Task Scheduling of SDR Kernels in Heterogeneous Chips
Opportunities and Challenges

Augusto Vega[1]
Aporva Amarnath[2]
Alper Buyuktosunoglu[1]
Hubertus Franke[1]
John-David Wellman[1]
Pradip Bose[1]

[1] IBM T. J. Watson Research Center
[2] University of Michigan

Acknowledgment

- Thanks to the many IBM colleagues who contribute to and support different aspects of this work + our esteemed university collaborators at Harvard, Columbia, and UIUC (Profs. David Brooks, Vijay Janapa Reddi, Gu-Yeon Wei, Luca Carloni, Ken Shepard, Sarita Adve, Vikram Adve, Sasa Misailovic) + many brilliant graduate students and postdocs!

- Special thanks to **Dr. Thomas Rondeau**, Program Manager of the DARPA MTO DSSoC Program

This research was developed, in part, with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This document is approved for public release: distribution unlimited.

Outline

- **Part 1: The Hardware Specialization Era**
  – And its impact on SDR applications

- **Part 2: Task Scheduling on Heterogeneous Platforms**
  – STOMP: Scheduling Techniques Optimization in heterogeneous Multi-Processors

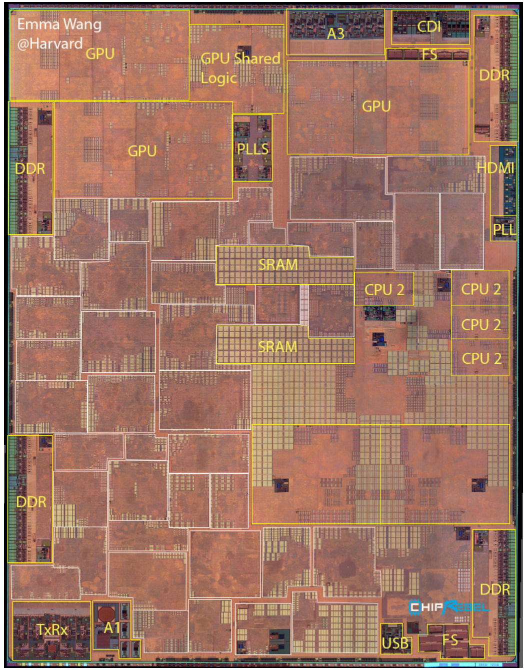
- **Part 3: New Scheduling Techniques**
  – Evaluation and future work

The Hardware Specialization Era Is Already Here...

- Heterogeneous system-on-chips (SoCs) are single chips comprising of many processing elements (PEs) of different nature like CPUs, GPUs and hardware accelerators

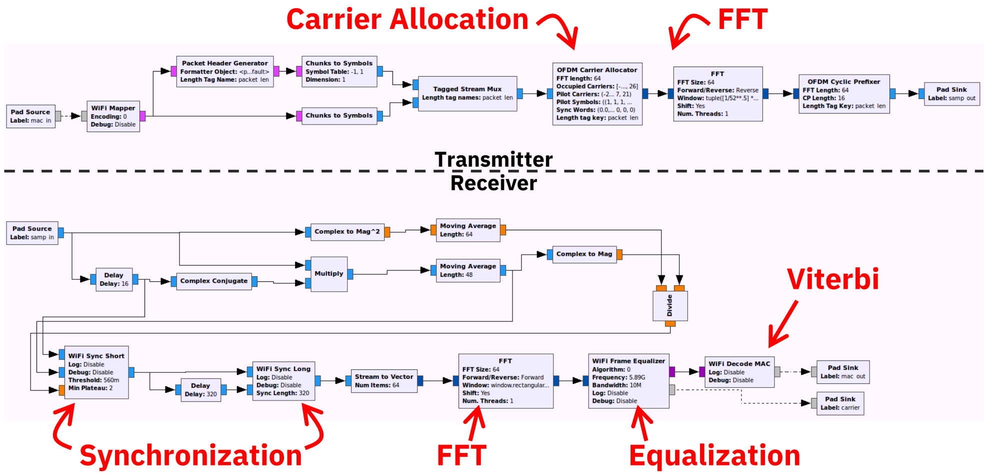
- Heterogeneous SoCs are **extensively** used today
  - Adopted by domains historically dominated by homogeneous architectures
  - Exploit heterogeneous characteristic of applications
  - Significant performance and power efficiency gains

Conventional schedulers are **not optimized** for the characteristics of heterogeneous chips which calls for more intelligent and efficient scheduling

