Fault Tolerance on Control Applications: Empirical Investigations of Impacts from Incorrect Calculations

Mikail Yayla, Kuan-Hsun Chen, Jian-Jia Chen

Department of Computer Science
Technical University of Dortmund, Germany

April 10, 2018

This research has been supported by DFG, as part of the Collaborative Research Center SFB876 subproject B2.

Outline

- Introduction
- System and Fault Model
- Studied Application with Fault Injection
- Empirical Approach for Obtaining (m,k)
- Experiment Results and Evaluation

Introduction (1/2)

High transient fault rates expected in future systems

- Corrupt execution state or cause soft-errors through bitflips
- Lead to catastrophic failures, e.g., JAXA satellite Hitomi

Software-based fault tolerance approaches

- No need for special hardware
- Higher utilization

Probability of faults is low

- Fully protecting everything is mostly over-provision
- Reduction of utilization is possible

Introduction (2/2)

Some errors can be tolerated in control applications

- If we allow to downgrade the quality of control
- High quality system with low utilization is desirable

Offline and online soft-error handling techniques

- Offline selective protection
- Online detection and compensation

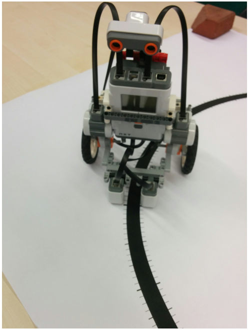
We need an approach to

- Quantify (m,k) robustness

Path Tracing Experiment[1]

LegoNXT path-tracing robot

- Only one task with two light sensors
- On a circular track

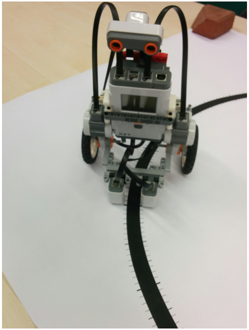


[1]Chen et al., *"Compensate or Ignore? Meeting Control Robustness Requirements through Adaptive Soft-Error Handling"*, LCTES 2016

Path Tracing Experiment[1]

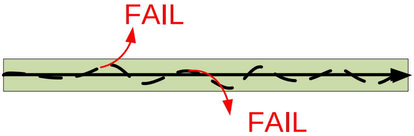
LegoNXT path-tracing robot

- Only one task with two light sensors
- On a circular track



Go forward and follow the path

- Decision is made for jobs to have error
- More steering
- Leaving the track: <u>mission failure</u>



[1] Chen et al., "*Compensate or Ignore? Meeting Control Robustness Requirements through Adaptive Soft-Error Handling*", LCTES 2016

System Model and Notation

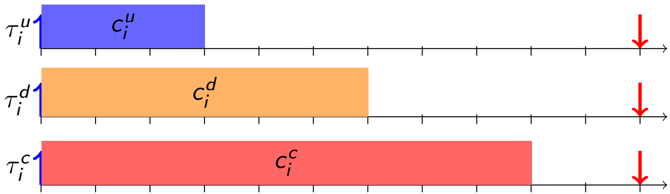
Robustness requirement

- Each task τ_i has a robustness requirement (m_i, k_i)
- m_i out of any k_i consecutive jobs have to be correct

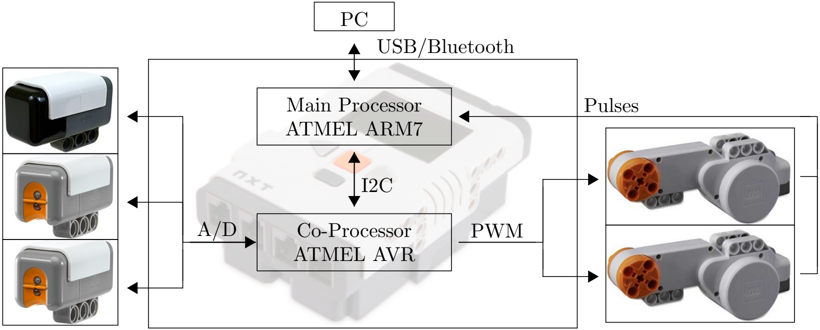
Example for (m,k)

- The bit sequence $\{1, 1, 0, 1, 1, 0, 1, 1\}$ fulfills $(2, 3)$
- If $\{0, 0, 1\}$ occurs, (2,3) is not fulfilled

