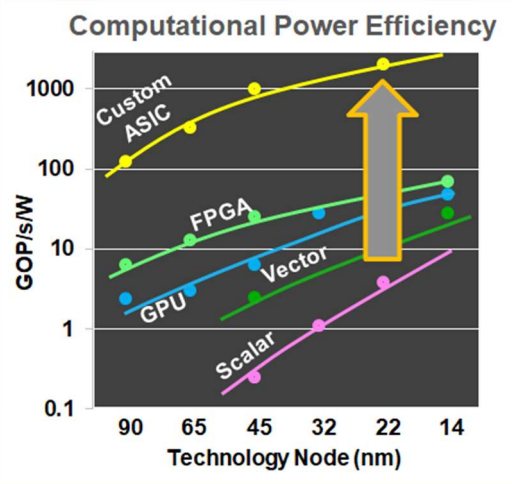
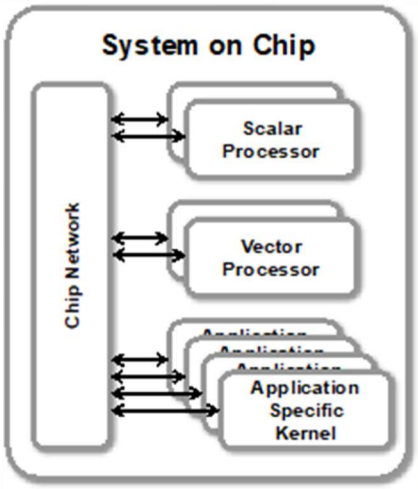
CEDR-API
Productive, Performant Programming of Domain-Specific Embedded Systems

Joshua Mack, Serhan Gener, Sahil Hassan, H. Umut Suluhan, Ali Akoglu

Department of Electrical and Computer Engineering, University of Arizona
{jmack2545, gener, sahilhassan, suluhan, akoglu}@arizona.edu

Motivation





- Heterogeneous computing holds a lot of potential, but it is difficult to effectively leverage
- One approach: domain-specific SoC (DSSoC) processors
  - Restrict the scope of the problem while still enabling large performance gains
- Goal: develop usable, domain-specific, coarse-scale embedded processors as a part of the DARPA DSSoC program

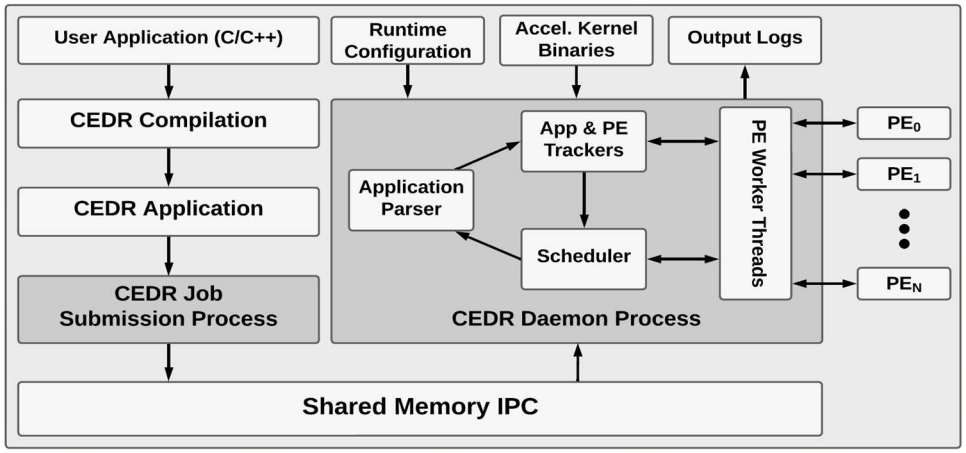
Motivation – DSSoC Programming

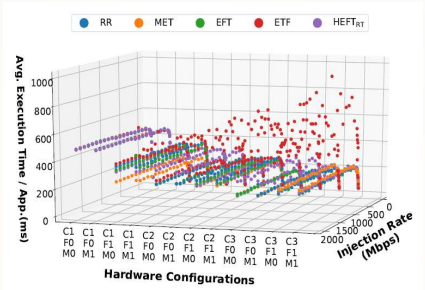
- Many heterogeneous programs are constructed in a vacuum via offline expert analysis
- Static approaches to programming heterogeneous systems are suboptimal
  - Experts are scarce; static schedules are greedy
  - Scheduling preferences can adjust based on power, energy, or execution considerations
- Operating systems aren't meeting the needs of heterogeneous SoCs[1]
- DSSoC platforms require novel compilation and runtime frameworks to enable cooperative heterogeneous computation

