18-613 Future of Computing

Milijana Surbatovich, Kiwan Maeng, Harsh Desai

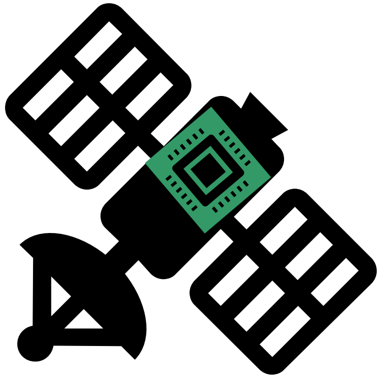
Carnegie
Mellon
University

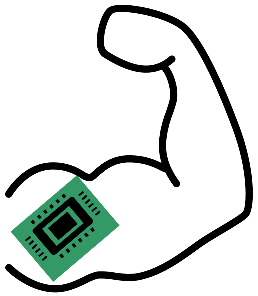
Electrical & Computer
ENGINEERING

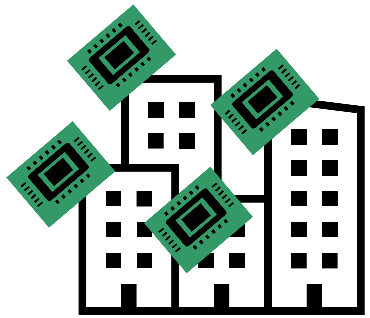
Outline

- **Basics of intermittent computing**
- PL for intermittent computing
- Systems for intermittent computing
- Architectures for intermittent computing

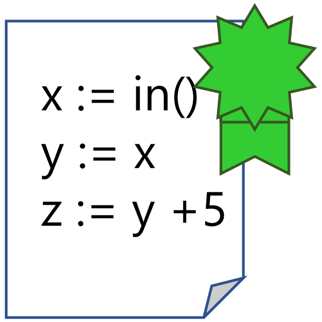
Batteryless Energy-harvesting Devices (EHDs) enable computing in inaccessible environments





