Deterministic Modeling and Simulation of Fault-Tolerant Real-Time Software

Dongha Kim and Hokeun Kim

School of Computing and Augmented Intelligence

Arizona State University

Time-Centric Reactive Software (TCRS '25) Workshop
at ESWEEK 2025, Taipei, Taiwan, on October 2, 2025

Contact:

- dongha@asu.edu
- hokeun@asu.edu

Websites:

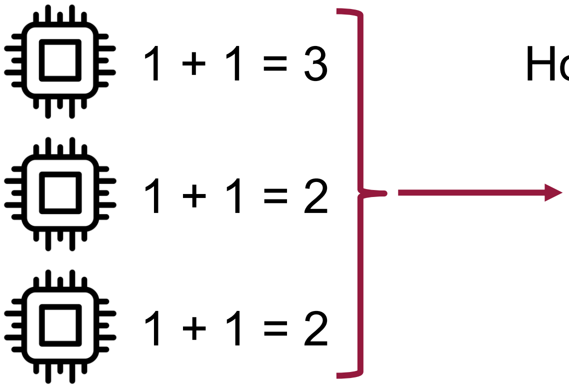
- https://labs.engineering.asu.edu/kim/
- https://jakio815.github.io/
- https://hokeun.github.io/

Introduction

- Some real-time systems have **hard time requirements**.
- Even if logic was designed flawless, **hardware faults can occur**. E.g., Soft errors
- Qantas Flight 72 (2008) [1] – A **single bit error** in one of the air data inertial reference units (ADIRU) caused the autopilot to dive the aircraft, resulting serious injuries.
- **How do we make them fault tolerant?**

[1] Bureau, Australian Transport Safety. "In-Flight Upset, 154 km West of Learmonth." Western Australia 7 (2008).

Hardware techniques?



How does the processor know this is wrong?

Pick the **most common** outcome

- Add processors doing same job

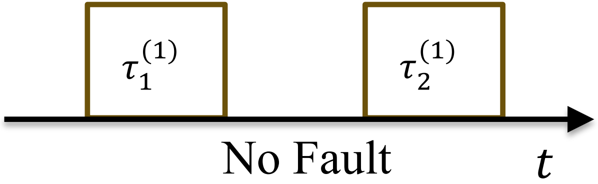
  -> Hardware techniques **require additional hardware components**.
  -> Hardware being complex, **increasing hardware fault rates** [2].

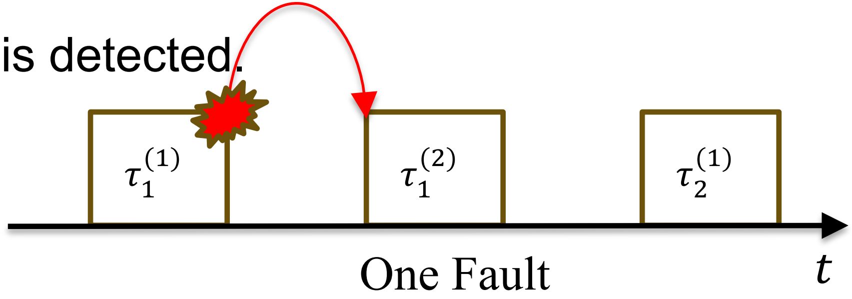
[2] Canal, Ramon, et al. "Predictive reliability and fault management in exascale systems: State of the art and perspectives." *ACM Computing Surveys (CSUR)* 53.5 (2020): 1-32.

Time redundancy fault tolerance [3]

1. Re-execution

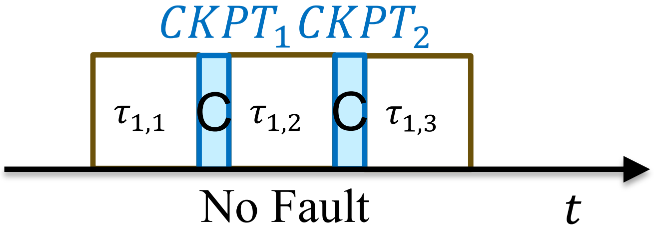
-> Restart the same task when failure is detected.