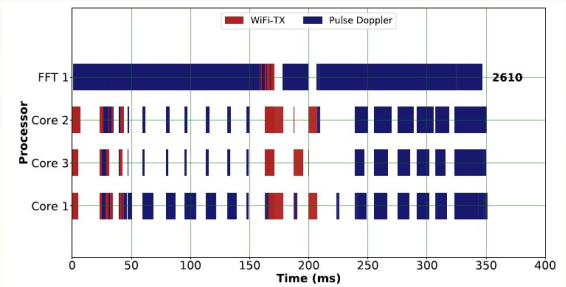
[1] T. Roscoe, “It’s time for operating systems to rediscover hardware,” USENIX Association, Jul. 2021

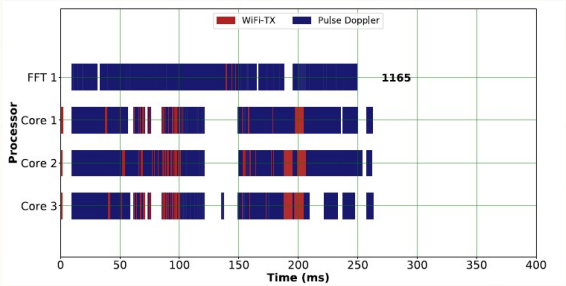
CEDR[1] – A Compiler-Integrated, Extensible DSSoC Runtime

- Open source[2], unified environment for programming and execution on DSSoCs
- Key features:
  - Portability across numerous Linux systems
  - Scalability to thousands of jobs
  - Flexibility to execute arbitrary, interleaved workloads on various accelerators
  - Supports arbitrary scheduling heuristics
- Enables large-scale DSE and codesign of applications, schedulers, and accelerators for DSSoCs