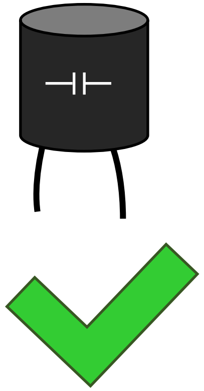


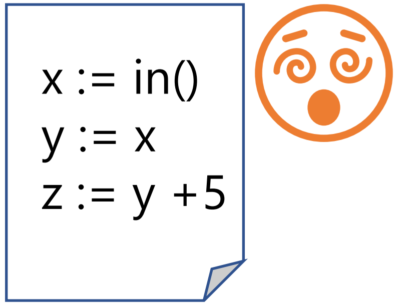
Maintenance expensive or impossible



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

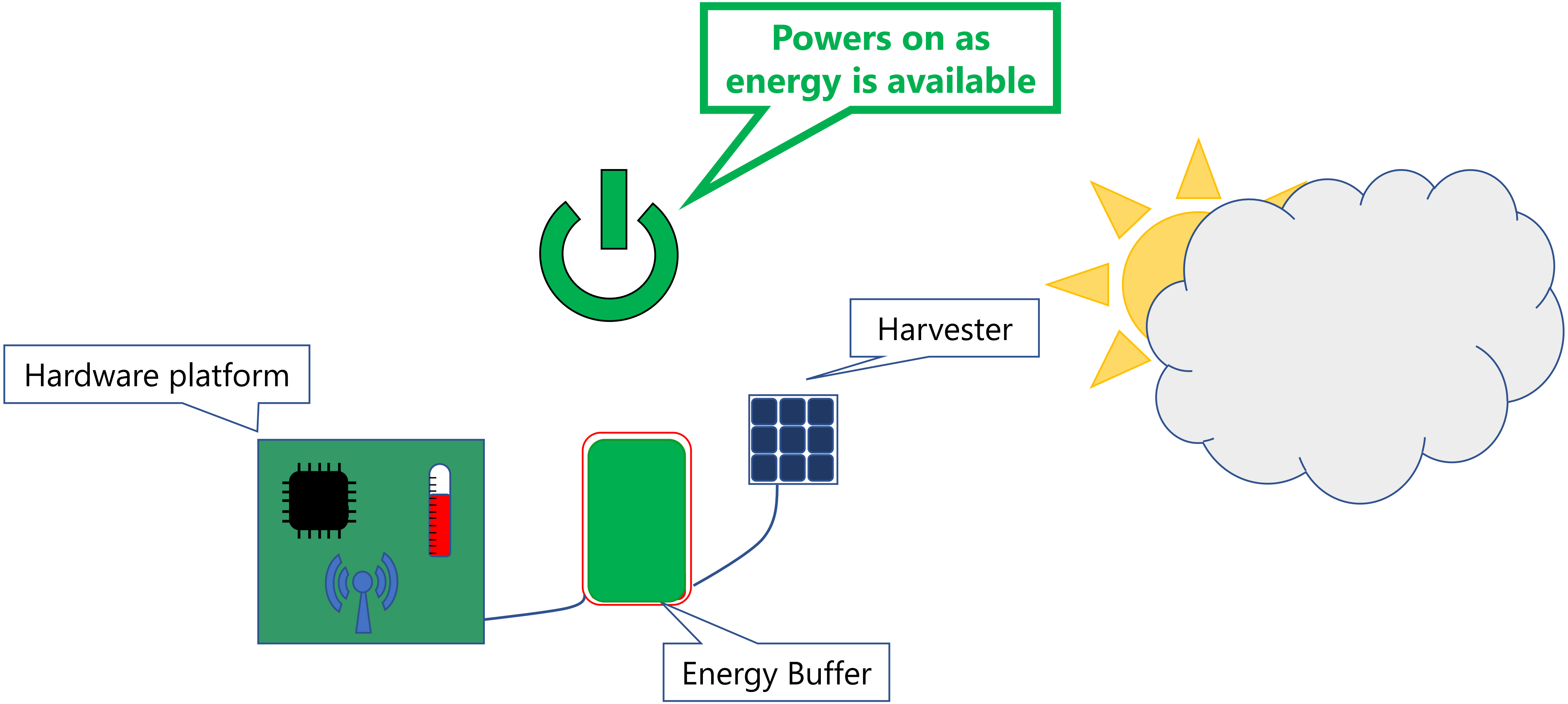
Batteryless EHDs



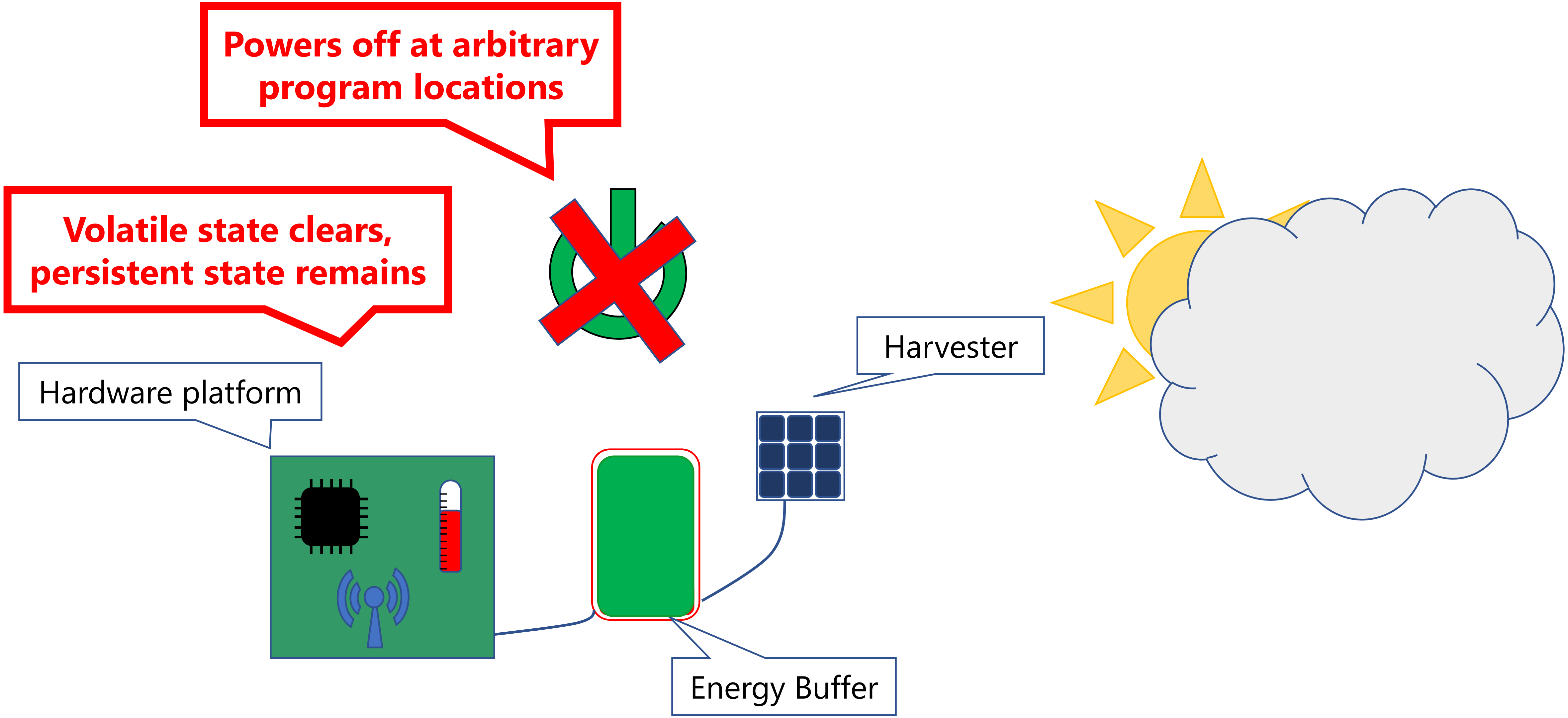


```
x := in()
y := x
z := y +5
```

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

Mixed-volatility Memory

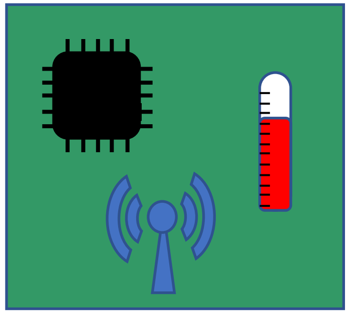
- Volatile Memory loses state when the power turns off
  - Register file, DRAM (traditional)
- Non-volatile Memory keeps state when the power turns off
  - Disk(traditional), Flash, STT-MRAM

