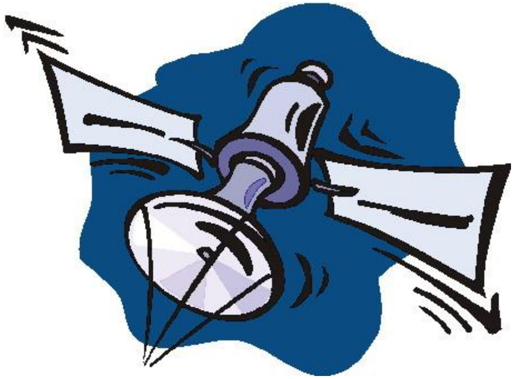
Towards a Formal Foundation of Intermittent Computing

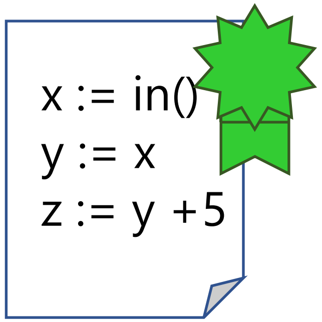
Milijana Surbatovich

Brandon Lucia, Limin Jia

Batteryless Energy-harvesting Devices (EHDs) enable computing in inaccessible environments

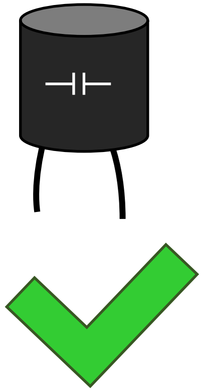


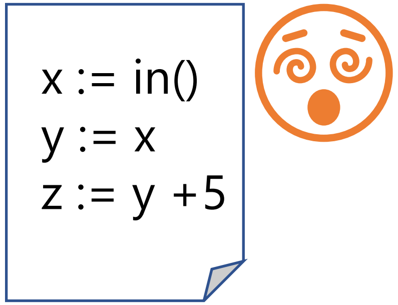
Maintenance expensive or impossible



```
x := in()
y := x
z := y +5
```