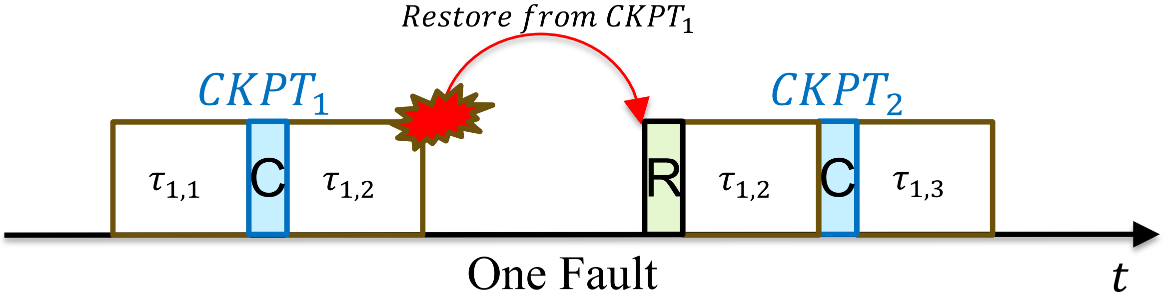




2. Checkpoint / Restart (Restore)

-> Create a checkpoint, which saves the state of the task, and restarts from the checkpoint.





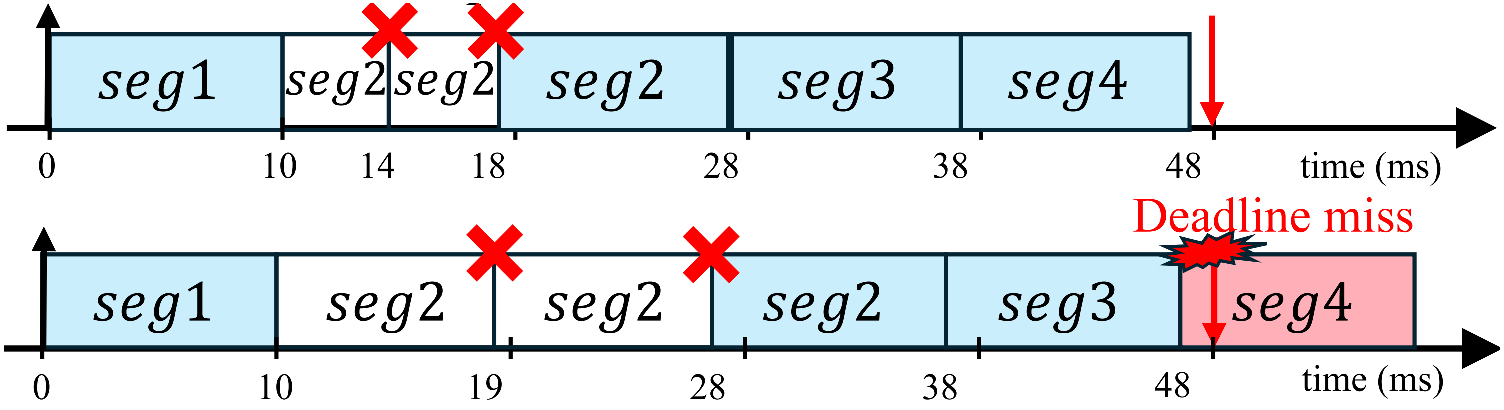
[3] F. Reghenzani, Z. Guo, and W. Fornaciari, "Software fault tolerance in real-time systems: Identifying the future research questions," ACM Computing Surveys, vol. 55, no. 14s, pp. 1–30, 2023.