Volatile

Registers

Peripherals

(Maybe) Stack



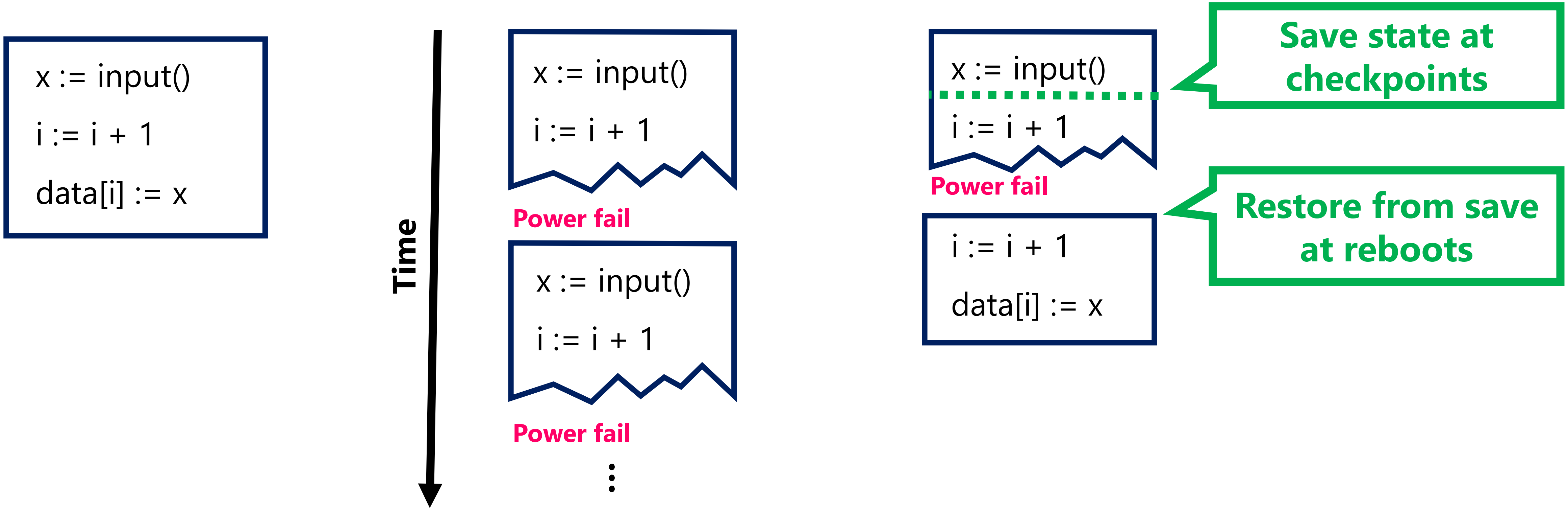
Non-volatile

Code

Program data

Programs checkpoint to make progress

If registers are cleared, program will restart from the beginning



```
x := input()
i := i + 1
data[i] := x
```

Time

```
x := input()
i := i + 1
```

Power fail

```
x := input()
i := i + 1
```

Power fail

⋮

```
x := input()
i := i + 1
```

Power fail

```
i := i + 1
data[i] := x
```

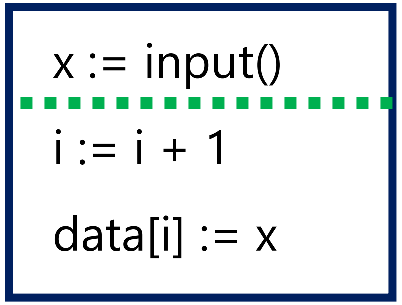
Save state at checkpoints

Restore from save at reboots

Checkpointing Methods

In-code checkpoints

- Programmer or compiler adds
- Re-execute from last checkpoint

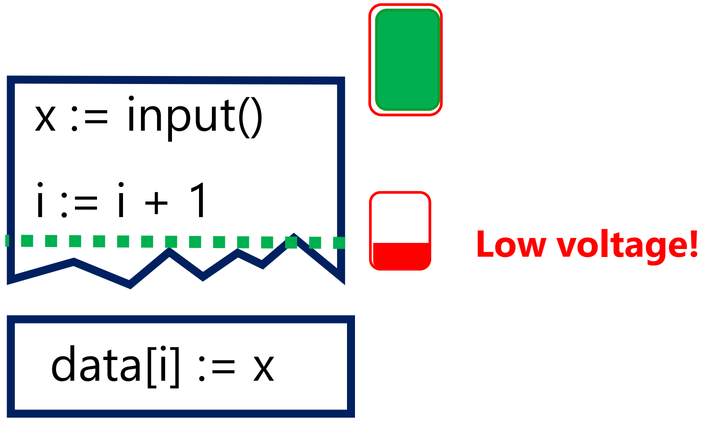


```
x := input()
i := i + 1
data[i] := x
```

(Focus of this section)

Just-in-time (JIT) checkpointing

- Hardware to monitor voltage
- Checkpoint on low power
- Generally no re-execution



```
x := input()
i := i + 1
data[i] := x
```

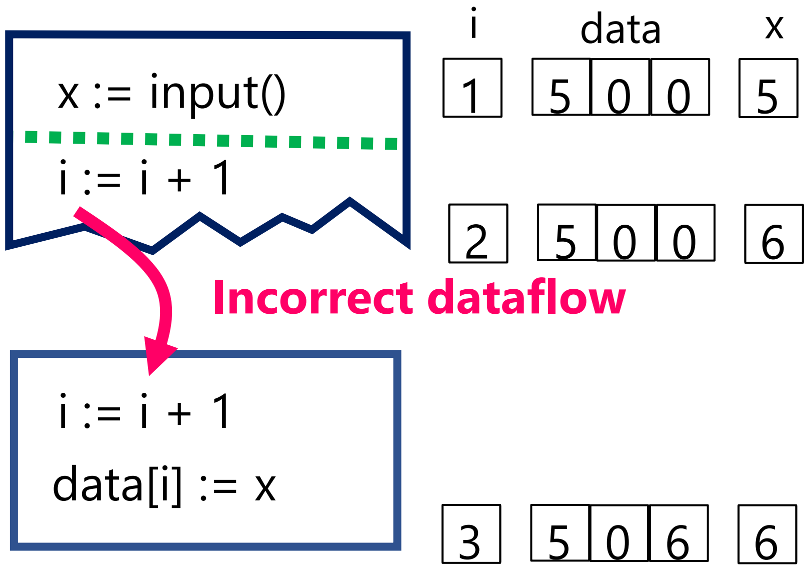
(More on this in next section)