Soft-error (↯) handling on task Level



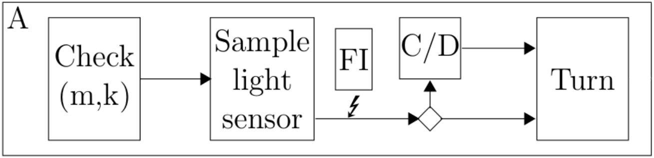
Lego NXT Robot and nxtOSEK



Original and Extended Fault Injection

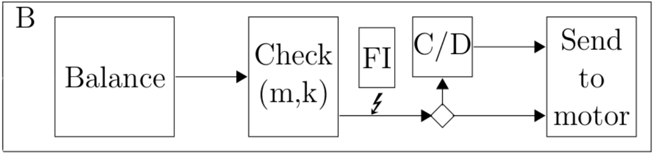
Original Design: Path tracing task

- Fault injection only in light sensor values

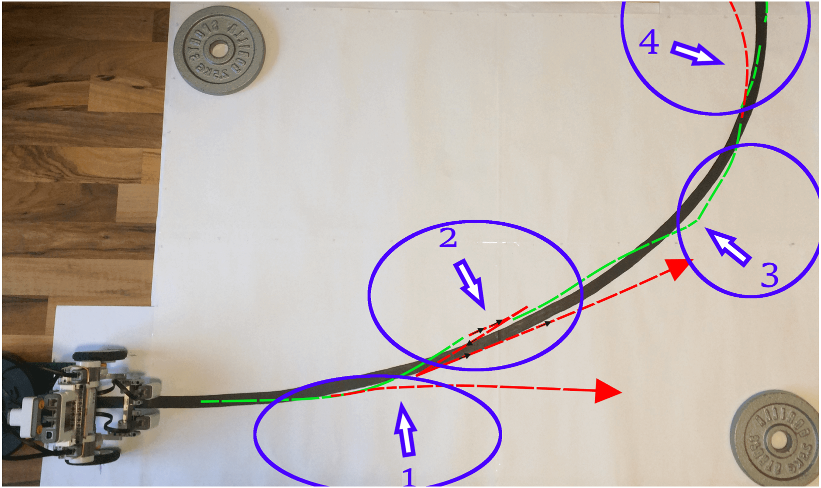


Extended Design: Balancing task

- Fault injection in light sensor and motor input values



Experiment Setup: Path Tracing and Balance



General Experiment Flow: Obtaining (m,k)

Inject faults with f_{max}

Find (m,k) candidates

Verify (m,k) candidates

Evaluate soft-error handling techniques

- Goal: show that (m,k) prevents mission failure
- Find the best (m,k) requirement: lowest ratio $\frac{m}{k}$
- Using (m,k) we can save utilization and protect the system

Fault Injection

Inject faults with f_{max}	Find (m,k) candidates	Verify (m,k) candidates	Evaluate soft-error handling techniques

- f_{max} is the maximum tolerable fault rate
- Running system with f_{max} never causes mission failure
- Any $f_{fail} > f_{max}$ leads to mission failure
- Min. number of correct instances

Finding (m,k) Candidates

Inject faults with f_{max}	Find (m,k) candidates	Verify (m,k) candidates	Evaluate soft-error handling techniques

- Goal: Min. number of correct instances in a sliding window
- Configure system with f_{max} and record bit sequence
- Example: $\{1, 1, 0, 1, 1, 0, 1, 1\}$
  - k = 2: (1,2)
  - k = 3: (2,3)
  - k = 4: (2,4)
- List of (m,k) candidates: <u>(2,4)</u>, (1,2), (2,3)

Verifying (m,k) Candidates

Inject faults with f_{max}	Find (m,k) candidates	Verify (m,k) candidates	Evaluate soft-error handling techniques

- Goal: Verify that (m,k) prevents mission failure
- Configure system with $f_{fail} > f_{max}$ and first (m,k) in list
- Run system with (2,4) and f_{fail}: system executes $\{0, 0, 1, 1\}$
  ↳ Runs without mission failure: Success!
  ↳ If mission failure, try next best requirement (1,2)

Finding (m,k) for Path Tracing and Balancing

Path tracing (m,k)

Fault rate %	20	25	30	35	40
(i,j)	(5,16)	(5,16)	(5,20)	(5,20)	(5,20)

Balancing (m,k)

Fault rate %	10	12	14	16
(i,j)	(11,16)	(10,16)	(10,16)	(8,16)

Verifying (m,k) for Path Tracing and Balancing

Path tracing (m,k)

(m,k)	(5,20)	(4,20)
$f = 60\%$	Y	Y
$f = 80\%$	Y	N
$f = 100\%$	Y	N

Balancing (m,k)

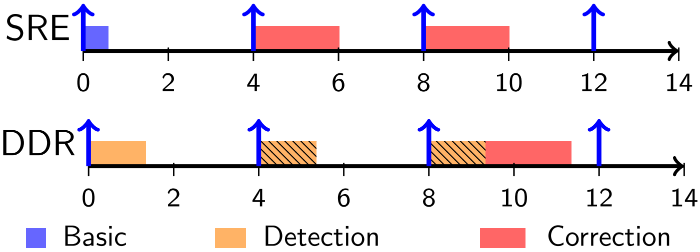
(m,k)	(16,16)	(14,16)	(11,16)	(10,16)
$f = 25\%$	Y	Y	Y	N
$f = 50\%$	Y	Y	N	N
$f = 100\%$	Y	N	N	N