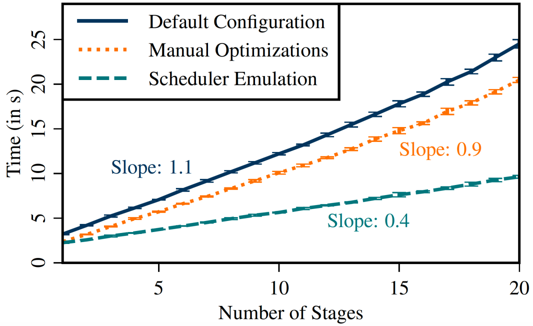


Source: https://www.sigarch.org/mobile-socs/

SDR and the Impact of Specialization & Task Scheduling



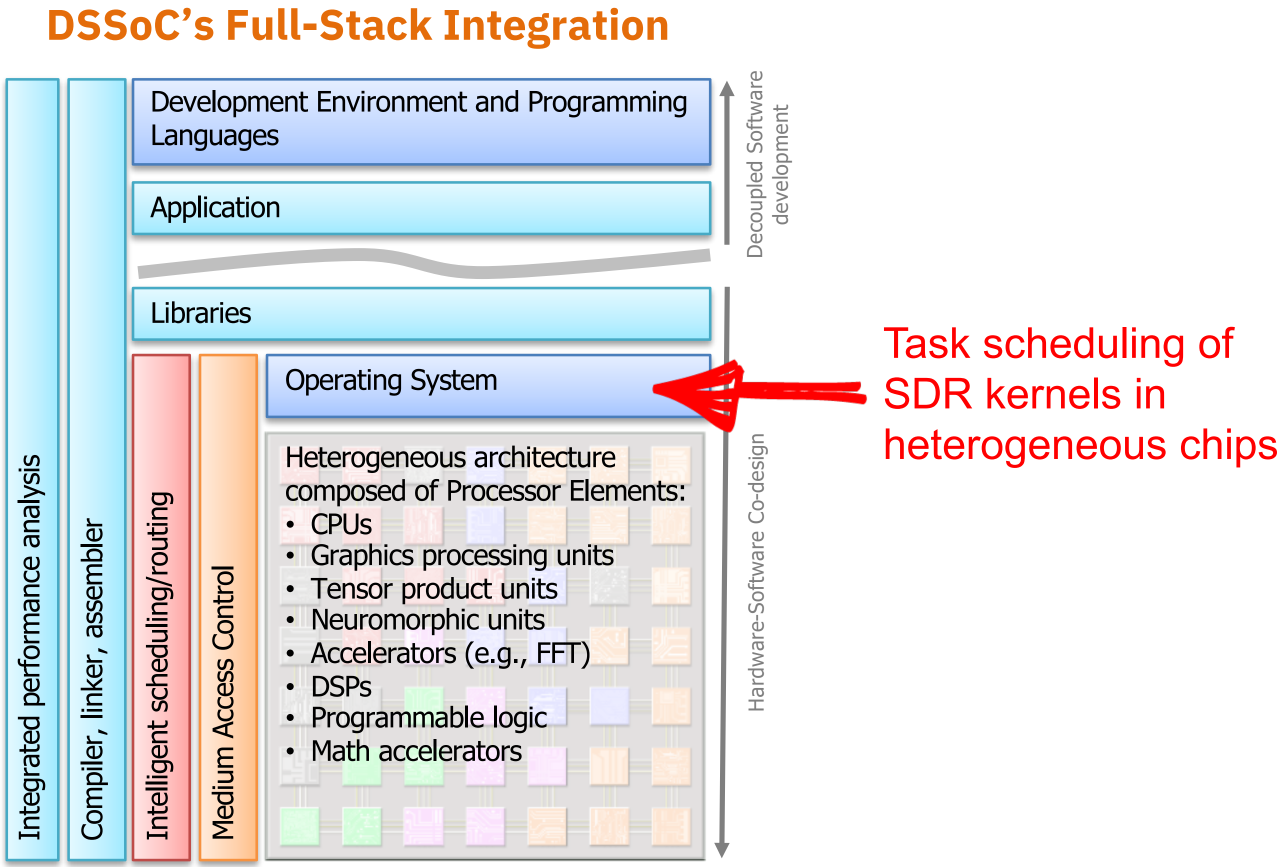
- A typical SDR application may consist of multiple and disparate kernels
- The underlying hardware may also provide accelerators for some or all of them
- However, in frameworks like GNU Radio, the scheduler mostly “ignores” these degrees of heterogeneity – which may provide significant benefits when properly exploited



Prior works have shown that there is significant room for improvement in the GNU Radio scheduler
– E.g. via simple scheduling optimizations to increase cache effectiveness [1]

[1] B. Bloessl, M. Müller, M. Hollick. “Benchmarking and Profiling the GNURadio Scheduler.” Proceedings of the 9th GNU Radio Conference. 2019.

The Big Picture (Where Does This Talk Fit In?)