Outline

- Basics of intermittent computing
- PL for intermittent computing
  - Memory bugs caused by intermittence
  - Formally Defining Correctness
  - Correct checkpoint set
- Systems for intermittent computing
- Architecture for intermittent computing

Systems must re-execute regions correctly

Write-After-Read (WAR)



```
x := input()
i := i + 1
```

```
i := i + 1
data[i] := x
```

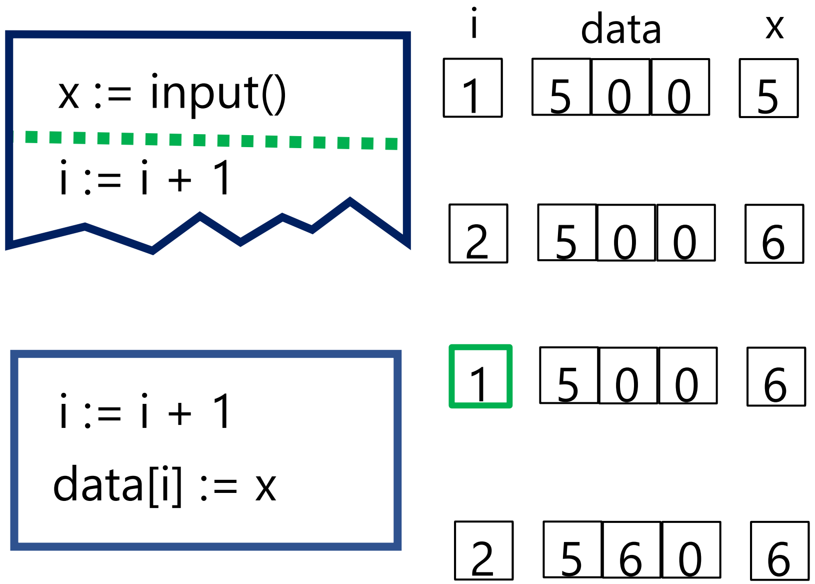
i	data			x
1	5	0	0	5
2	5	0	0	6
3	5	0	6	6

Incorrect dataflow

Skipped slot!

Must save starting value of i

Correct Execution



```
x := input()
i := i + 1
```

```
i := i + 1
data[i] := x
```

i	data			x
1	5	0	0	5
2	5	0	0	6
1	5	0	0	6
2	5	6	0	6

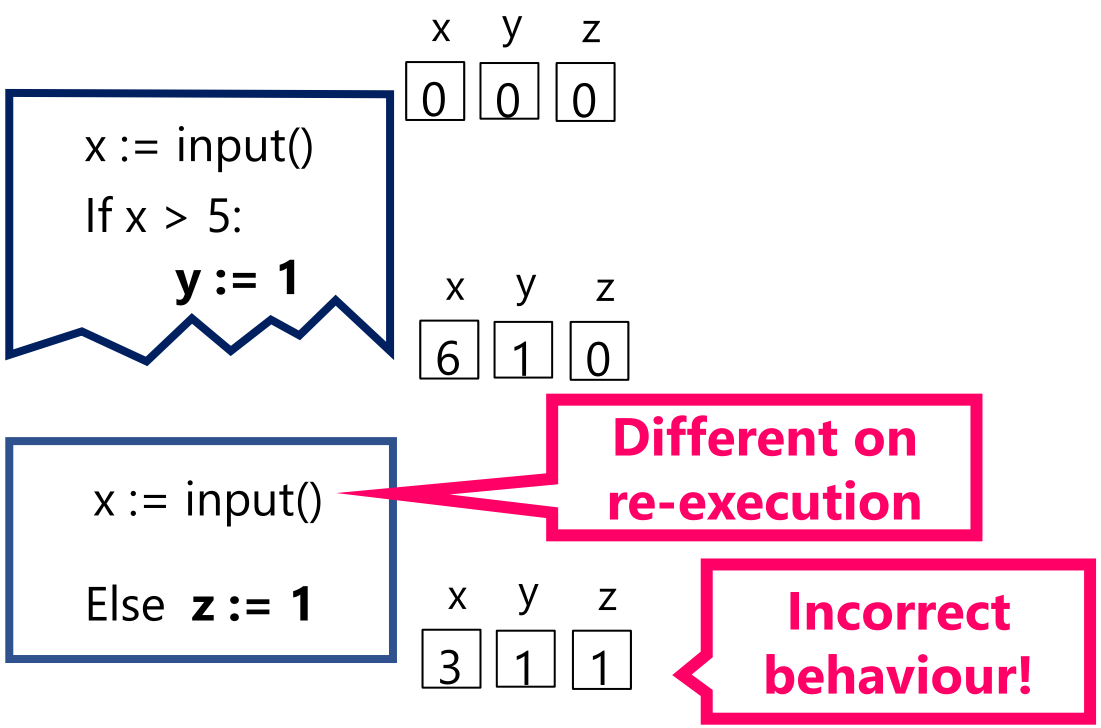
State-of-the-art is to add WAR variables to the checkpoint set