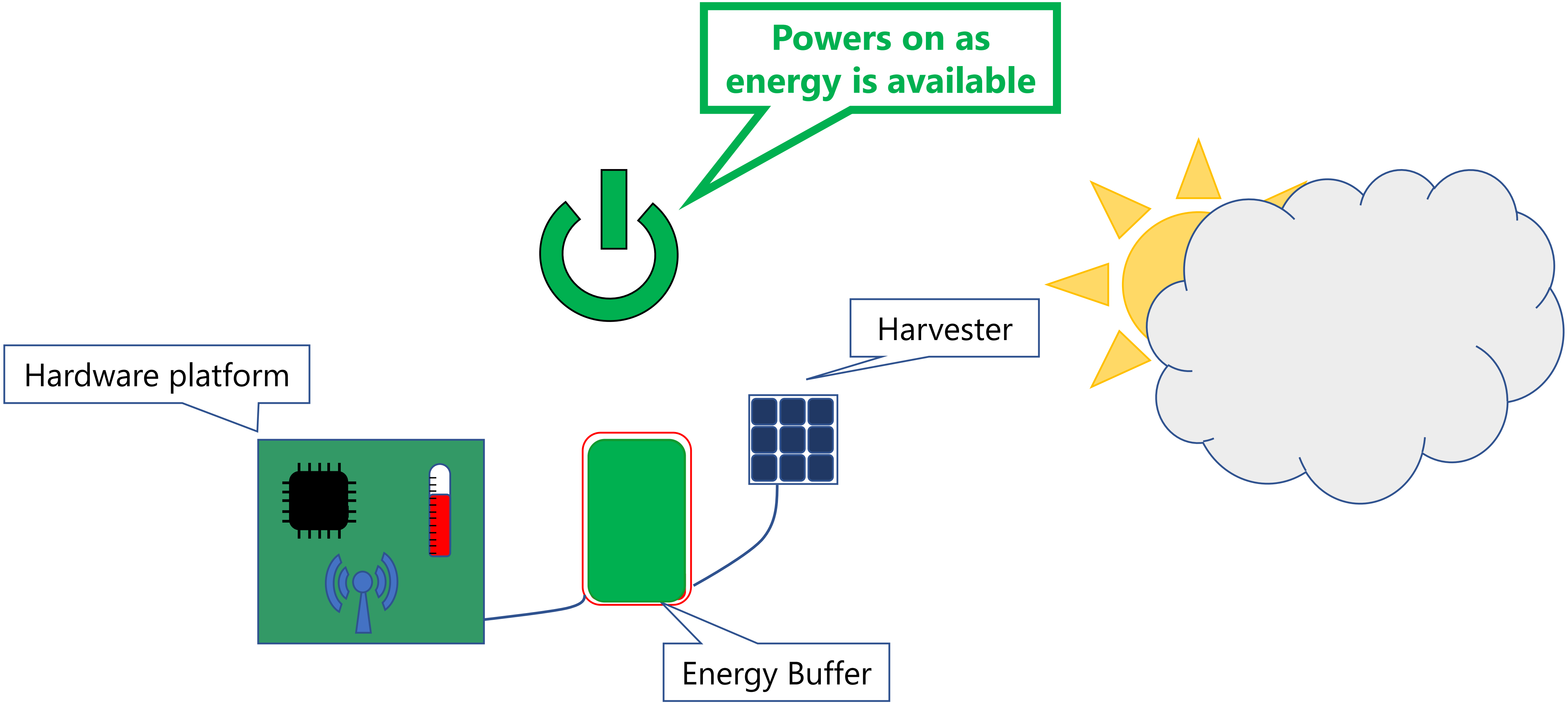
Batteryless EHDs



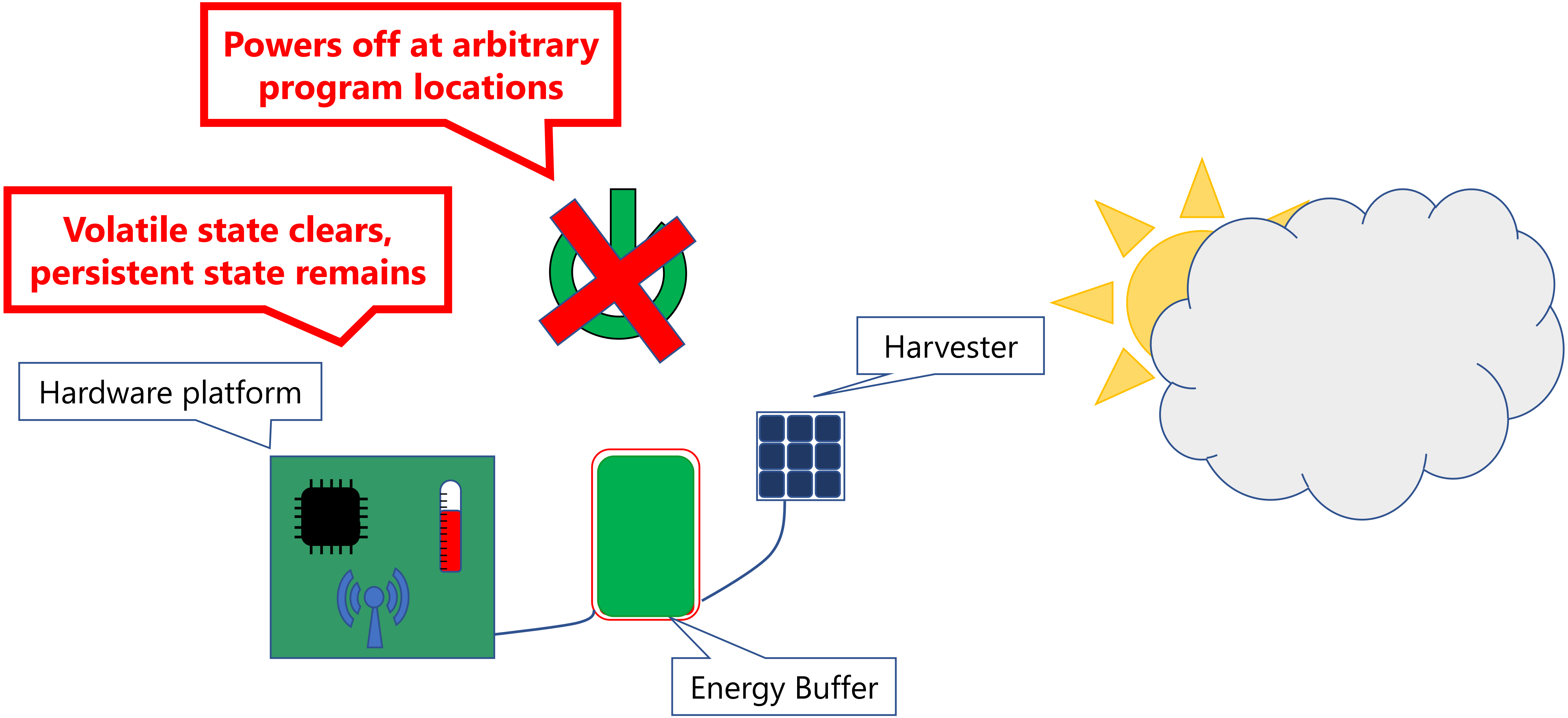


```
x := in()
y := x
z := y +5
```

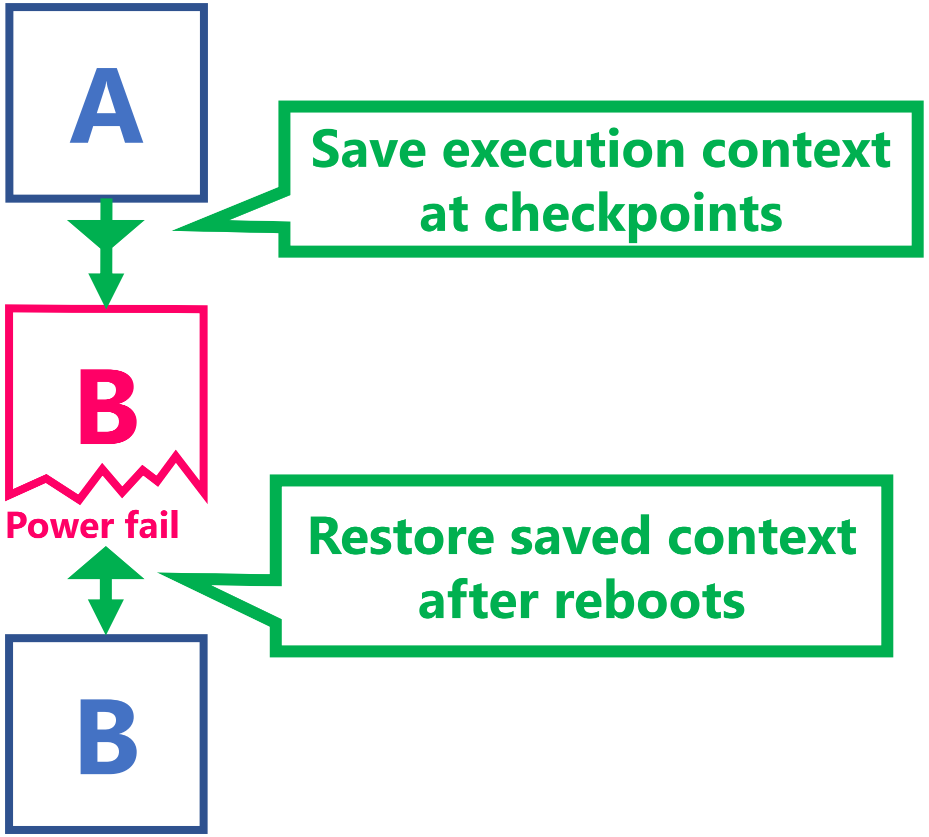
Intermittent execution in energy harvesting devices



Intermittent execution in energy harvesting devices

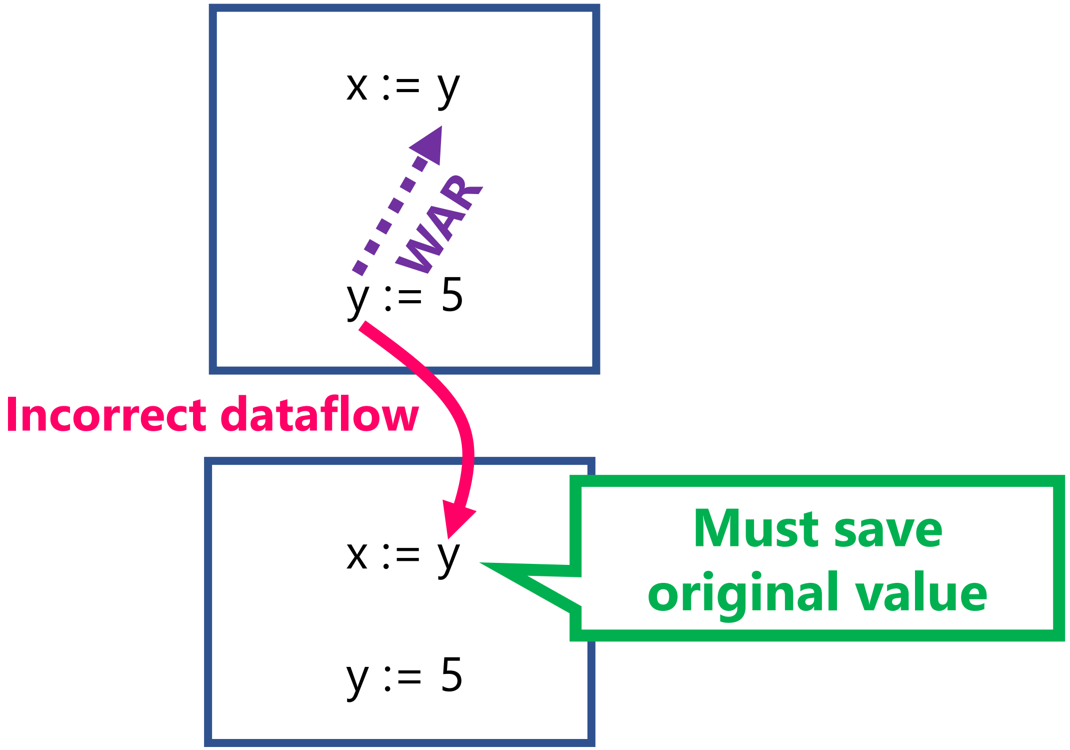

Powers off at arbitrary program locations
Volatile state clears, persistent state remains
Hardware platform
Harvester
Energy Buffer

Preserving progress by saving state


A
Save execution context at checkpoints
B
Power fail
Restore saved context after reboots
B

Systems must re-execute regions correctly

Write-After-Read (WAR)



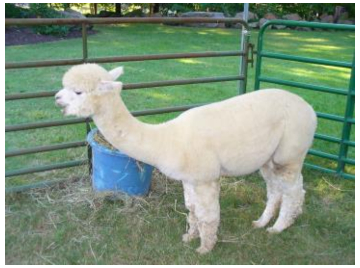
```
x := y

y := 5
```

```
x := y

y := 5
```