Outline

- **Part 1: The Hardware Specialization ERA**
  - And its impact on SDR applications

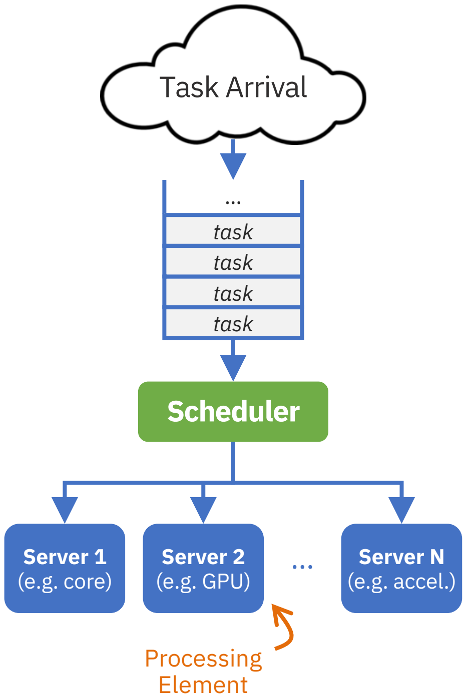
- **Part 2: Task Scheduling on Heterogeneous Platforms**
  - STOMP: Scheduling Techniques Optimization in heterogeneous Multi-Processors

- **Part 3: New Scheduling Techniques**
  - Evaluation and future work

STOMP

- STOMP (**S**cheduling **T**echniques **O**ptimization in heterogeneous **M**ulti-**P**rocessors) is an open-source customizable Python-based simulator for fast prototyping of SoC scheduling policies
  - Check it out: https://github.com/IBM/stomp

- It consists of three main elements:
  - **Tasks:** units of work (aka *jobs*, *threads*, *processes*)
    - Executed in the heterogeneous SoC
    - Typically described as task types (e.g. *fft*, *decoder*, etc.)
  - **Servers:** processing units that can execute tasks
    - Different servers execute tasks with different "efficiency"
    - E.g. an FFT task on DSP accelerator *vs* general-purpose CPU
  - **Scheduler:** dynamically maps tasks to servers during the execution
    - It supports user-defined scheduler algorithms



STOMP Overview

Python

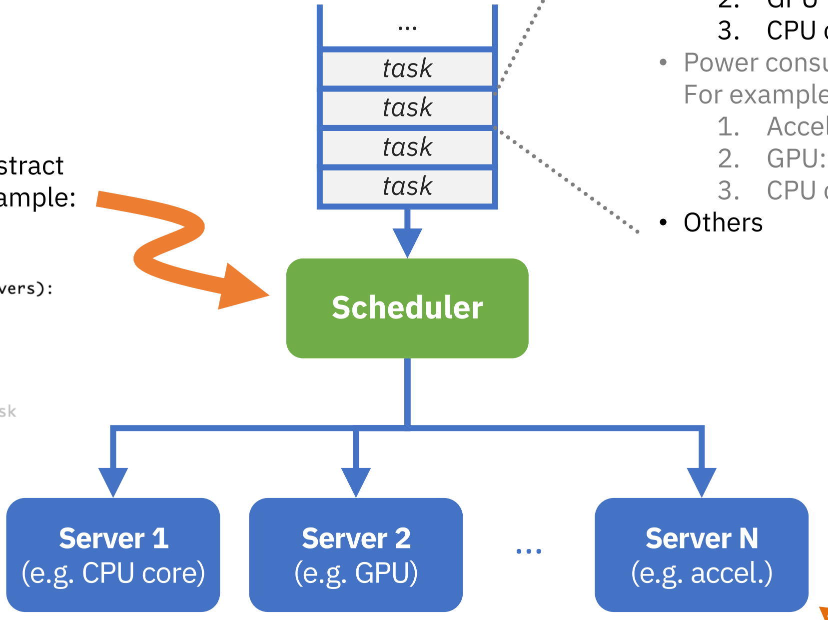
“Pluggable” Scheduling Policy

- The user is only required to implement the abstract Python class `BaseSchedulingPolicy` – for example:

```
class SchedulingPolicy(BaseSchedulingPolicy):

    def assign_task_to_server(self, sim_time, tasks, servers):

        if (len(tasks) == 0):
            # There aren't tasks to serve
            return None

        # Look for an available server to process the task
        for server in servers:

            if (not server.busy):

                # Pop task in queue's head
                task = tasks.pop()
                server.assign_task(sim_time, task)

                return server

        return None
```

Task arrival

- Probabilistic (e.g. exponential)
- Realistic (trace-based)



