Automatic Recovery From Runtime Failures

By Antonio Carzaniga, et al.

Presented By Ben Cheung and Vishnu Prasad

Motivations

- Amazon Store Server Crash
- Day Trading

- There are bugs everywhere, some known, some unknown.
  - “Whatever can go wrong, will go wrong” - Murphy’s Law
  - When the unknown bugs strike, will you be ready? Will you be safe?

Background - Previous Related Work

- Running copies of the system for fault tolerance
  - “The N-version approach to fault-tolerant software”, A. Acizienis
  - “System structure for software fault tolerance” , B. Randell

- Expensive to implement
- Inefficient because of correlating faults

Background - Previous Related Work

Other works addressed issues in specific areas.

- Data Structures
- Configuration Incompatibilities
- Infinite Loops

- Too Direct, not general enough of a technique.

Research Questions

- Is there a way to correct or avoid runtime errors on the fly?
- Is this possible to do without incurring a large overhead time?
- Is it possible to do this generally?

Is there a way to correct runtime errors on the fly?

- Libraries are redundant
- Exploit these redundancies to find workarounds
- Replace error-causing code with workarounds

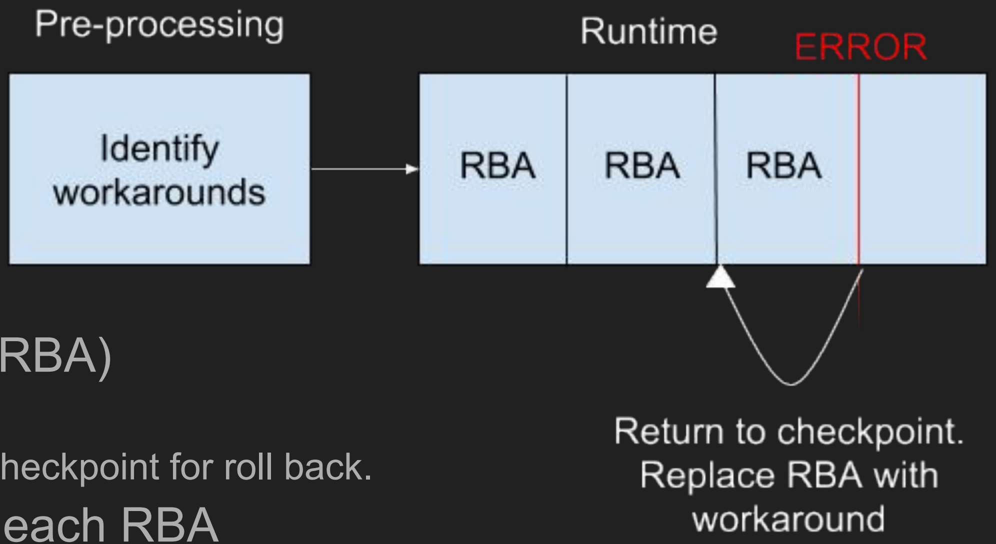
What is a Workaround?

- Semantically equivalent code
- Different implementation
- Product of redundancy
- Identified manually

Simplified Example

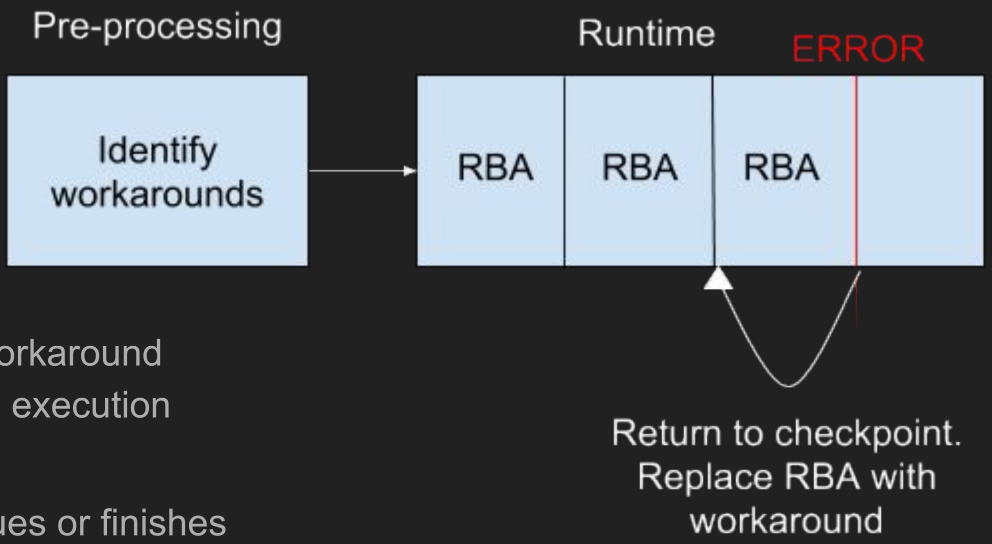
```
//Two semantically identically methods that might exist in a library

  getA(int first, int second){
    int index = (first + second) % 10;
    return array[index];
  }

  getB(int first, int second){
    int index = ( (first % 10) + (second % 10) ) % 10;
    return array[index];
  }


  List.getA(2147483646, 10); // fails

  List.getB(2147483646, 10); // replacement succeeds
```