Tradeoff Between Timeliness and Reliability

- Enhancing fault tolerance through re-execution or checkpointing **can negatively impact schedulability** due to fault detection and recovery.
  -> Deadlines *can* be missed!
- However, **testing this is not easy**.
  Especially when we want deterministic results.
- We want to ensure if deadlines always miss, or not.
  -> We want deterministic results!
- However, failures can lead to non-deterministic results…

Motivating Example

- e.g.) Task split into 4 segments (3 checkpoints)



- Same number and sequence of failures can lead to **different deadline behaviors depending on failure timing**.

  -> Unreliable schedulability analysis.

  -> No repeatability

- ***How should we verify that scheduling including fault tolerance techniques meet their deadlines with determinism?***

Tasks Models, Assumptions and Timing Semantics

Task Model

$\tau_{i,j:[k]}^{(n)}$

- i : Task number
- j : Instance number
- k : Segment number
- n : Number of executions of segment
- $C_{i:[k]}^{F}$: Segment's worst case execution time (WCET) including Failure detection and recovery.

System and Failure Assumptions

- Scheduling is **weakly hard real-time** and **non-preemptive**.
- Failures can **occur in any segment**.
- Watchdogs **detect all failures**.
- Detection/recovery **add small time overhead**.
- **No failures during** failure detection and recovery.
- Each task **can abort at checkpoints**.

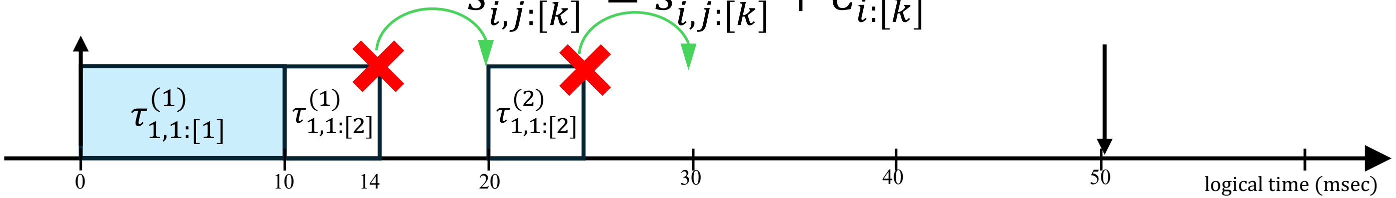
Timing Semantics

- Physical Time: Wall clock time.
- Logical Time: Abstraction of ordering of events.
- **Logical Execution Time (LET):** Abstraction of actual execution time.

Approach: Advancing Logical Time

- When segment fails, **advance the logical time** as much as the WCET $C^{F}_{i:[k]}$ of the segment.

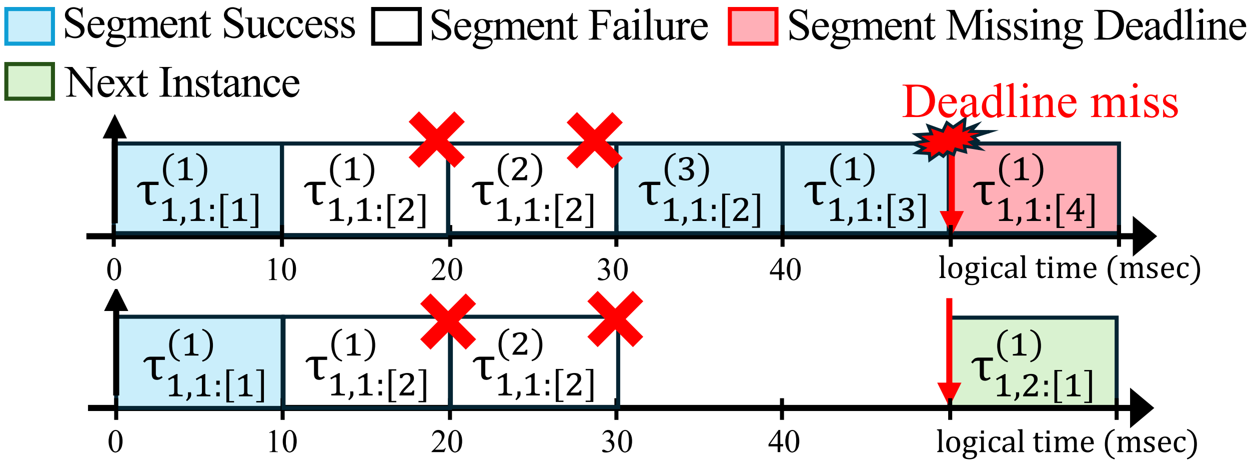
$$s^{(n+1)}_{i,j:[k]} = s^{(n)}_{i,j:[k]} + C^{F}_{i:[k]}$$