Processing Element

JSON

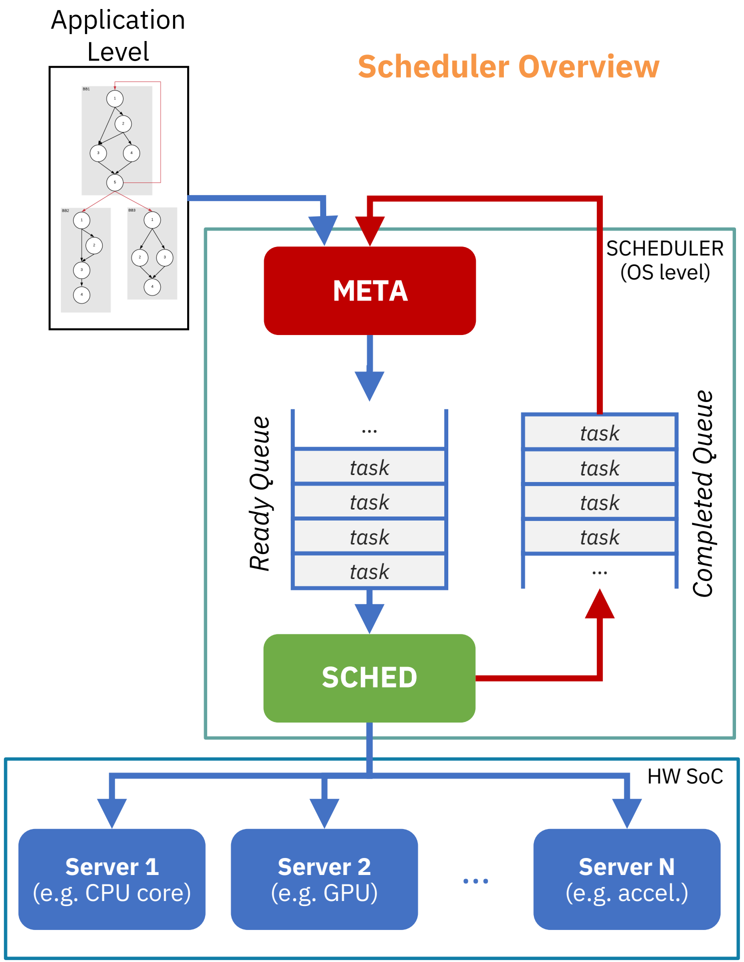
Task attributes

- Service time (probabilistic or trace-based)
- Target processing elements
  For example:
  1. Accelerator
  2. GPU
  3. CPU core
- Power consumption
  For example:
  1. Accelerator: 100 mW
  2. GPU: 400 mW
  3. CPU core: 900 mW
- Others

Future work

STOMP Intrinsic Operation

- STOMP consists of two integral parts:
  - **Meta scheduler** (“META”) ➜ pre-processor that aids in the scheduling decision
  - **Task scheduler** (“SCHED”) ➜ assigns ready tasks to available servers (PEs) to optimize the overall response time
- META and SCHED communicate via two queues: *ready* and *completed*
- **Input:** directed acyclic-graphs (DAGs) of multiple tasks with associated real-time constraints (priority and deadline)

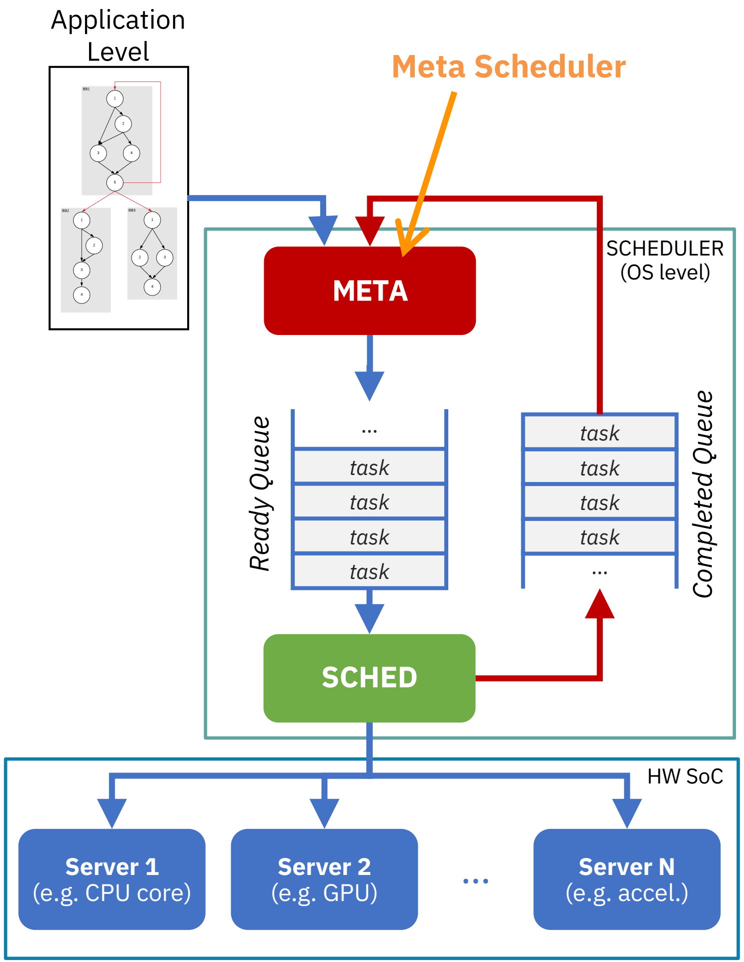


Meta Scheduler ("META")

- META tracks heuristics associated with the DAG:
  - Task dependencies, DAG deadline and available slack, DAG and tasks priority
- Then orders *ready* tasks based on a **"rank"**
  - Can be computed in different ways
  - For example, as a function of task's priority, slack and worst-case execution time (WCET)

$$Rank_i = \frac{Task_i\ Priority}{Task_i\ Slack\ - Task_i\ WCET}$$

- Drops non-critical priority DAGs if deadline is missed
  - All remaining tasks in the DAG are dropped
  - Help reduce task traffic in the system