Alpaca



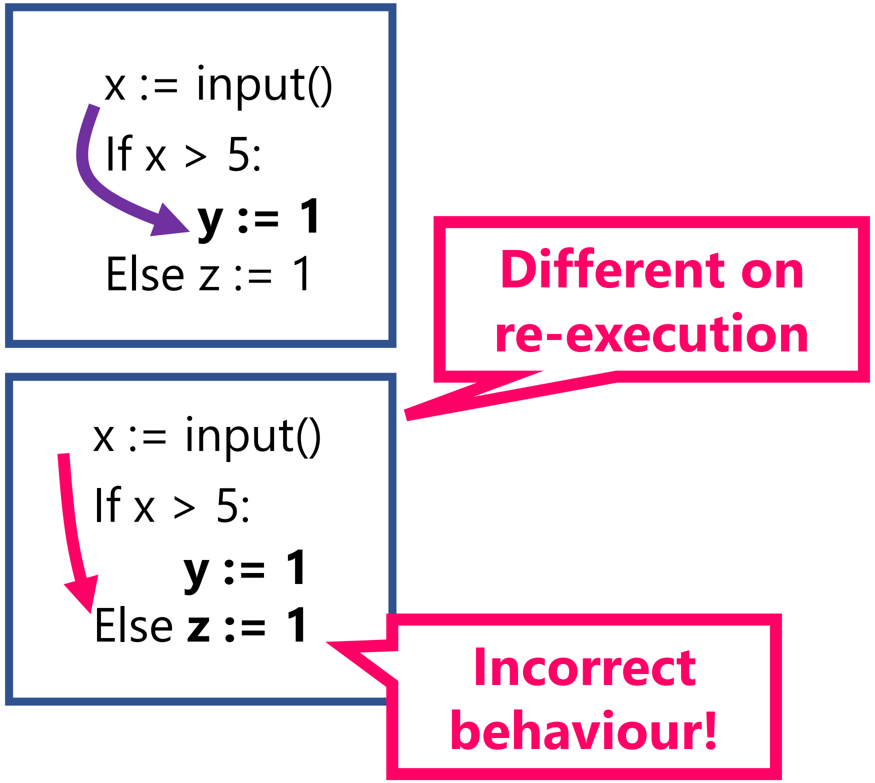
Adds value of non-volatile variables with a WAR dependence to the saved execution context

K. Maeng, A. Colin, B. Lucia. Alpaca: Intermittent Execution without Checkpoints. OOPSLA '17

Others: DINO, Ratchet,
Chinchilla

Input re-executions are not handled correctly

Repeated-Input-Operation (RIO)



```
x := input()
If x > 5:
    y := 1
Else z := 1
```

Different on re-execution

```
x := input()
If x > 5:
    y := 1
Else z := 1
```

Incorrect behaviour!

IBIS

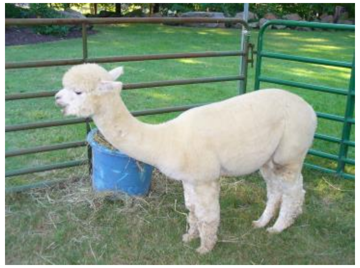
Detects and reports input-dependent branches that write to different sets of variables

M. Surbatovich, L. Jia, B. Lucia. I/O Dependent Idempotence Bugs in Intermittent Systems. OOPSLA '19

The need to formalize intermittent execution

No formal spec in existing works → systems subtly incorrect

Our correctness definitions address both WAR and RIO problems, which no existing work has done

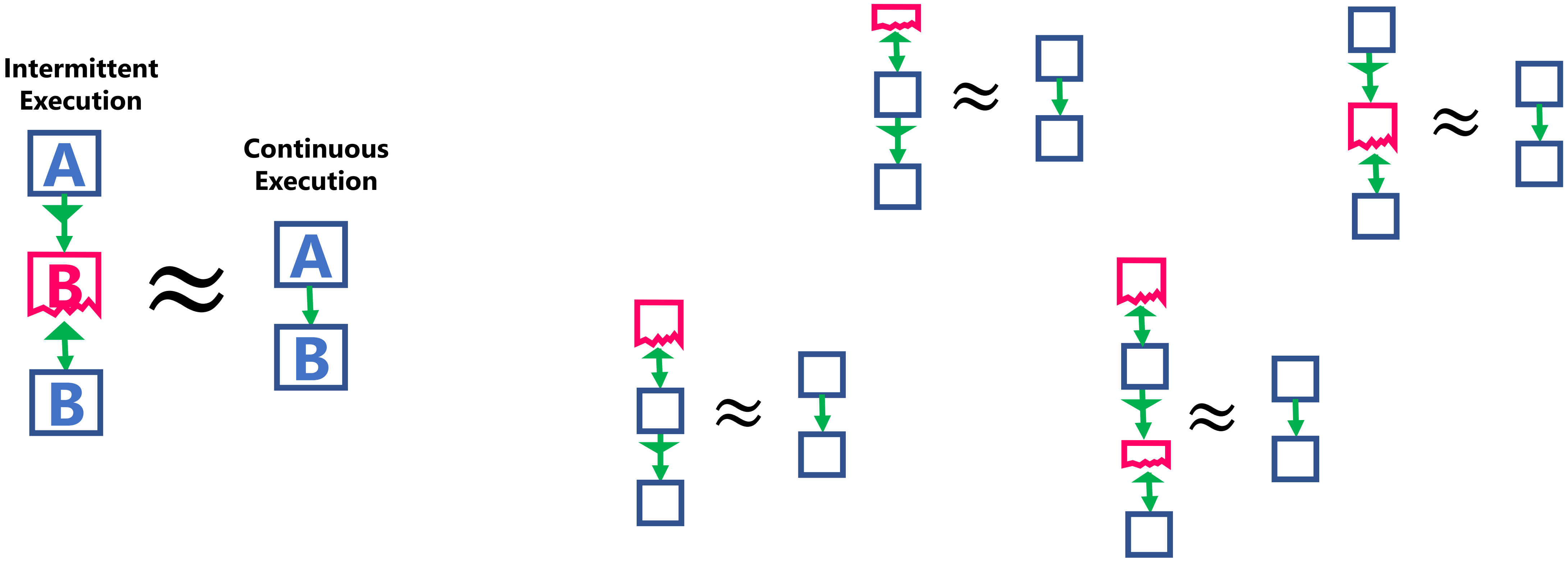


Outline

- Challenge of intermittence
- **Memory consistency correctness definition**
- Memory relations
- Correct checkpoint set
- Evaluation and conclusion

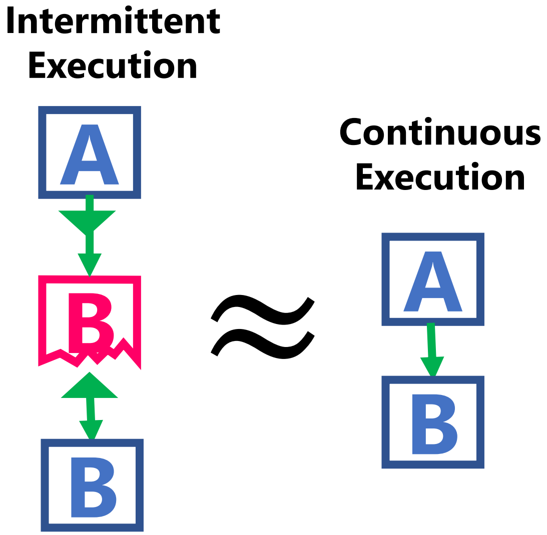
Correct intermittent execution

Continuous execution specifies correct program behaviour



Difficulty of reasoning about equivalence

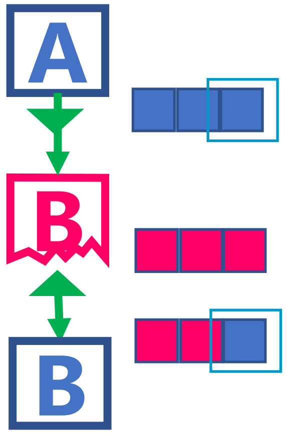
Equivalence: memory reads and memory state at checkpoints