Overall Utilization (1/4): Compensation Techniques

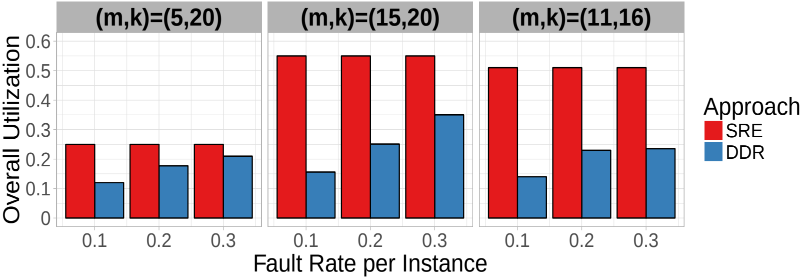
Inject faults with f_{max}	Find (m,k) candidates	Verify (m,k) candidates	Evaluate soft-error handling techniques

Static Reliable Execution, Dynamic Detection and Recovery

- Example: specify system with (2,3) requirement and $\{0, 1, 1\}$



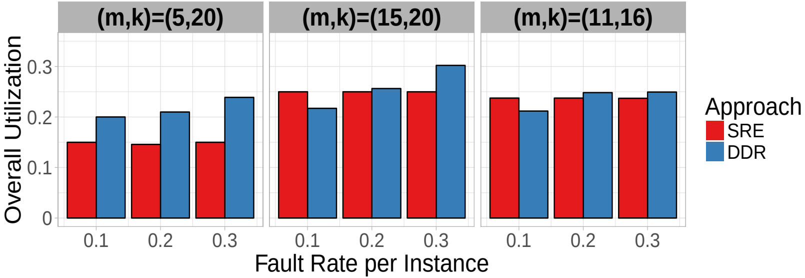
Overall Utilization (2/4): Task Configurations



Task execution times are specified as:

- $c_i^c = 700\ \mu s$, $c_i^d = 120\ \mu s$, and $c_i^u = 100\ \mu s$
- DDR always outperforms SRE

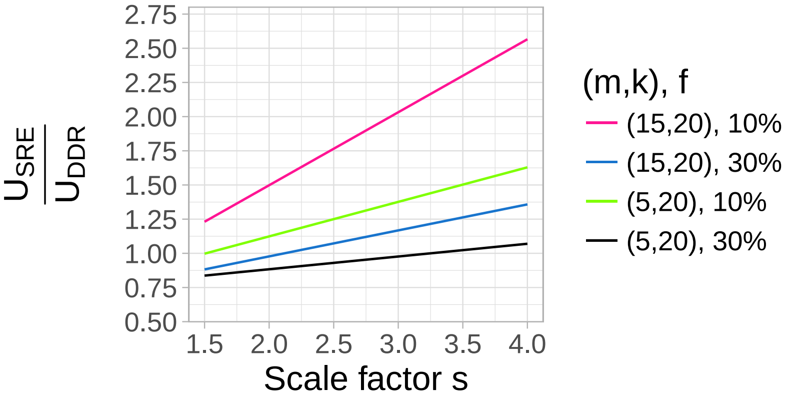
Overall Utilization (3/4): Task Configurations



Task execution times are specified as:

- $c_i^c = 300\ \mu s$, $c_i^d = 200\ \mu s$, and $c_i^u = 100\ \mu s$
- In some cases SRE dominates DDR

Overall Utilization (4/4): Scale Factor



The scale is the deciding factor

- If the scale factor $s = \frac{c_i^c}{c_i^d}$ is low, SRE can be better than DDR

Conclusion and Outlook

Limitations

- No mathematical theory for deriving (m,k)
- Soft-errors are ideally detectable and recoverable
- Only one platform for experiments

Future Work

- Implement fault tolerance techniques in RTEMS
- Obtain (m,k) on Raspberry PI running RTEMS
- Mathematical theory to derive (m,k) for predefined controllers

Thank you!

Concurrent fault injection into Motors and Sensors

Inject faults in path tracing and balance task realistic

- (5, 20) and $f = 60\%$, (11, 16) and $f = 25\%$
  ↳ Robot fails in 10% of all runs
- Same experiment setting with (8, 20) and (13, 16)
  ↳ Success
- If (m, k) requirements are enforced concurrently, m may need to be increased to prevent mission failure

Overall Utilization Equations

Static Reliable Execution

- $\mathbb{U}_{\text{SRE}} = \frac{u \cdot c_i^u + r \cdot c_i^c}{p \cdot T_i}$

Dynamic Detection and Recovery

- $\mathbb{U}_{\text{DDR}} = \frac{c_i^d \cdot (d - r) + c_i^c \cdot (d + r)}{p \cdot T_i}$