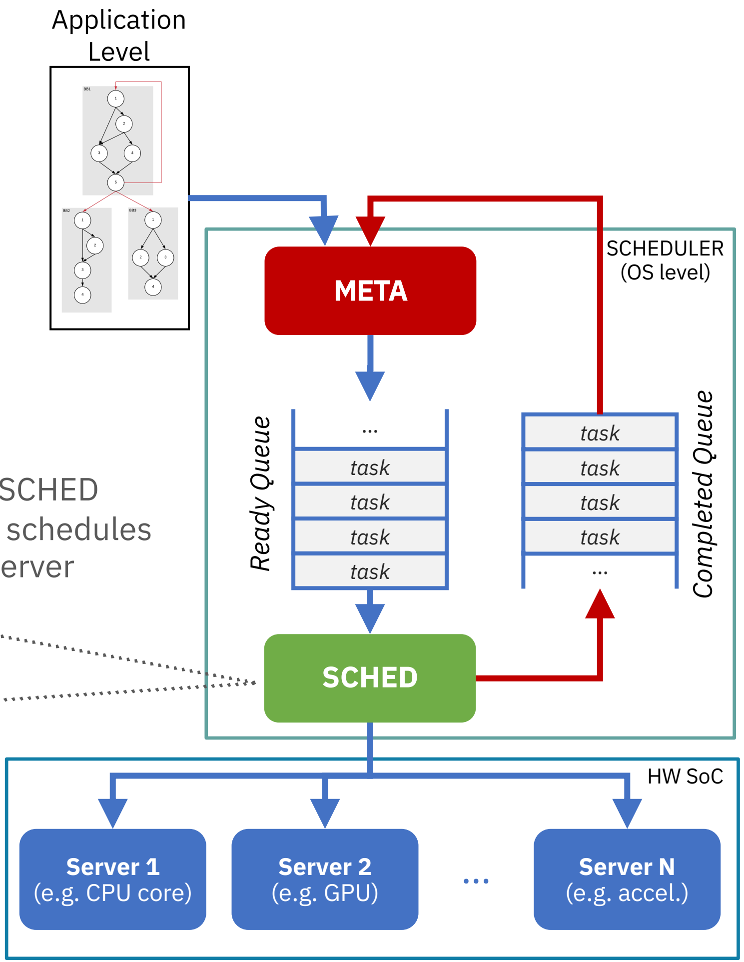


Task Scheduler ("SCHED")

The user primarily defines the assignment actions:
(here the task is scheduled to the fastest server type)

```
from stomp import BaseSchedulingPolicy

class SchedulingPolicy(BaseSchedulingPolicy):

  def init(self, servers, stomp_stats, stomp_params):
    ...

  def remove_task_from_server(self, sim_time, server):
    ...

  def assign_task_to_server(self, sim_time, tasks):

    if (len(tasks) == 0):
    # There aren't tasks to serve
      return None

    # Determine task's best scheduling option (target server)
    target_server_type = tasks[0].mean_service_time_list[0][0]

    # Look for an available server to process the task
    for server in self.servers:
      if (server.type == target_server_type and not server.busy):
        # Pop task in queue's head and assign it to server
        server.assign_task(sim_time, tasks.pop(0))
        return server
    return None
```

Invoked by SCHED each time it schedules a task to a server



Simulation Parameters and Configuration

- Example `stomp.json` configuration file:

```
"general" : {
    "logging_level":        "INFO",
    "random_seed":          0,
    "working_dir":          ".",
    "basename":             "",
    "pre_gen_arrivals":     false,
    "input_trace_file":     "",
    "output_trace_file":    ""
},

"simulation" : {
    "sched_policy_module":  "policies.simple_policy_ver3",
    "max_tasks_simulated":  10000,
    "mean_arrival_time":    50,
    "distribution":         "Poisson",
    "power_mgmt_enabled":   false,
    "max_queue_size":       1000000,
```

```
"servers" : {
          "cpu_core" :  { "count" : 8 },
          "gpu" :       { "count" : 2 },
          "fft_accel" : { "count" : 1 }
      },

"tasks" : {
          "fft" : {
              "mean_service_time" : {
                  "cpu_core"  : 500,
                  "gpu"       : 100,
                  "fft_accel" : 10
              },

              "stdev_service_time" : {
                  "cpu_core"  : 5.0,
                  "gpu"       : 1.0,
                  "fft_accel" : 0.1
              }
          },
   ...
```

Example Using a Simple DAG

- **Input:** priority-1 5-node DAG with varying kernels
  – Deadline of DAG is set to 1100 units of time

- Time 0: META pushes *Task 0* to *ready queue* with a rank

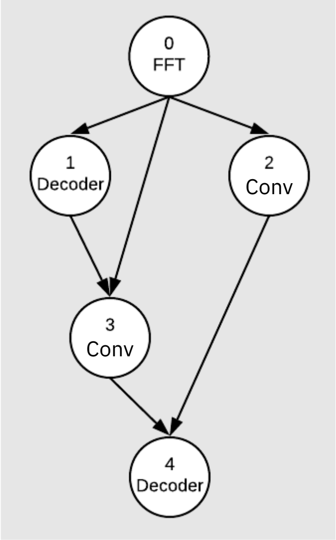
$$Rank_i = \frac{Task_i\ Priority}{Task_i\ Slack\ - Task_i\ WCET}$$