- Ensure determinism.
  - Results only depend on the ***sequence of failures (number, order)***, not their ***timing***.
  - Motivational example leads to two different results, which is non-deterministic.
    -> We guarantee the system fails or succeeds deterministically.
  - Limitations: This approach is very conservative.

Approach: Proactive Task Instance Abortion

- Monitor both task deadlines, and the cumulative execution time.
- **Abort instances** if they can no longer meet deadlines.

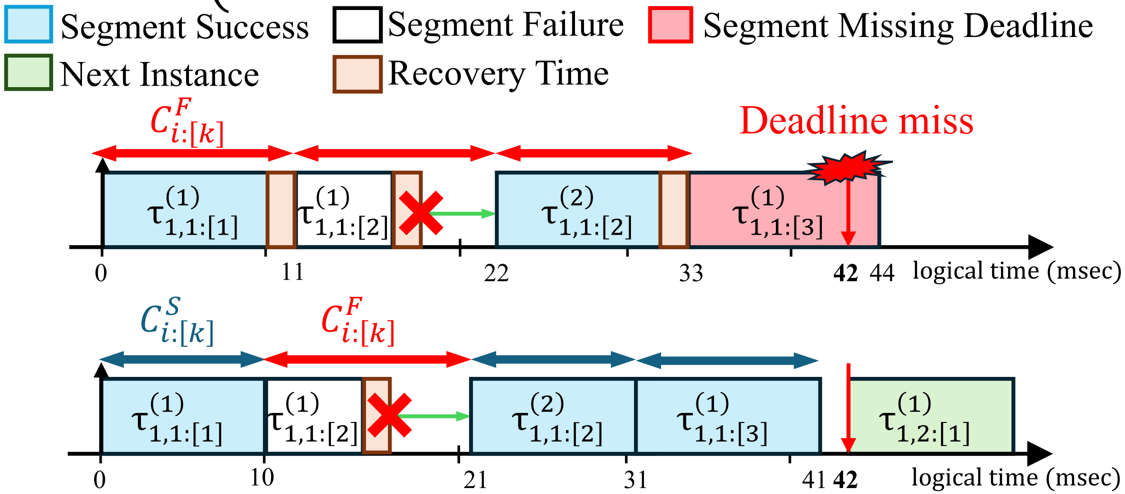


- Avoid utilization waste.
- Prevent deadline misses propagating to next instances.
- Start next instance on time (can be critical when data freshness is important)