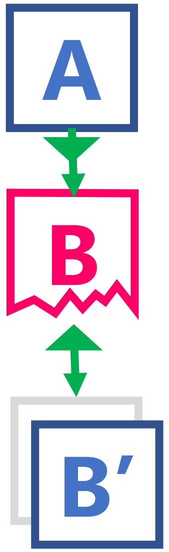
Intermittent Execution

Continuous Execution

Reboots don't restore to the exact same state

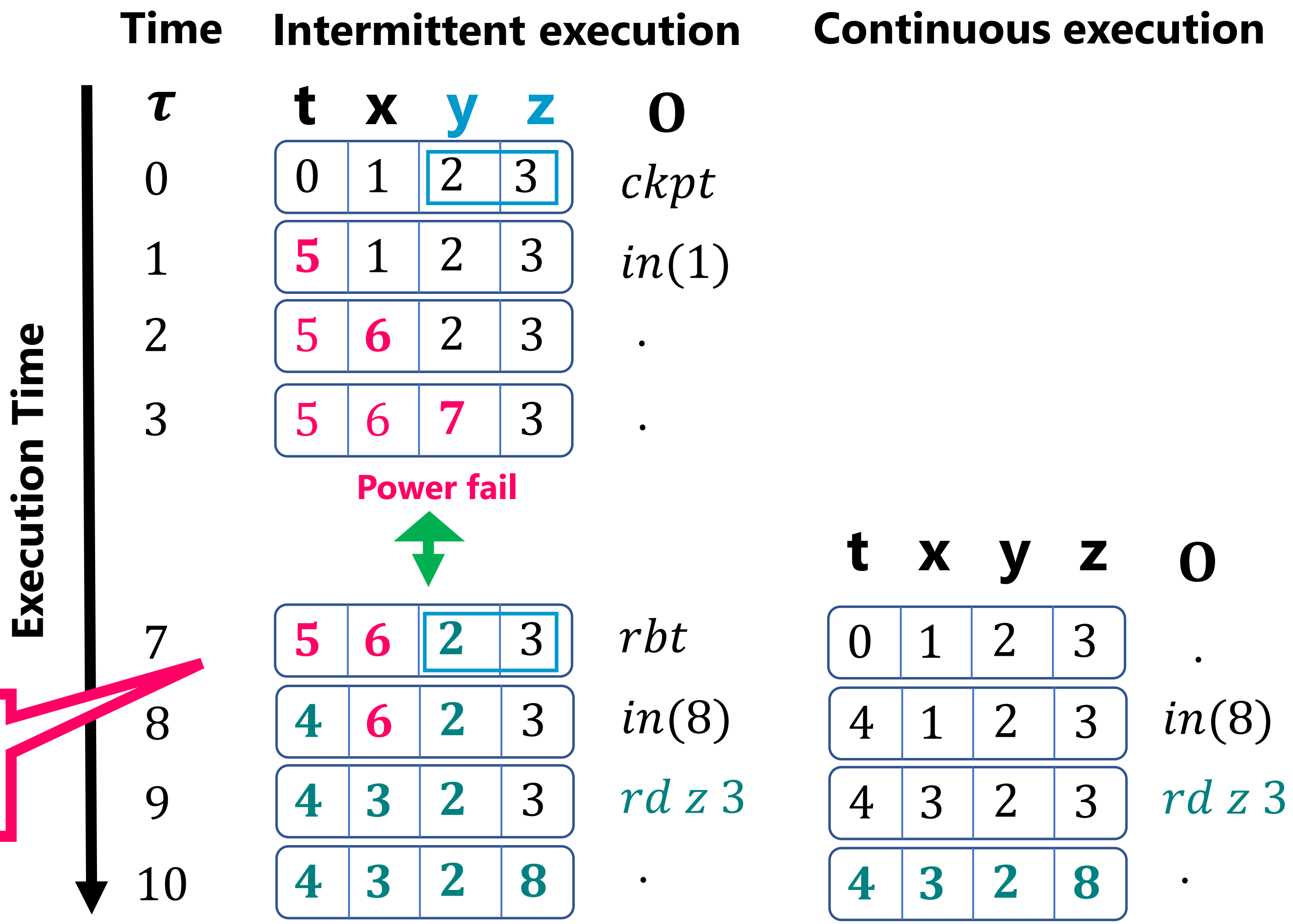


Inputs cause different paths to be taken



Memory can be different at many points

```
checkpoint(y,z)
t = temp();
if t >= 5
then x := 6;
     y := 7;
else x := z;
     z := 8;
```



Time τ	Intermittent execution t	x	y	z	O	Continuous execution t	x	y	z	O
0	0	1	2	3	*ckpt*					
1	5	1	2	3	*in*(1)					
2	5	6	2	3	.					
3	5	6	7	3	.					
Power fail										
7	5	6	2	3	*rbt*	0	1	2	3	.
8	4	6	2	3	*in*(8)	4	1	2	3	*in*(8)
9	4	3	2	3	*rd z* 3	4	3	2	3	*rd z* 3
10	4	3	2	8	.	4	3	2	8	.

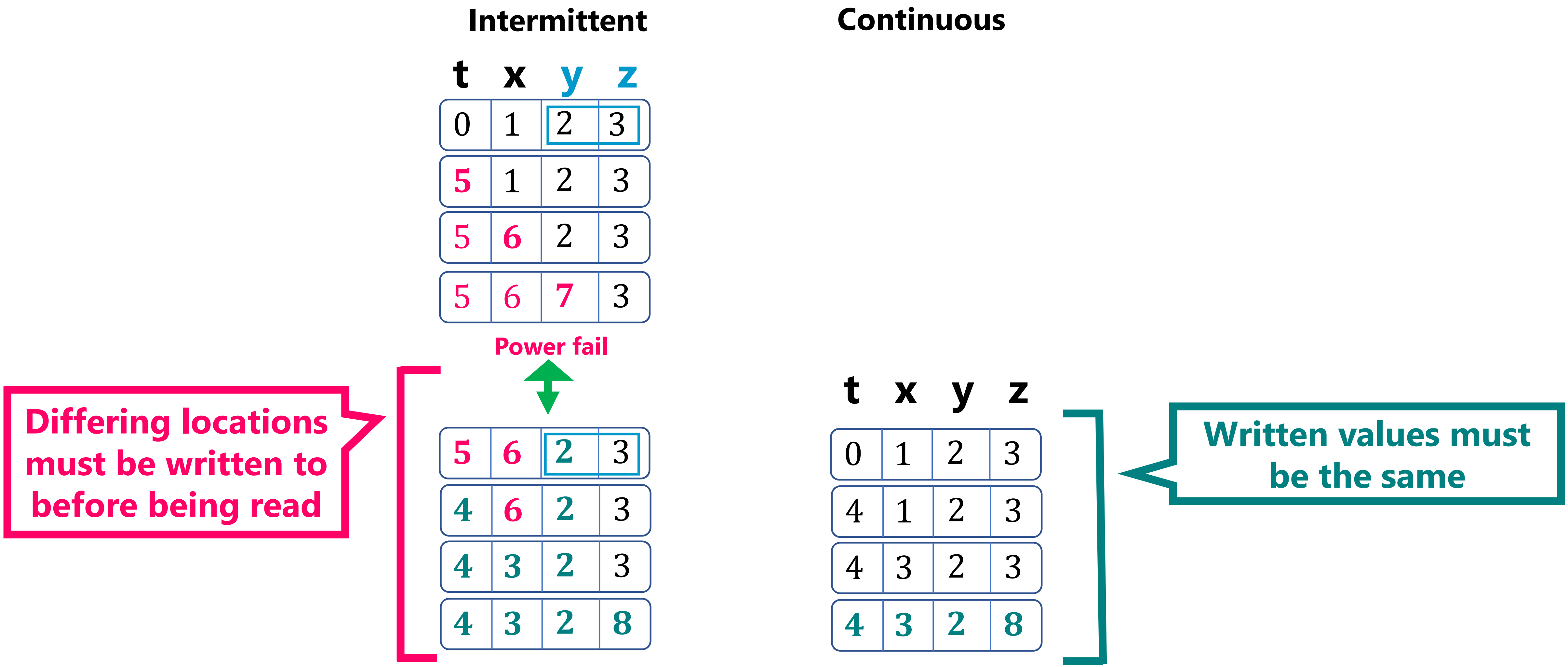
Not same state as at checkpoint

How different can memory get that the differences still resolve?