DAG's priority

$$Rank_0 = \frac{1}{500\ - 500} = \infty$$

- *Task 0* completes execution in 10 units of time because it was run on the accelerator
  – META then calculates the remaining slack of the DAG and next available tasks

5-node DAG



Task	CPU	GPU	Accel
FFT	500	100	10
Convolution	200	150	10
Decoder	200	150	None

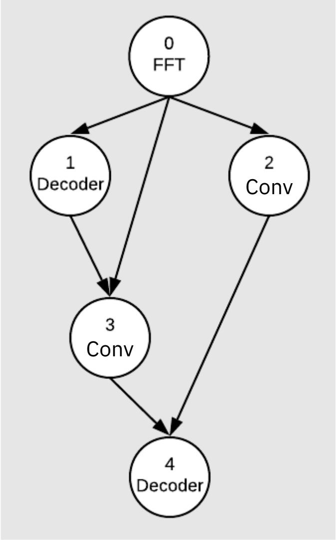
Tasks' Execution Times

Example Using a Simple DAG (cont'd)

- Time 10: *Task 1* and *Task 2* become ready
  - Scheduled in the order of their rank
  - *Task 1* has a higher rank than *Task 2*
    - $Rank_1 = 1/(363-200) = 1/163$
    - $Rank_2 = 1/(545-200) = 1/345$
  - **This process continues for all tasks in the DAG**

- **Multi-DAG execution:**
  - Multiple DAGs arrive consecutively
  - At every stage, ready tasks are scheduled in rank order across all DAGs

5-node DAG



Task	CPU	GPU	Accel
FFT	500	100	10
Convolution	200	150	10
Decoder	200	150	None

Tasks' Execution Times

Outline

- Part 1: The Hardware Specialization ERA
  - And its impact on SDR applications
- Part 2: Task Scheduling on Heterogeneous Platforms
  - STOMP: Scheduling Techniques Optimization in heterogeneous Multi-Processors
- **Part 3: New Scheduling Techniques**
  - Evaluation and future work

Evaluation

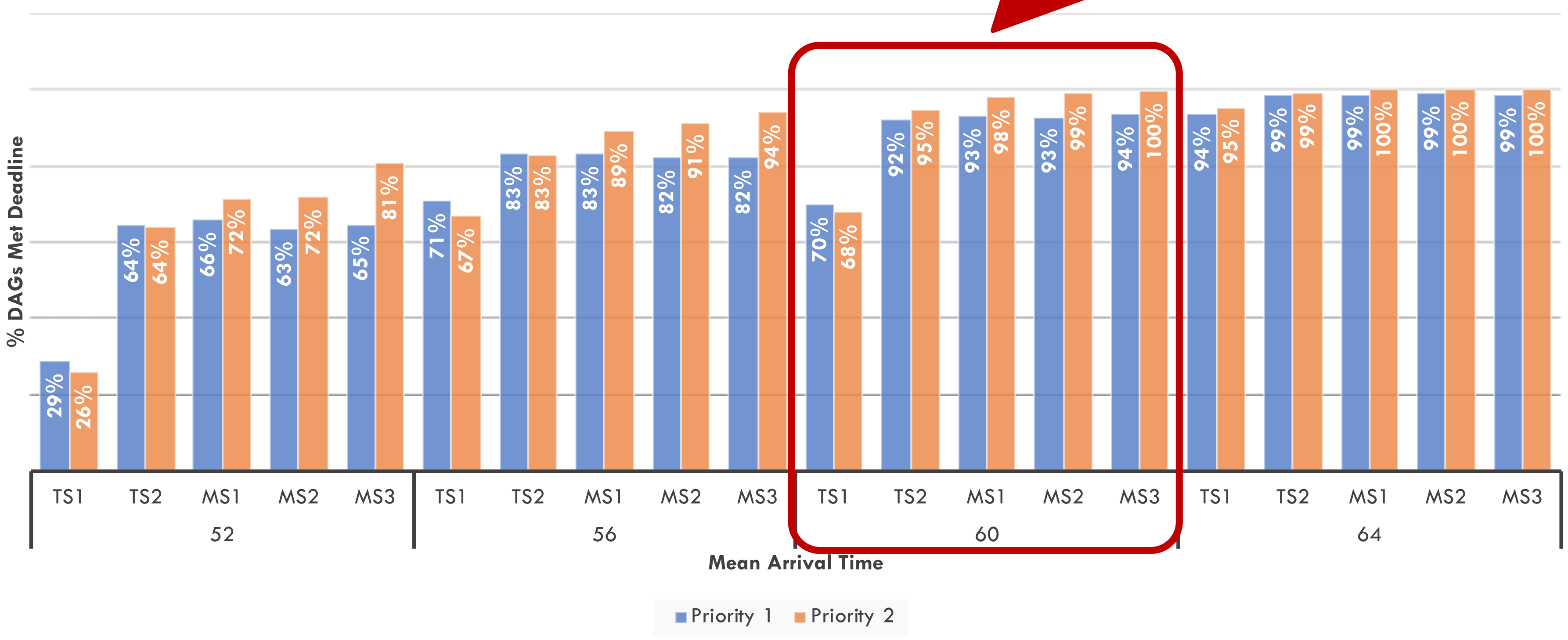
- DAG trace: **1,000** 5- and 10-node static DAGs
  - Priority: 1 or 2 assigned randomly
  - Deadline: critical path length considering worst-case execution times
- Task types:
  - FFT, Convolution, Decoder
- Metric of evaluation:
  - Met deadline