Preprocessing Step



1. Identify Roll-back Areas (RBA)
    a. Library Calls
    b. Each RBA will be a checkpoint for roll back.
2. Prepare workarounds for each RBA

Runtime Step



1. Checkpointing at RBAs
2. When error is thrown:
    a. RBA replaced with an unused workaround
    b. Rollback to checkpoint, continue execution
3.
    a. No more errors: program continues or finishes
    b. No more workarounds to try: program ends unsuccessfully

Overhead Cost?

- Checkpointing costs
- Rollback, replacement costs
- Re-doing execution

Evaluation - Setup and Problem

- ARMOR System
- 2 Libraries: JodaTime, Guava
- Used on 4 Applications:
    - Fb2pdf
    - Carrot2
    - Caliper
    - Closure

Armor Pre-processing

- 63 Rewriting rules for Guava
- 100 Rewriting rules for JodaTime

TABLE II

RESULTS OF THE PREPROCESSING ON THE SELECTED APPLICATIONS

Application	*Caliper*	*Carrot2*	*Closure*	*Fb2pdf*
Total RBAs	130	139	2099	17
RBAs with variants	60	106	687	17

Effectiveness

TABLE III

MUTATION ANALYSIS AND EFFECTIVENESS OF ARMOR

				Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Total mutants				21297	21297	21297	16858
Relevant mutants				309	187	344	2200
execution	*success*	*equivalent*		210	120	177	1805
		non-equivalent	*detected*	0	2	0	0
			not detected	0	8	3	1
	loop	*detected*		0	1	0	0
		not detected		12	9	15	47
	error			87	47	149	347
Total mutants run with ARMOR				87	50	149	347
Mutants where ARMOR is successful				(28%) 24	(48%) 24	(47%) 70	(19%) 67

- 19%-48% Effective
- Avoiding Runtime Errors is possible

Runtime Overhead

TABLE IV
OVERHEAD INCURRED BY ARMOR IN NORMAL NON-FAILING EXECUTIONS (MEDIAN OVER 10 RUNS)

		Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Time (seconds)	Original total running time	30.13	2.43	5.40	2.26
	Exception-handling only (no checkpoints)	(1%) 30.41	(69%) 4.15	(95%) 10.53	(68%) 3.79
	Snapshot-based checkpoints	(5%) 31.78	(117%) 5.32	>1h	(121%) 4.99
	Change-log-based checkpoints	(2%) 30.87	(94%) 4.75	(194%) 15.90	(114%) 4.70
Memory (MB)	Original total memory allocated	1.40	8.87	30.56	17.90
	Snapshot-based checkpoints	12.30	23.78	—	90.94
	Change-log-based checkpoints	10.18	11.37	120.58	25.93
Number of recorded checkpoints (approx.)		30	2,350	1,255,000	4
Values saved in change-log-based checkpoints (approx.)		26,000	270,000	1,880,000	9,000

- Overhead ranges from 1%-194%
- A 194% overhead to avoid runtime errors may be worth the tradeoff

How Is It Better?

- Less costly than Replicated server methods.
  - No copies
  - Retries with different variations of the same method.

- More General and extensible
  - Other work directed at Data Structures, infinite loops, configuration incompatibilities etc.
  - Generic because it finds workarounds inherent in the libraries
  - Can work for most libraries with redundancy

Contributions

1. ARMOR

2. Technique using workarounds and rollbacks

Citations

1. Carzaniga, Antonio, et al. "Automatic recovery from runtime failures." Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013.

Discussion Questions

1. Pre-processing needs to be done for each library individually: is this feasible?

Discussion Questions

2. What downsides can you foresee in this research?

Discussion Questions

3. Exactly how redundant is the typical library?

Discussion Questions

4. ARMOR uses runtime exceptions to detect errors: how does it ignore exceptions which the developer catches and handles themselves?

Discussion Questions

5. Would it be feasible to make the code replacements permanent instead of dynamically-inserted for the purpose of making the system more error resistant?