Proposed: Enhancement to Execution Model

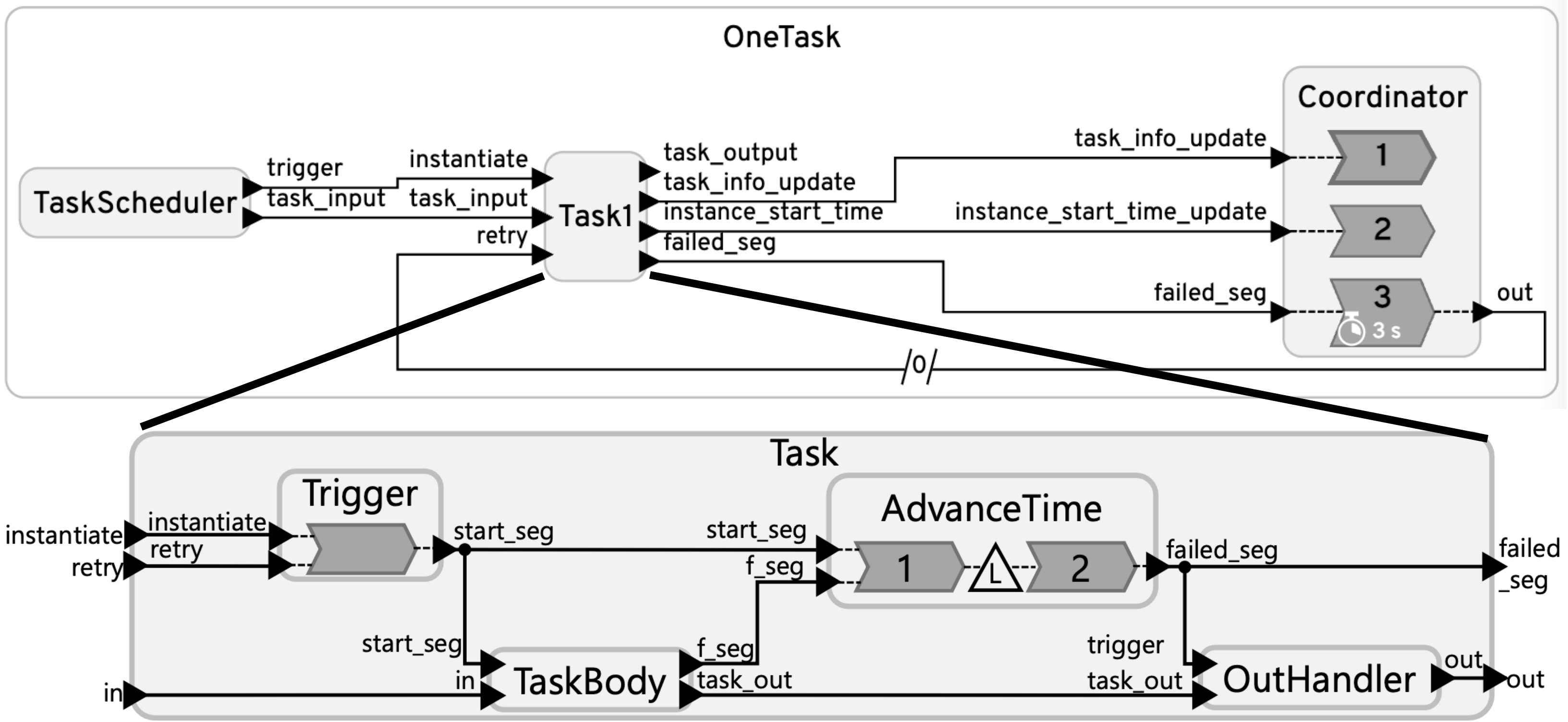
- Advancing $C^{F}_{i:[k]}$ is conservative, which can lead to many deadline misses.
- Inefficiency -> System always advances logical time including recovery time.
- New approach: Distinguish WCET as **Succeed** ($C^{S}_{i:[k]}$) and **Failure** ($C^{F}_{i:[k]}$).
- **Succeed WCET** ($C^{S}_{i:[k]}$): Exclude failure detection and recovery time.

$$f^{(n)}_{i,j:[k]} = \begin{cases} s^{(n)}_{i,j:[k]} + C^{S}_{i:[k]}, & (segment\ succeeds) \\ s^{(n)}_{i,j:[k]} + C^{F}_{i:[k]}, & (segment\ fails) \end{cases}$$



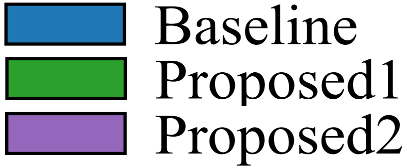
Runtime Design

LINGUA FRANCA: coordination language for deterministic, time-sensitive programs.