Task	CPU	GPU	Accel
FFT	500	100	10
Convolution	200	150	10
Decoder	200	150	None

- Baseline task schedulers with META dependency tracking only
  - **TS1:** non-blocking task scheduler
  - **TS2:** non-blocking task scheduler assuming tasks ahead in queue are scheduled
- TS2 scheduler with both META dependency tracking *and* pre-processing
  - **MS1:** rank based on task's deadline and average execution time, and priority
  - **MS2:** rank based on task's deadline and maximum execution time, and priority
  - **MS3:** rank based on task's available slack and maximum execution time, and priority

Evaluation: Met Deadline

MS3 meets deadline for 33% and 5% more tasks than TS1 and TS2, respectively



Running STOMP

```
ripper 00:44 ~/research/IBM/STOMP/stomp_clean:
```

Summary and Path Forward

- STOMP is in active development with a number of additional items being worked on
  - More complete input trace format, more statistics and data about the runs

- And there are some extensions planned
  - Power consumption models and power management features
  - **Machine learning-based scheduling policies**

- And work to move from the abstract to the more concrete
  - Analysis of GNU Radio workloads to generate more realistic DAG traces

- **But STOMP already provides plenty of opportunity and capability to explore the problem space – readily available now:**

https://github.com/IBM/stomp

(check out *dev* for leading-edge features)

Thank You!

Photo by Balthazar Korab
Source: http://www.shorpy.com/node/15488

ajvega@us.ibm.com

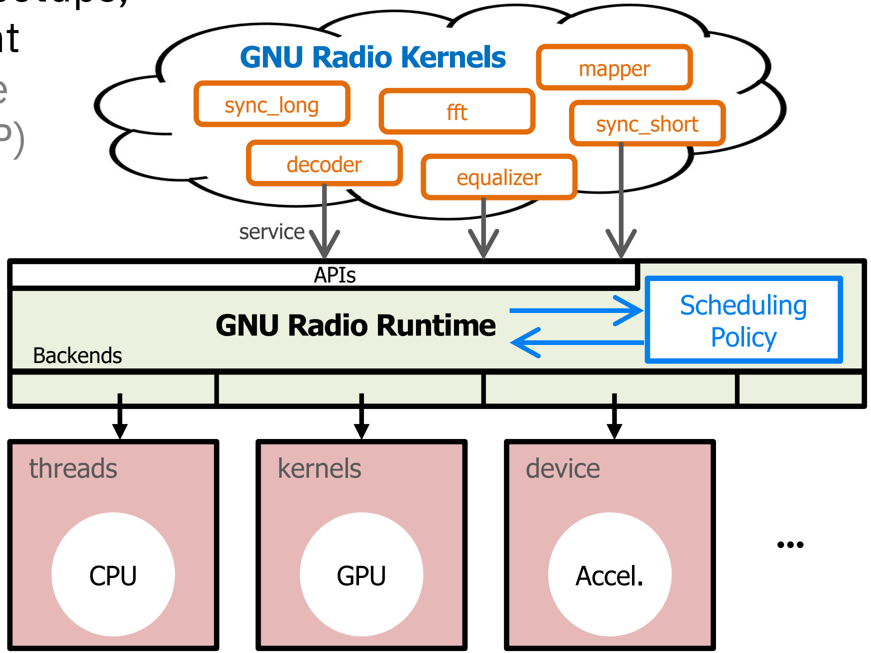
https://github.com/augustojv

Smart Scheduler Roadmap and Big Picture

- **STOMP is only intended for early-stage evaluation of smart scheduling policies**