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

Input re-executions are not handled correctly

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

Repeated-Input-Operation (RIO)



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

```
x := input()

Else  z := 1
```

x	y	z
0	0	0
6	1	0
3	1	1

Different on re-execution

Incorrect behaviour!

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 condition addresses both WAR and RIO problems, which no existing work has done

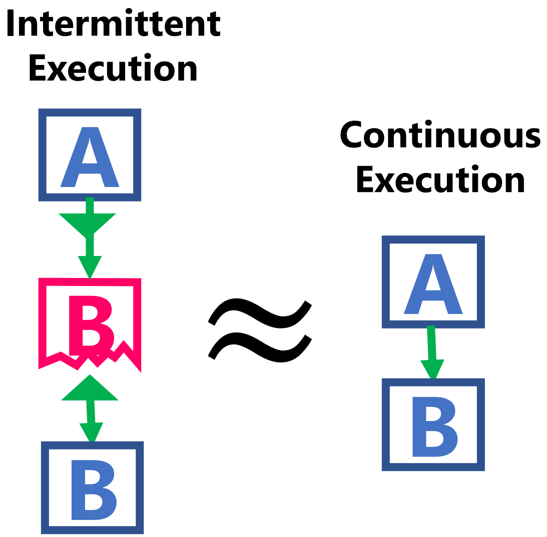
Outline

- Basics of intermittent computing
- PL for intermittent computing
  - Memory bugs caused by intermittence
  - **Formally Defining Correctness**
  - Correct checkpoint set
- Systems for intermittent computing
- Architecture for intermittent computing

What does it mean to be correct?