Evaluation

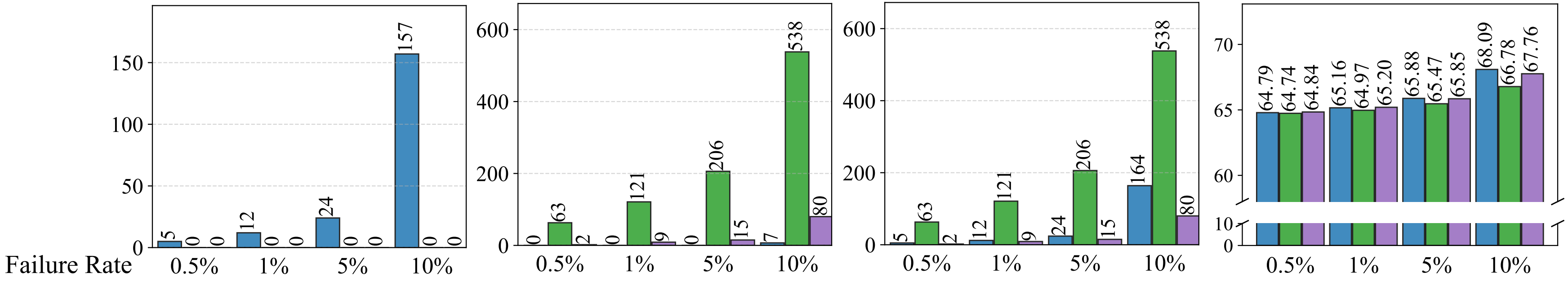
- Use same task, actual execution time is uniformly sampled (80% to 100%).
- 10,000 runs with failure rate: 0.5%, 1% 5% and 10%.



: Re-executes failed segment

: Advances logical time as much as $C^{F}_{i:[k]}$

: Advances logical time depending on success ($C^{S}_{i:[k]}$) and failure ($C^{F}_{i:[k]}$)



(a) Deadline misses.

(b) Execution failures.

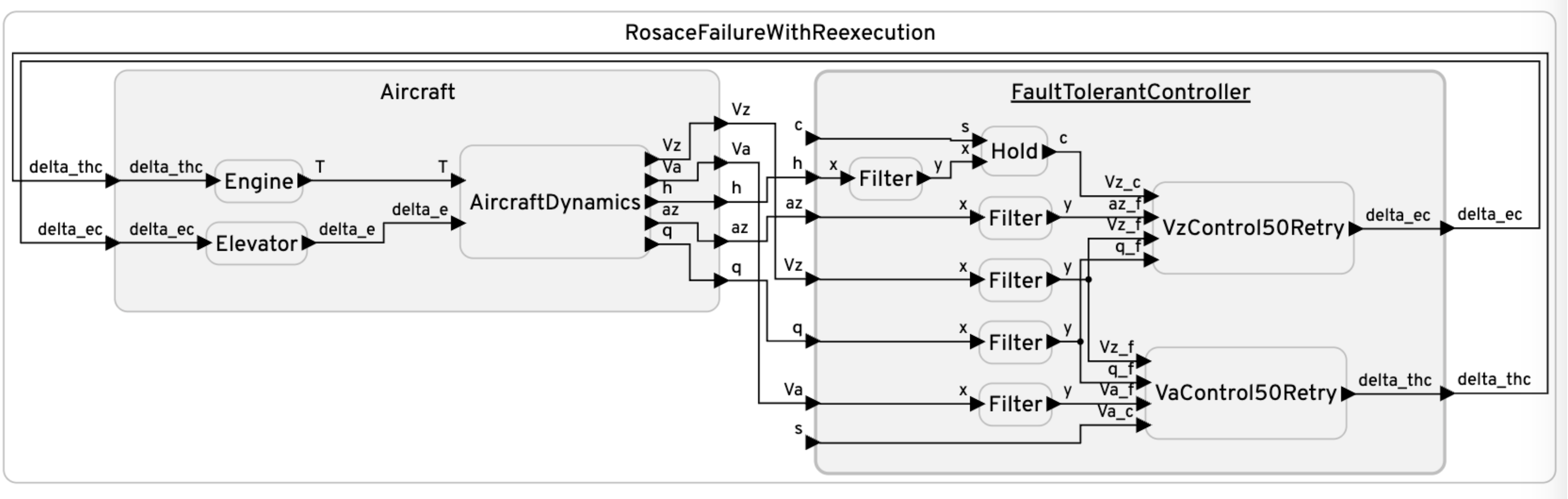
(c) Sum of deadline misses and execution failures.