Outline

- Challenge of intermittence
- Memory consistency correctness definition
- **Memory relations**
- Correct checkpoint set
- Evaluation and conclusion

Any differences must resolve on re-execution



Intermittent

t	x	y	z
0	1	2	3
5	1	2	3
5	6	2	3
5	6	7	3

Power fail

5	6	2	3
4	6	2	3
4	3	2	3
4	3	2	8

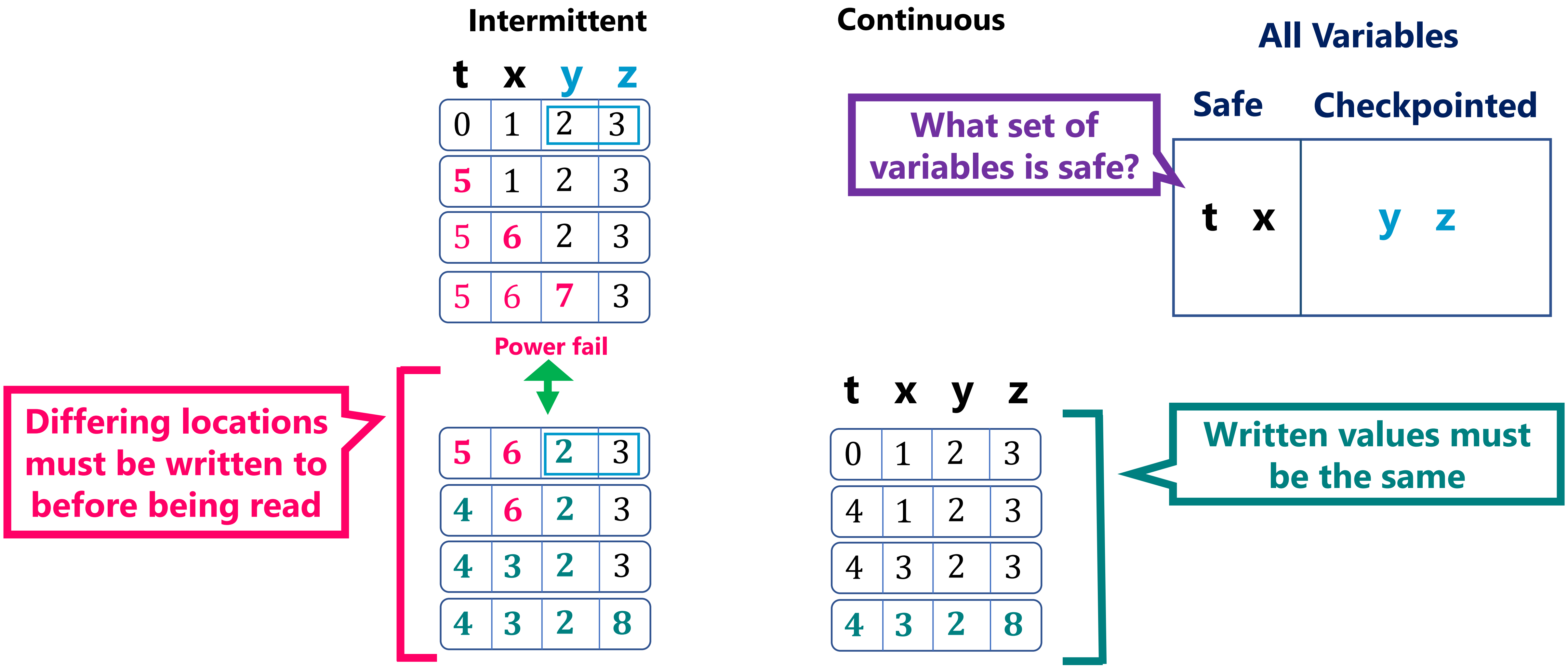
Differing locations must be written to before being read

Continuous

t	x	y	z
0	1	2	3
4	1	2	3
4	3	2	3
4	3	2	8

Written values must be the same

Any differences must resolve on re-execution



Intermittent

t	x	y	z
0	1	2	3
5	1	2	3
5	6	2	3
5	6	7	3

Power fail

5	6	2	3
4	6	2	3
4	3	2	3
4	3	2	8

Differing locations must be written to before being read

Continuous

What set of variables is safe?

All Variables

Safe	Checkpointed
t x	y z

t	x	y	z
0	1	2	3
4	1	2	3
4	3	2	3
4	3	2	8

Written values must be the same

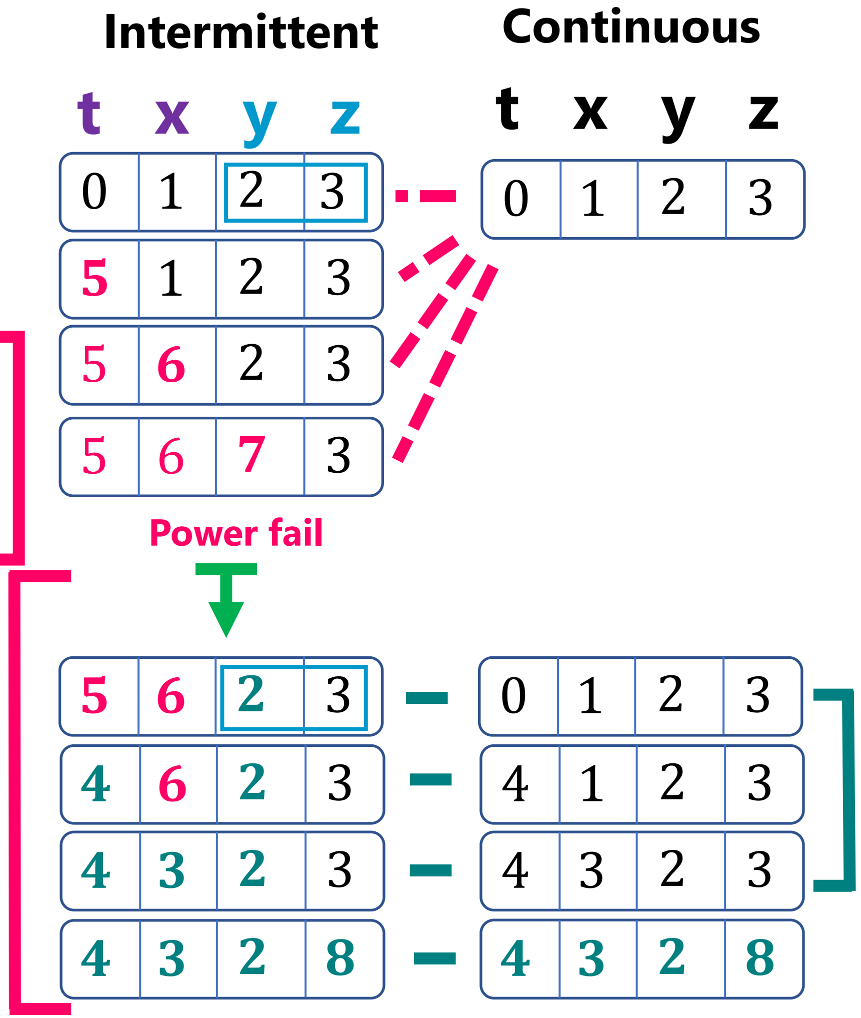
Must-first-write set

The ***must-first-write*** set – must-write variables with no preceding read

Any execution writes to these variables before reading them

```
checkpoint(y,z)
t = temp();
if t >= 5
then x := 6;
     y := 7;
else  x := z;
     z := 8;
```