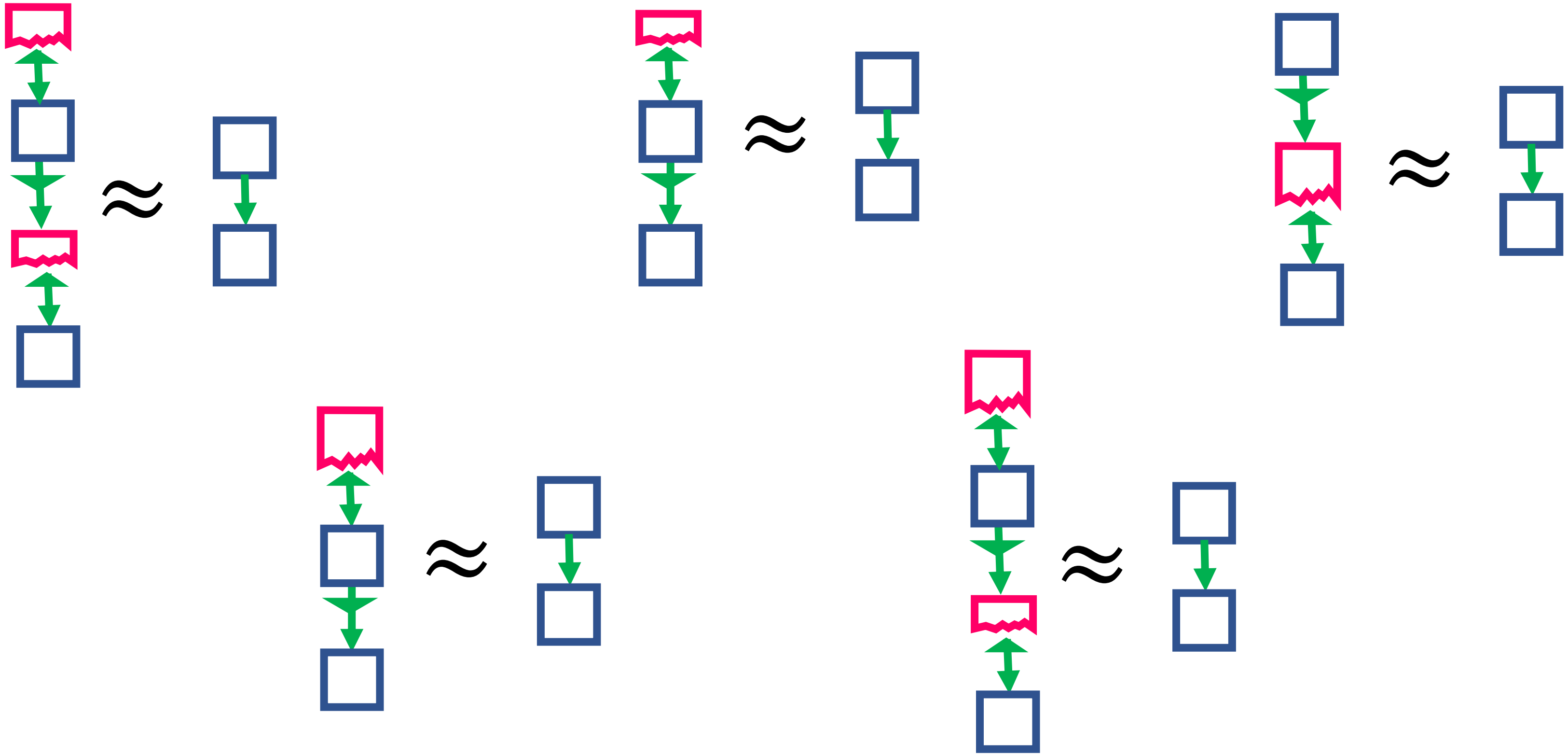
Continuous execution specifies correct program behaviour

- If intermittent execution is equivalent to a continuous execution, then it is correct



Equivalence must hold for ALL intermittent executions of a program

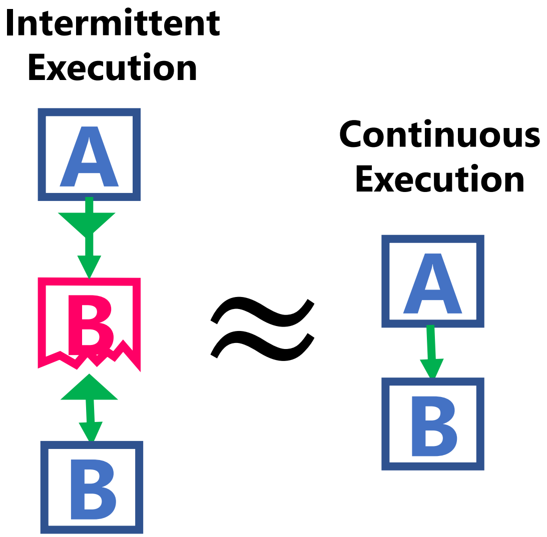
If equivalence only holds for some executions, then a program is only sometimes correct, which is no good



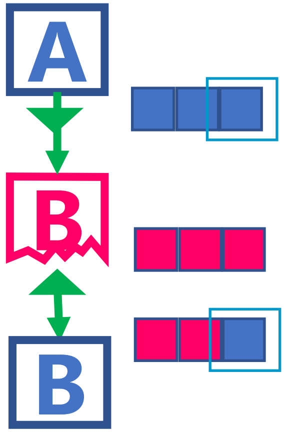
What makes equivalence difficult?