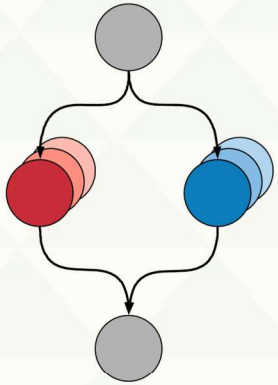


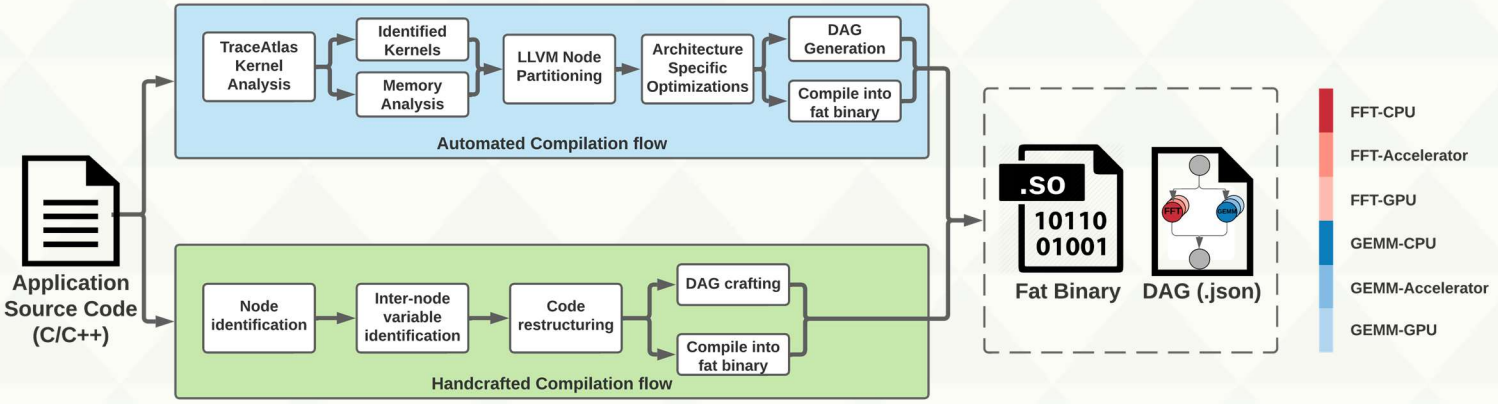


COLLEGE OF ENGINEERING
Electrical & Computer Engineering

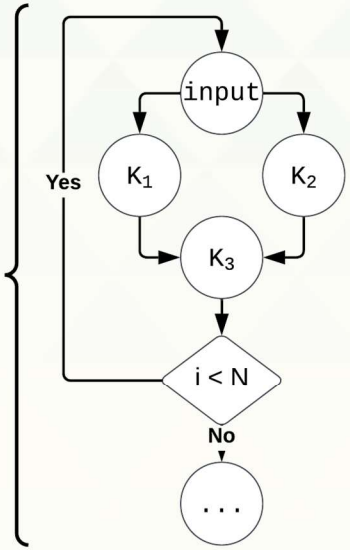
[1] J. Mack et al. “CEDR - A Compiler-integrated, Extensible DSSoC Runtime,” ACM Transactions on Embedded Computing Systems (TECS), April 2022, https://doi.org/10.1145/3529257
[2] Project Homepage: https://ua-rcl.github.io/CEDR/

Limitations of Current Compilation Tooling





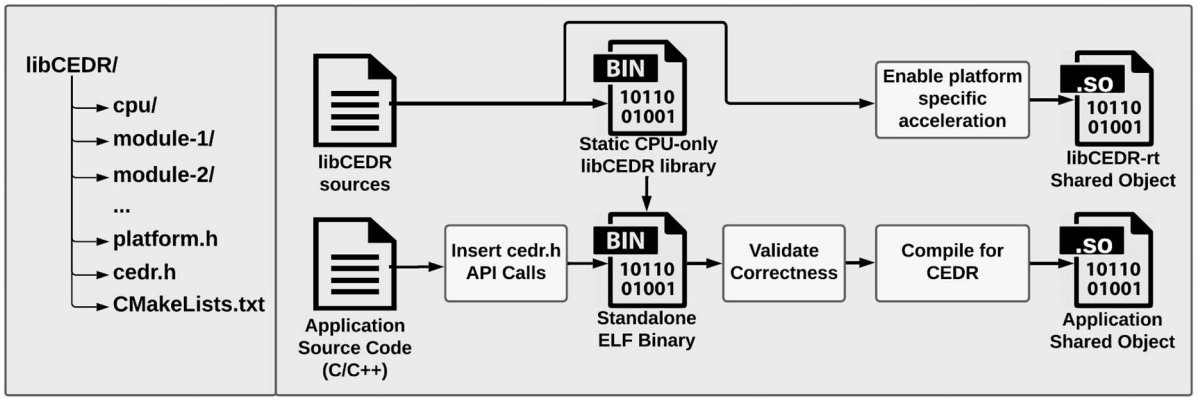

```
...
for (i = 0; i < N; i++) {
   input = read_input();
   out1 = KERNEL1(input);
   out2 = KERNEL2(input);
   KERNEL3(out1, out2);
}
...
```



- Existing software compilation toolchain represents applications as DAGs
  - Automated transformation of C/C++ to DAGs remains a difficult challenge
  - Hand-crafting optimized DAGs is laborious
- DAGs conceptually fail to capture many desirable program structures

libCEDR

```
CEDR_FFT(…);
| -> CEDR_FFT_cpu(…);
| -> CEDR_FFT_accel(…);
| -> …
CEDR_GEMM(…);
| -> CEDR_GEMM_cpu(…);
| -> CEDR_GEMM_accel(…);
| -> …
…
```




```
in1 = get_input();
CEDR_FFT(in1, out1);
func(out1, out2);
std::cout << out2;
```