(d) CPU utilization (%).

- Proposed2 approach **has a smaller number in the overall failures** than the Baseline, which are the sum of deadline misses and execution failures.

Case Study : ROSACE Benchmark

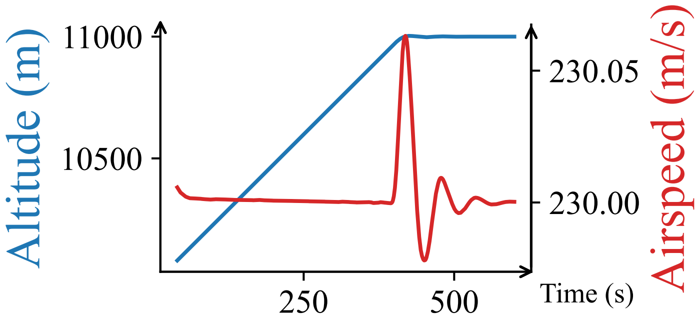
- **ROSACE**: Research Open-source Avionics and Control Engineering [2][3][4]



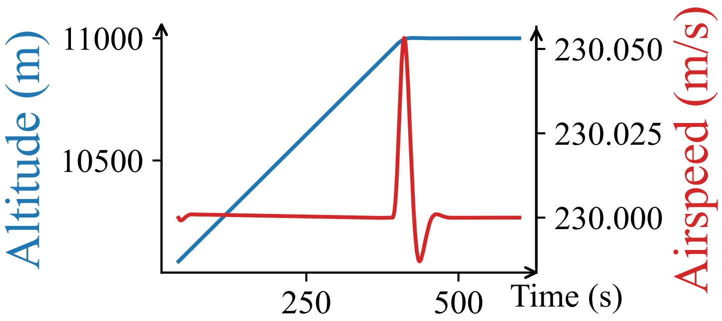
[2] C. Pagetti, D. Saussi´e, R. Gratia, E. Noulard, and P. Siron, “The ROSACE case study: From Simulink specification to multi/many-core execution,” in 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2014, pp. 309–318.
[3] H. Deschamps, G. Cappello, J. Cardoso, and P. Siron, “Coincidence problem in CPS simulations: the R-ROSACE case study,” in 9th European Congress Embedded Real Time Software and Systems ERTS2 2018, 2018, pp. pp–1.
[4] E. A. Lee, D. Saussie, and C. Pagetti, “Aircraft controller – the ROSACE case study,” https://github.com/lf-lang/playground-lingua-franca/tree/main/examples/C/src/rosace, Lingua Franca Playground.

Case Study : ROSACE Benchmark

- **Inject failures at 40% rate** into the true airspeed (V_a) and vertical speed (V_z) controller.



(a) Original ROSACE software with failure injection.



(b) Modified ROSACE for fault tolerance simulation with failure injection and Proposed2.

- In (a), the aircraft oscillates and destabilizes under faults.
- In (b), the aircraft recovers quickly and maintains stable.

Summary

- **Deterministic execution models**
  - Ensure determinism in fault-tolerant real time systems.
- **Simulation runtime**
  - Implemented using Lingua Franca (LF) to support realistic software-level simulations
- **Validated performance**
  - Experiments and ROSACE case study show deadline misses are avoided and utilization waste reduced.

https://github.com/asu-kim/fault-tolerant-real-time

Authors:
Dongha Kim, and Hokeun Kim
Contact:
dongha@asu.edu, hokeun@asu.edu

Websites:
- https://labs.engineering.asu.edu/kim/
- https://jakio815.github.io/
- https://hokeun.github.io/