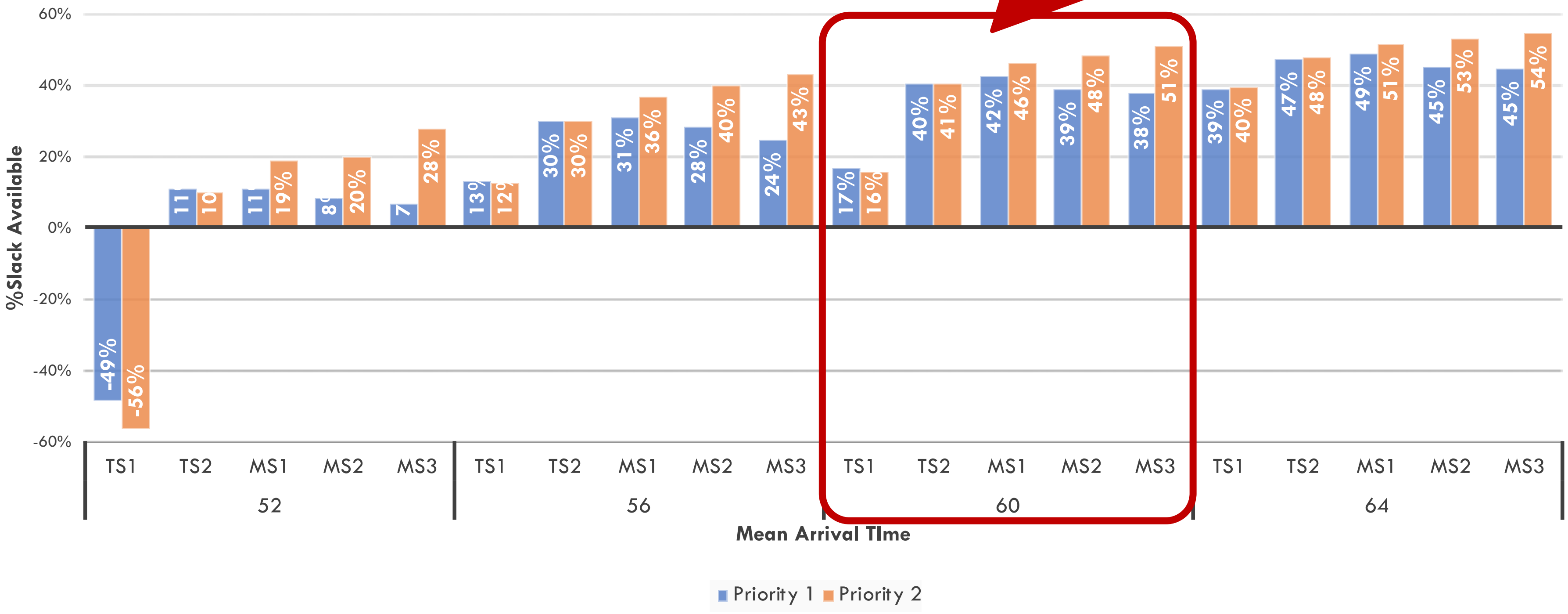
- Ultimately these policies should be ported to real setups, e.g. as part of the GNU Radio run-time environment
  - GNU Radio makes run-time decisions using the specified policy (originally developed in STOMP)

- We can also use existing software middleware frameworks (e.g. OpenCL, OpenMP, OpenSSL) to prototype scheduling policies
  - Target architectures: IBM P9, NVIDIA Xavier



Evaluation: Slack Available

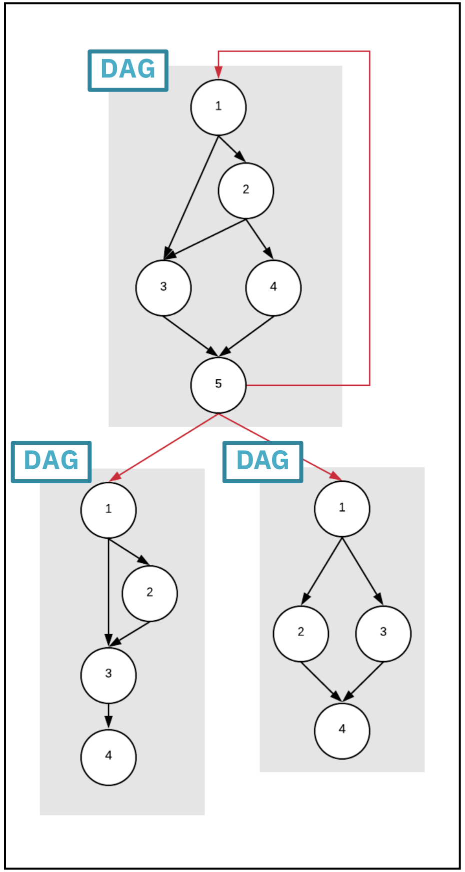
MS3 results in 35% and 10% more slack than TS1 and TS2, respectively



STOMP Inputs

- Domain-specific applications ➜ control flow graphs
- Control flow graphs are divided into directed acyclic-graphs (DAGs) of multiple tasks
  – **Task:** unit of work that can execute on a server (PE)
- DAG trace as input
  – **Compile-time:** applications are known and DAGs are static
  – **Runtime:** DAGs arrive dynamically with variable arrival rate
- Each DAG has real-time constraints associated to it
  – A priority and a deadline
  – Determined at run-time based on the environment and functions of each DAG

Control Flow Graph



Scheduling Mechanism

- When a DAG arrives, META pushes ready tasks to the *ready queue* ordered by rank
  - SCHED then schedules them onto servers (PEs)
- Once a task completes:
  - SCHED pushes it into the *completed queue*
  - Task ID and execution time are passed back to META
  - META pops the completed task and finds its parent DAG
- META checks for resolved dependencies and finds ready tasks, then:
  - Calculates deadline of the new ready tasks
  - Assigns new priority based on the remaining slack
  - Updates rank of ready tasks and re-orders them
  - If remaining slack is negative and task has non-critical priority, drops the DAG