- Developed a library of hardware-agnostic API calls coupled with platform-specific implementations
- Allows for CEDR-independent CPU-only functional verification
  - Applications can trivially be executed on arbitrary CEDR-compatible platforms by adjusting `libCEDR-rt.so`
- Includes a task synchronization methodology that preserves functional correctness

API Synchronization

User Code

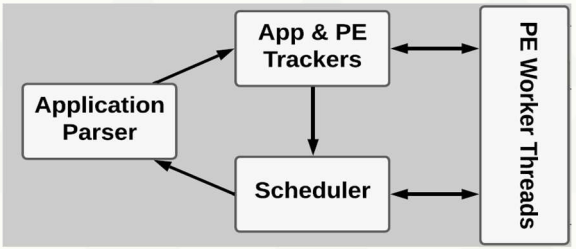
Blocking API Call

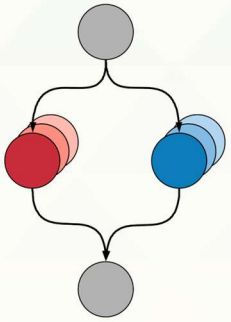
User Code

```
barrier = cedr_barrier_init(NUM_TASKS = 1);

// Dispatch to the CEDR runtime
enqueue_cedr_task(API_TYPE, args, &barrier)
// Task is scheduled and run asynchronously

while (barrier.tasks_done != NUM_TASKS) {
  pthread_cond_wait(barrier.cond);
}
```

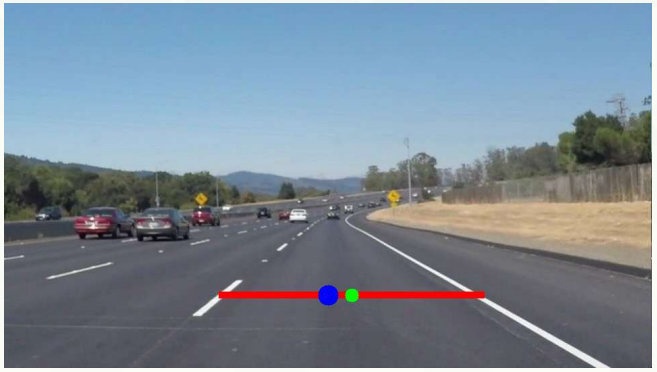




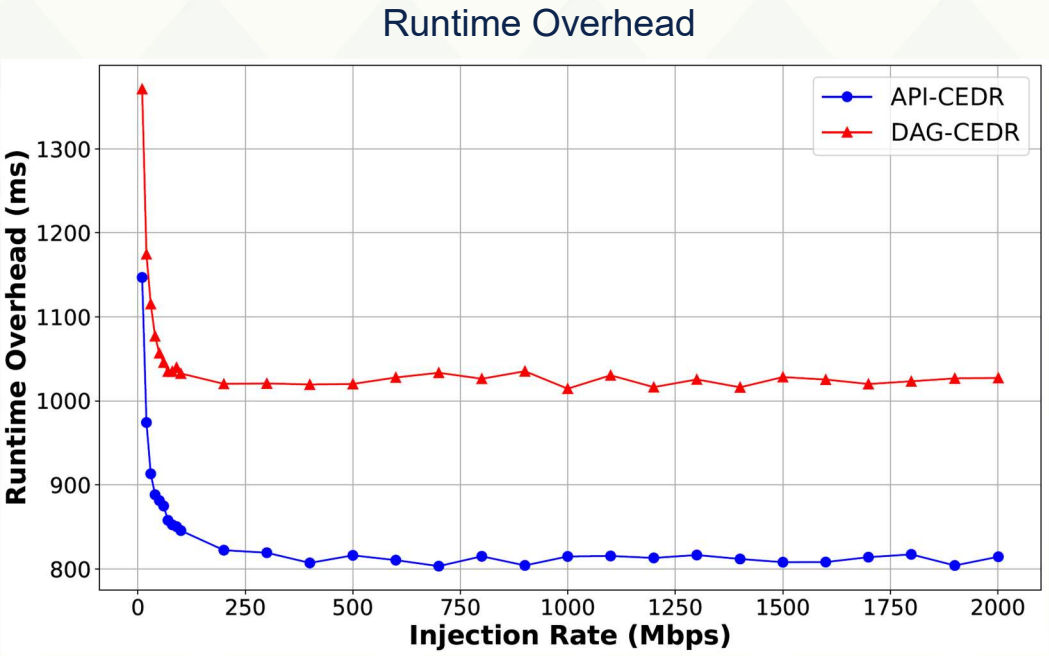
- Kernel executions within CEDR are asynchronous relative to user applications
- Users focused on correctness can leverage **blocking** APIs that abstract away this full process
  - Still enables the benefits of heterogeneous scheduling and dispatch
- Performance programmers can leverage **non-blocking** APIs that enable DAG-like parallelization speedups