Defining allowable differences



Intermittent

t	x	y	z
0	1	2	3
5	1	2	3
5	6	2	3
5	6	7	3

Continuous

t	x	y	z
0	1	2	3

Power fail

t	x	y	z
5	6	2	3
4	6	2	3
4	3	2	3
4	3	2	8

t	x	y	z
0	1	2	3
4	1	2	3
4	3	2	3
4	3	2	8

Relation 1: Differing locations must be in the MstFstWt set or the checkpoint set

Always writes to MstFtWt + checkpoint

Relation 2: Differing locations must be in the MstFstWt set and cannot have been written to yet.

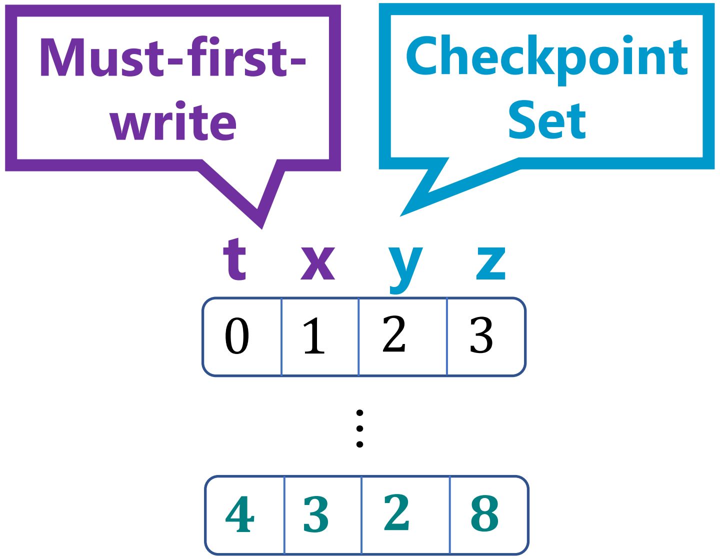
Each write resolves differences

Final state matches

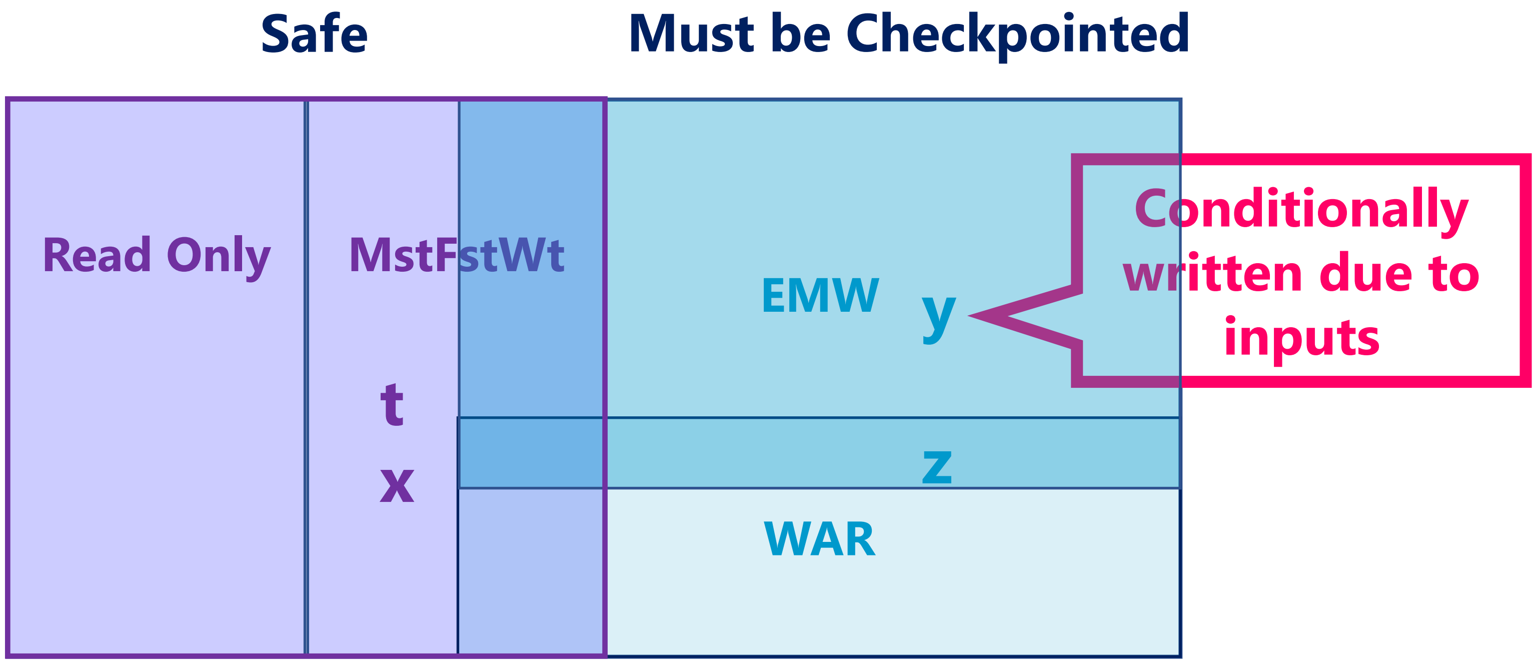
Outline

- Challenge of intermittence
- Memory consistency correctness definition
- Memory relations
- **Correct checkpoint set**
- Evaluation and conclusion

Only checkpointing WAR variables is incorrect



```
checkpoint(y,z)
t = temp();
if t >= 5
then x := 6;
     y := 7;
else x := z;
     z := 8;
```



Exclusive May-Write set:

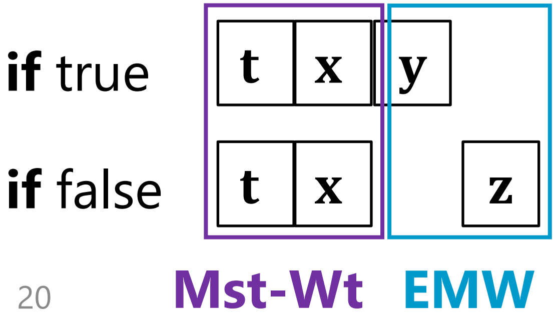
may-writes minus must-write

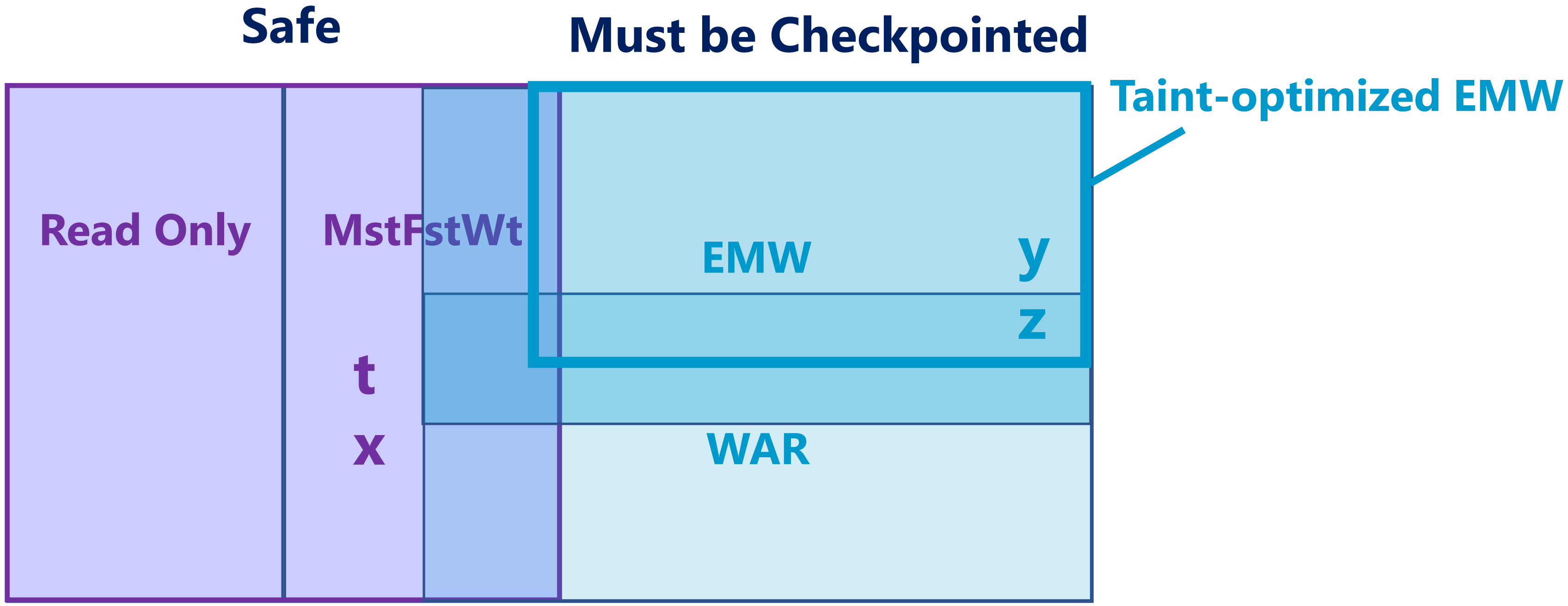
Collecting Exclusive May-Writes

Only inputs can cause a different path to execute after reboot

Use static taint analysis to identify input-dependent branches

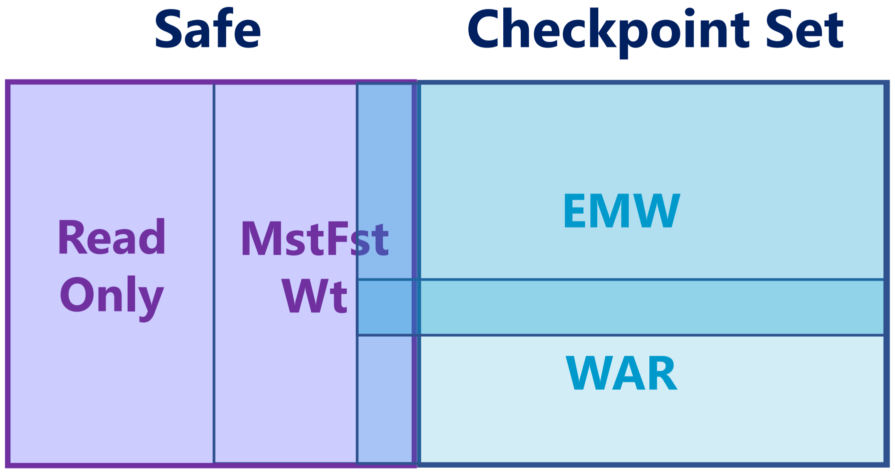
```
checkpoint(y,z)
t = temp();
if t >= 5
then x := 6;
         y := 7;
else  x := z;
        z := 8;
```

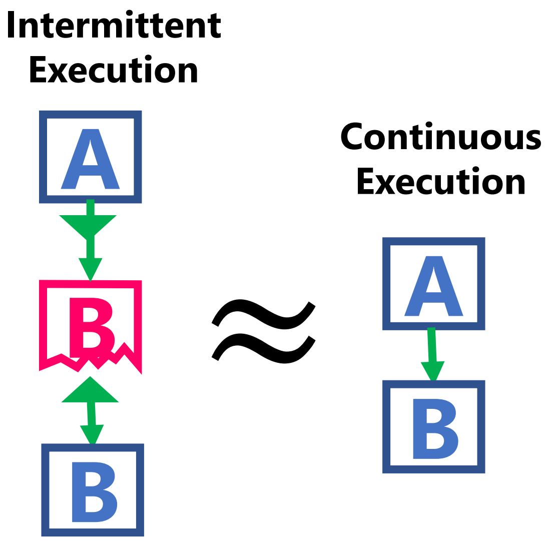




Correctness Theorem

If all unsafe WAR and EMW variables are in the checkpointed set, then an intermittent program will execute correctly





Implementation

Compiler pass implemented in LLVM

Two versions: taint-optimized EMW and basic EMW

Analysis added to Alpaca, which tracks WAR

More in paper…

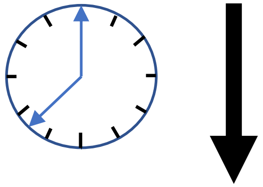
Outline

- Challenge of intermittence
- Memory consistency correctness definition
- Memory relations
- Collecting the correct checkpoint set
- **Evaluation and conclusion**

