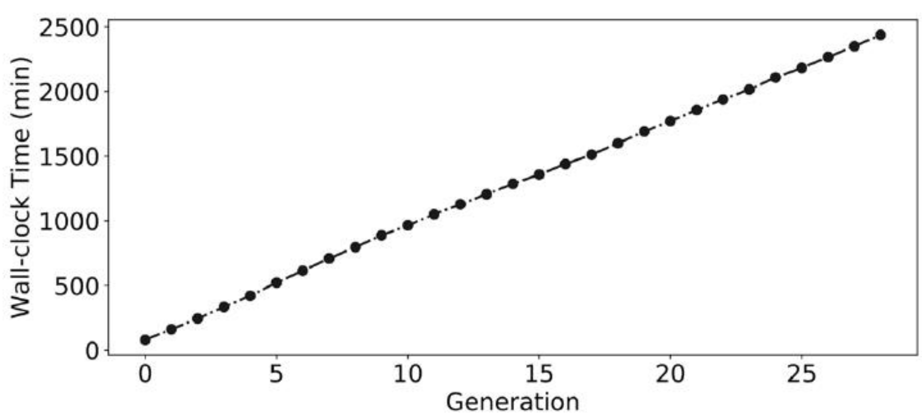
Energy Optimization

- **Optimization progress**
  - Best energy efficiency with dual-core device
  - 12-device processing cluster, FTP 3x3, 4 layers
  - Less than 3% energy on radio interface



Exploration Runtime

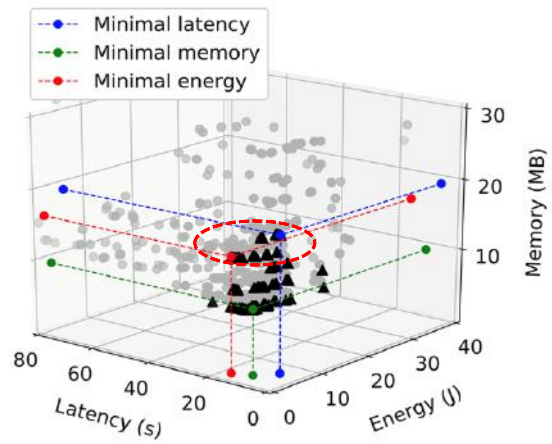
- **Accumulated runtime**
  - Evaluation fitness with 6 processes
  - An average of 80 min for one generation



Pareto Front of DeepThings Design Space

- **Trade-offs between latency and energy, memory**

Design Metrics	Device	Cluster Size	Grid Dimension	Fusing Depth
Latency	**Powerful**	**Medium**	**Coarse**	**Medium**
Energy	**Medium**	**Large**	**Coarse**	**Small**
Memory	**-**	**-**	**Fine**	**Small**



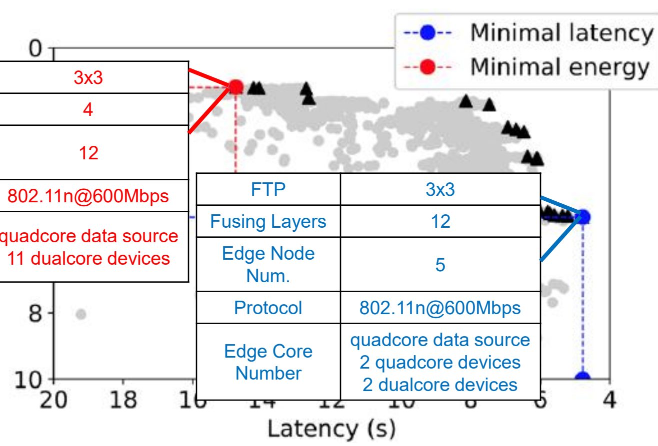
Minimal memory

FTP	5x5
Fusing Layers	4

Pareto Front of DeepThings Design Space

- **Trade-offs between latency and energy, memory**

Design Metrics	Device	Cluster Size	Grid Dimension	Fusing Depth
Latency	**Powerful**	**Medium**	**Coarse**	**Medium**
Energy	**Medium**	**Large**	**Coarse**	**Small**
Memory	**-**	**-**	**Fine**	**Small**



FTP	3x3
Fusing Layers	4
Edge Node Num.	12
Protocol	802.11n@600Mbps
Edge Core Number	quadcore data source 11 dualcore devices

FTP	3x3
Fusing Layers	12
Edge Node Num.	5
Protocol	802.11n@600Mbps
Edge Core Number	quadcore data source 2 quadcore devices 2 dualcore devices

Simulation & Exploration Summary

- **NoS simulator (NoSSim)**
  - SystemC based host-compiled system model
  - INET/OMNeT++ network simulation
  - ➢ Fast, comprehensive and flexible network/system co-simulation

Open-source release at https://github.com/SLAM-Lab/NoSSim

- **NoS design space exploration (NoSDSE)**
  - Automatic simulation model instantiation
  - Genetic algorithm based multi-objective optimization
  - ➢ Fully automated, simulation-based DSE

Open-source release at https://github.com/SLAM-Lab/NoSDSE

Summary & Conclusions

- **The era of network-level design?**
  - Joint network/system co-design
  - Beyond traditional system-level design
- **Unique challenges?**
  - Resource-constrained heterogeneous distributed computing
  - Network uncertainties & correctness/real-time guarantees

➢ **Network-level design automation**
  - ➢ MoCCs for specification & analysis (→ RADF)
    - Compilers, static analysis and real-time scheduling (→ QLA-RTS)
  - ➢ Simulators and performance models (→ NoSSim)
    - Design space exploration (→ NoSDSE)
  - ➢ Runtime and middleware systems (→ DeepThings)
    - Dynamic application partitioning, mapping and scheduling