Experimental Setup

- Platforms:
  - NVIDIA Jetson AGX Xavier (8 CPU, 1 GPU)
  - Xilinx Zynq Ultrascale+ ZCU102 (4 CPU, FPGA PL)
- Applications:
  - WiFi TX (TX)
  - Pulse Doppler radar (PD)
  - Lane Detection (LD)
  - Swept across injection rates of 100 Mbps to 2000 Mbps
    - Injected in application batches
- Schedulers:
  - Round Robin (RR)
  - Earliest Finish Time (EFT)
  - Earliest Task First (ETF)
  - Runtime variant of HEFT (HEFT-RT)
- All (API-CEDR) applications leverage non-blocking APIs



Results – Runtime Overhead ZCU102



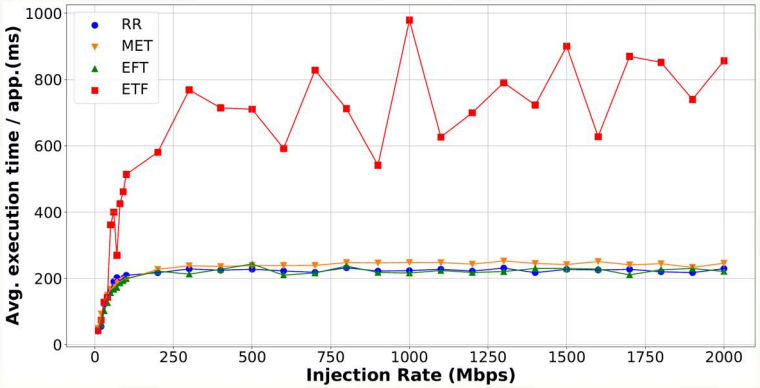
- Workload: 5 TX, 5 PD
- Hardware: ZCU102, 3 CPU, 1 FFT, 1 MMULT
- Scheduler: RR
- API-CEDR: 19.52% average reduction relative to DAG-CEDR
  - Reflects runtime simplifications (no DAG parsing, ready queue simplifications)

Results – Execution Time ZCU102

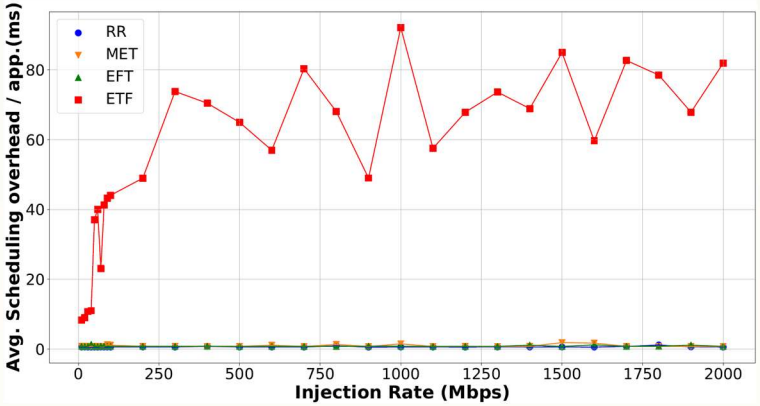
- Workload: 5 PD, 5 TX
- Hardware: 3 CPU, 1 FFT, 1 MMULT
- ETF: 700ms/app reduces to 425ms/app
  - 70ms avg overhead to 1.15ms
  - Quadratic runtime in ready queue size
- Other schedulers: 200ms/app to 350ms/app