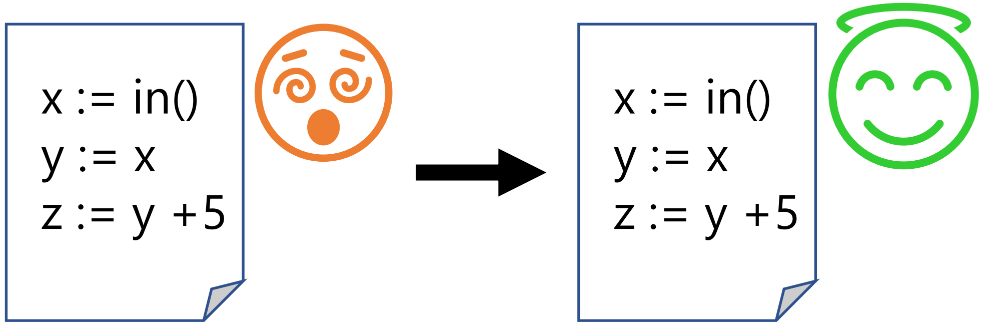
Goal of evaluation

Show that Modifying Alpaca with EMW is practically efficient

1) Low runtime overhead



2) Low programmer burden

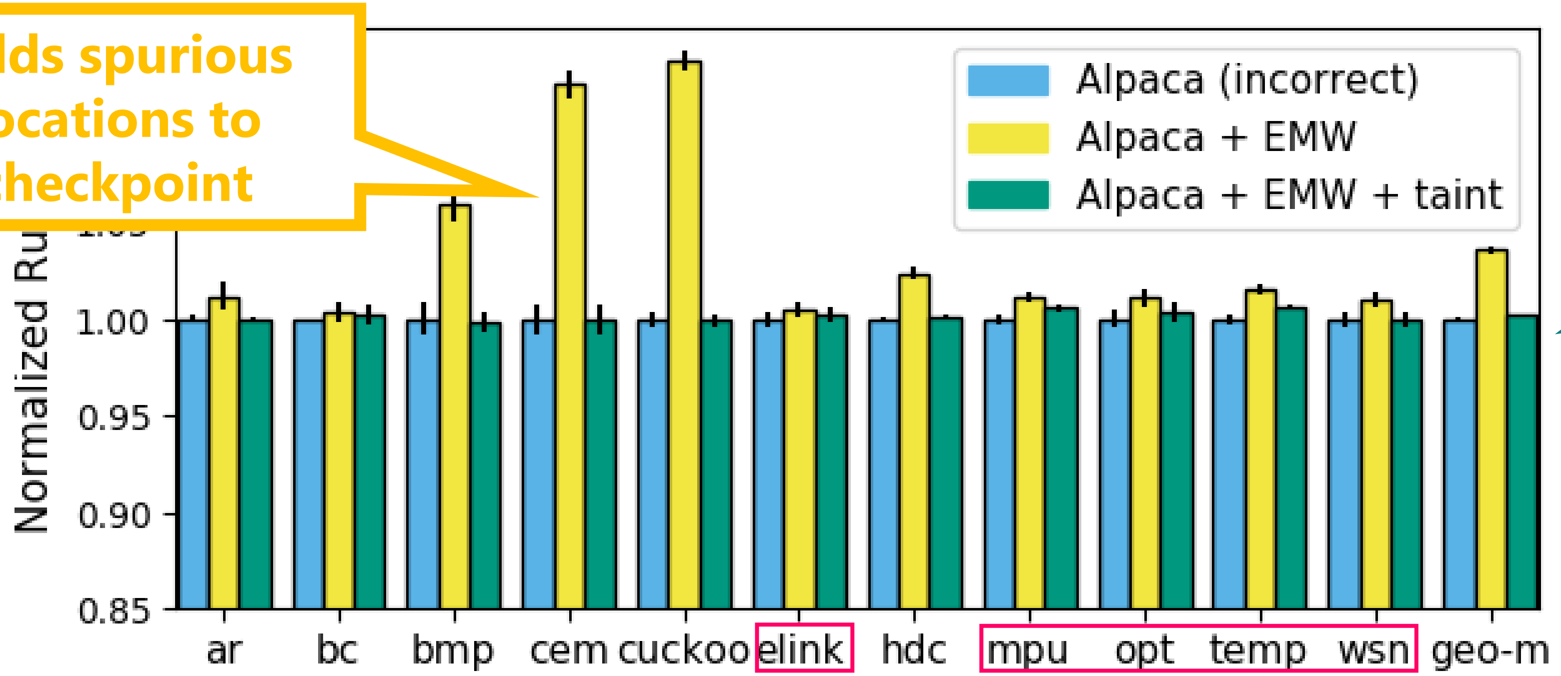


```
x := in()
y := x
z := y +5
```

```
x := in()
y := x
z := y +5
```

EMW has little performance penalty

Experiments run on benchmarks from prior work on real hardware



Adds spurious locations to checkpoint

Low mean overhead

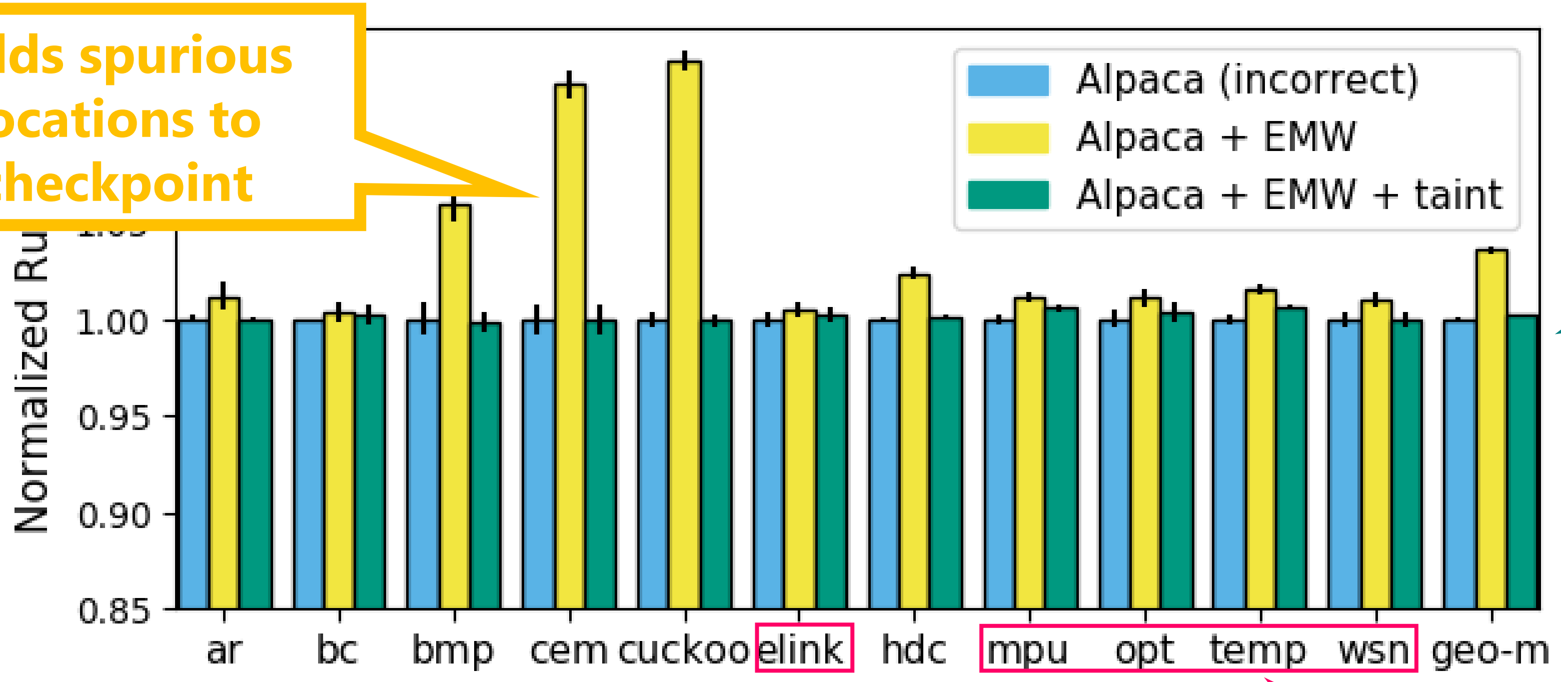
Had input-dep branches

EMW needs little to no programmer effort

Manual Fix, could still be incorrect

No effort, higher overhead

Specify input functions, little overhead



Adds spurious locations to checkpoint

Low mean overhead

Had input-dep branches

M. Surbatovich, L. Jia, B. Lucia. I/O Dependent Idempotence Bugs in Intermittent Systems. OOPSLA ‘19

More in paper

Proving equivalence between execution models

Collection and checking algorithms

Implementation and experiment details

Application Discussion

Connection to related work

Persistent Memory Models

Persist vs execution order

Multi-threaded executions

ISA persistency semantics

[Raad et al., Israelevitz et al., Pelley et al.]

Crash Consistency

Equivalence of crashy execution to non-crashy

Automated proof tools:

- Yggdrasil, CHL
- Fault Tolerant Resource Reasoning
- Crash Consistency through Reachability

[Bornholt et al., Chen et al., Ntzik et al., Koskinen and Yang]

This work

Explicitly considers **non-deterministic inputs**

Defines **correctness conditions for intermittent executions**

Summary

Intermittent computing systems need to be correct and reliable

We develop a framework and give a formal definition of correctness

We apply the framework to reason about equivalence and develop a compiler analysis to make existing systems correct

Towards a Formal Foundation of Intermittent Computing

Milijana Surbatovich

Brandon Lucia, Limin Jia

Carnegie Mellon University

Electrical & Computer ENGINEERING