DAG-CEDR

Execution Time

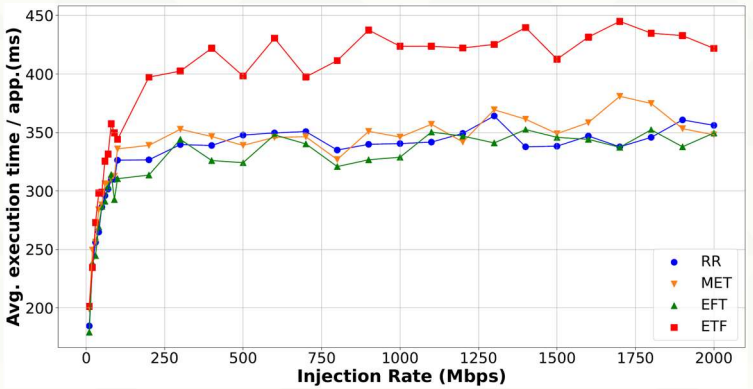


Scheduling Overhead

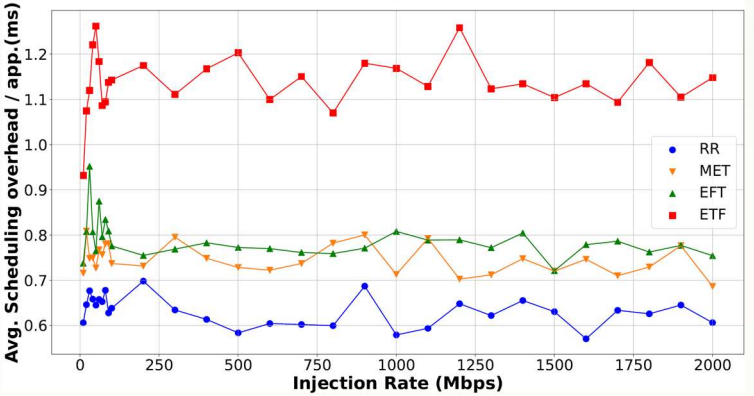


API-CEDR

Execution Time



Scheduling Overhead

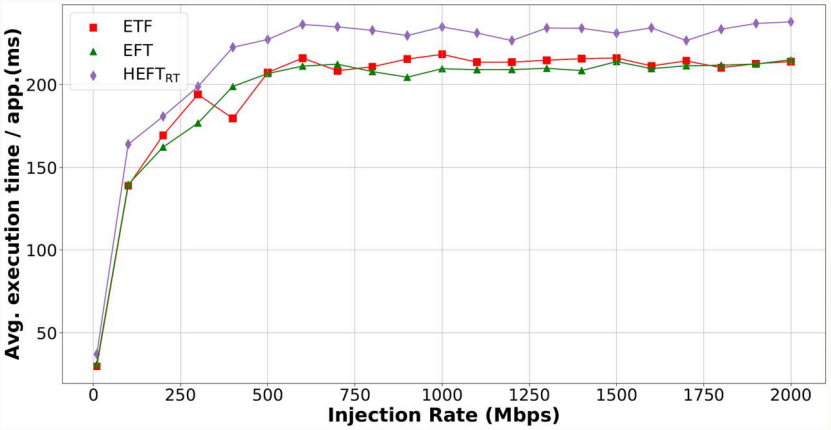


COLLEGE OF ENGINEERING
Electrical & Computer Engineering

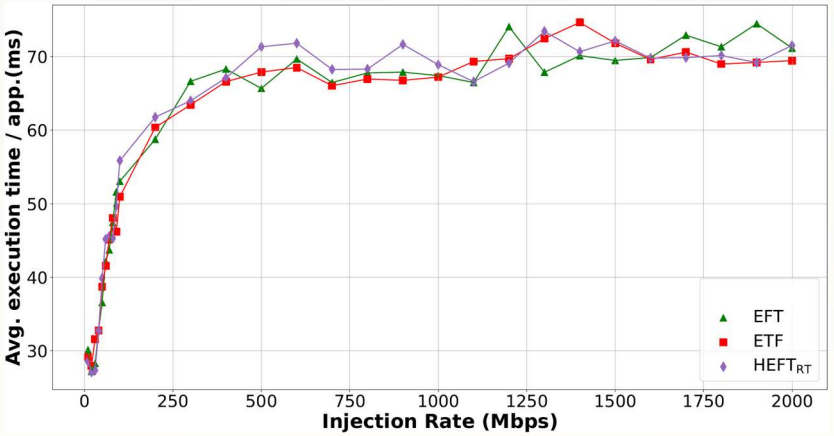
Results – Execution Time AGX

- Workload: 5 PD, 5 TX
- Hardware: 3 CPU, 1 GPU
  - 4 CPUs available for application threads + runtime daemon
- Consistent reduction across all schedulers is likely due to reduced thread contention & reduced runtime overhead

Execution Time – DAG-CEDR

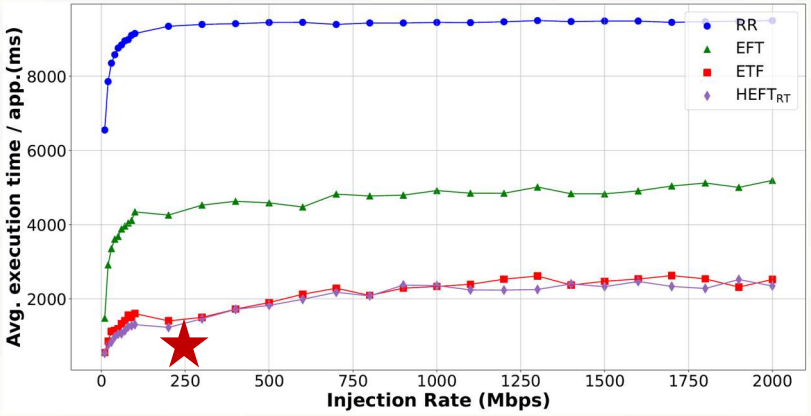


Execution Time – API-CEDR

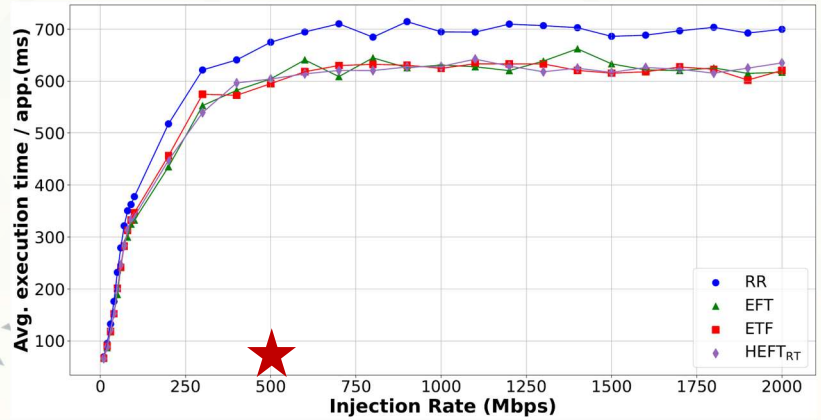


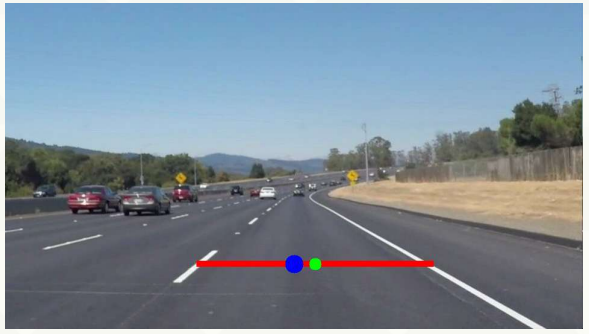
Results – Execution Time with LD

Execution Time - ZCU102



Execution Time - AGX



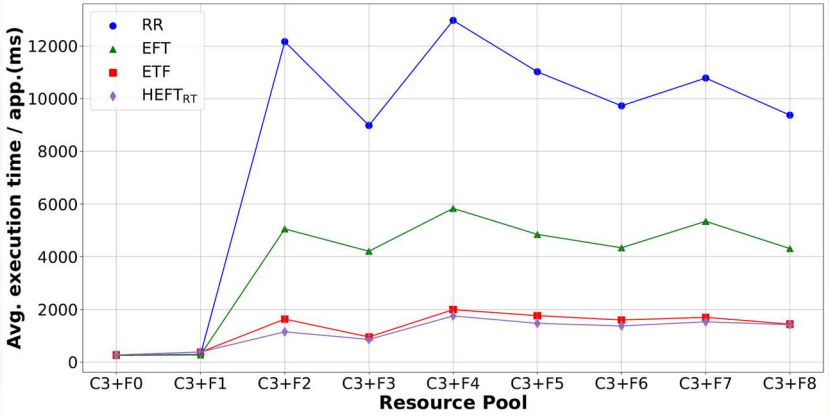


- LD is quite intensive relative to PD, TX
- Workload: 1 LD, 20 PD, 50 TX
- Hardware:
  - ZCU102: 3 CPU, 8 FFT
  - AGX: 7 CPU, 1 GPU
- Complex schedulers outperform simple heuristics
- AGX shows much less variation likely for similar thread-contention reasons

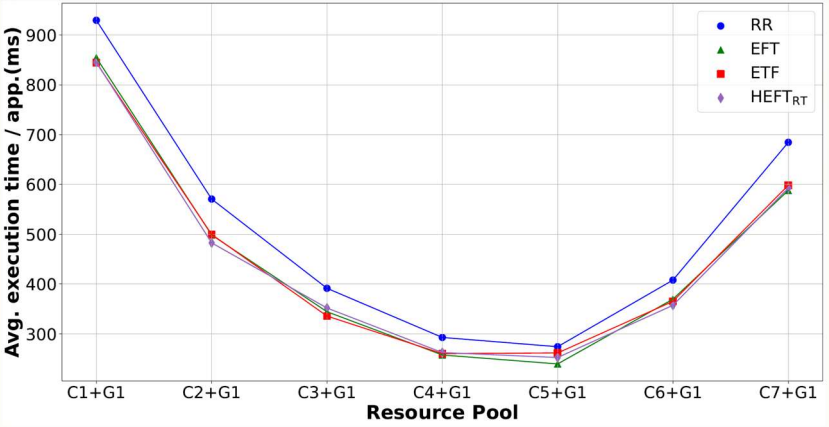
Results – Hardware Scalability

- Workload: 1 LD, injected at 300 Mbps (ZCU102), 500 Mbps (AGX)
- Sweep hardware configuration and scheduler
- Sweep hardware configurations from:
  - ZCU102: 3 CPU 0 FFT through 3 CPU 8 FFT
  - AGX: 1 CPU 1 GPU through 7 CPU 1 GPU
- ZCU102:
  - Increased accelerators increases threads-per-CPU, slowing overall execution
- AGX:
  - Abundance of CPU cores allows for high degree of parallelism
- Other runtime/scheduling overhead differences are negligible

Execution Time – ZCU102



Execution Time – AGX



Conclusions & Future Work

Conclusions

- Presented a new programming methodology for the CEDR framework
- Expanded application corpus to the domain of autonomous vehicles
- Reduced runtime overhead relative to the previous runtime
  - Care must be taken to avoid other threading-related bottlenecks

Future Work

- Explore lightweight, many-core CPU architectures that allow coupling accelerators with cores tailored for accelerator management
- Continue to iterate on runtime architectures that seek to reduce total number of contending threads

Thank You!

Links

- Project Homepage: https://ua-rcl.github.io/CEDR/
- Source Code: https://github.com/UA-RCL/CEDR/

Contact

- Joshua Mack – jmack2545@arizona.edu
- Sahil Hassan – sahilhassan@arizona.edu
- Ali Akoglu – akoglu@arizona.edu

Questions?