Many dimensions (time, energy…); this project looks at **memory**

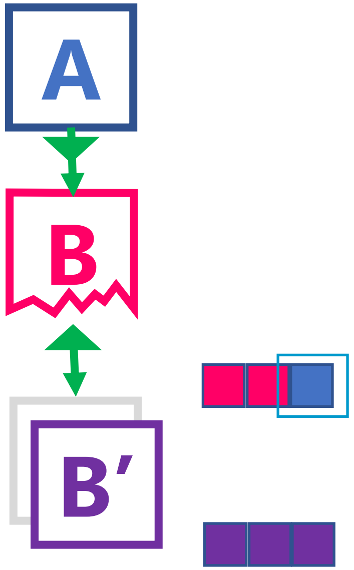
Equivalence: memory reads and memory state at checkpoints



Reboots don’t restore to the exact same state

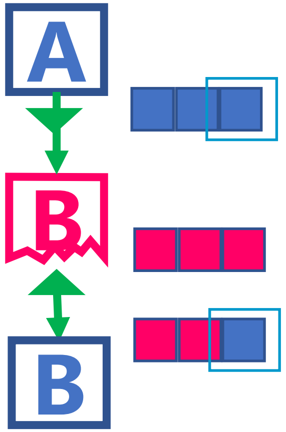


Inputs cause different paths to be taken



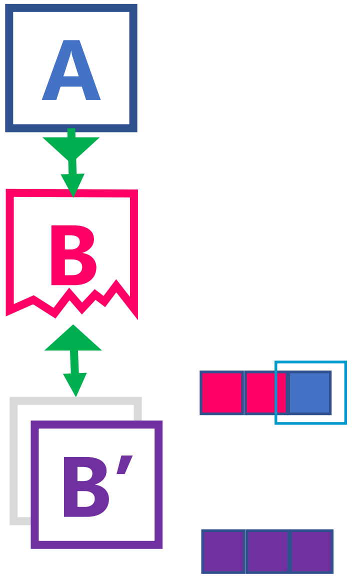
Defining acceptable differences

Reboots don't restore to the exact same state



..so differing locations should be written on re-execution (before being read)

Inputs cause different paths to be taken

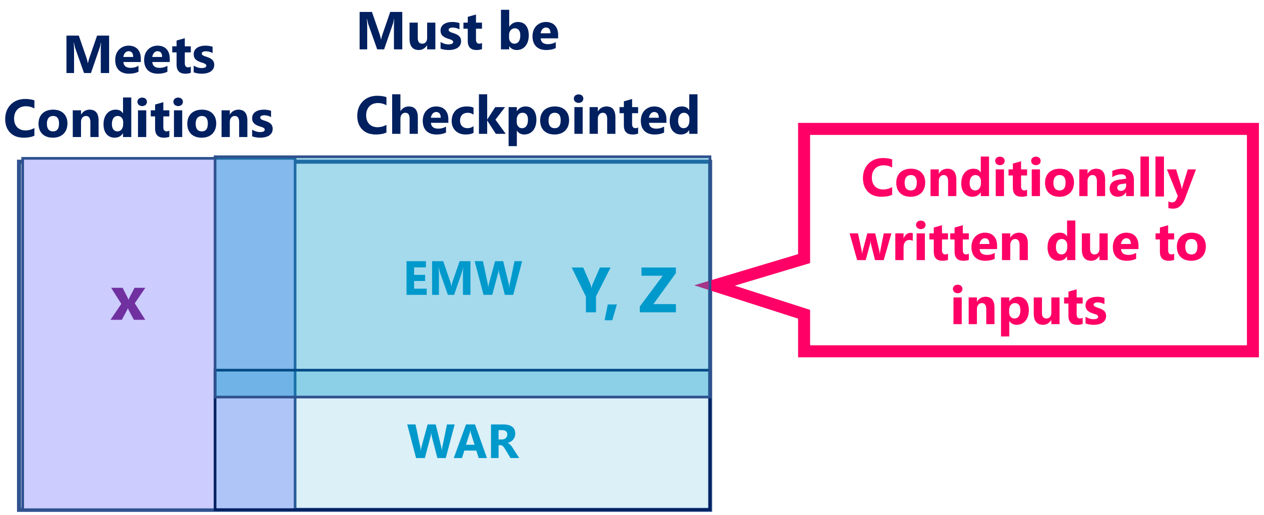


..so differing locations should be written on all paths dependent on inputs

Locations that don't fit these conditions must be checkpointed!

Many systems don't satisfy this constraint

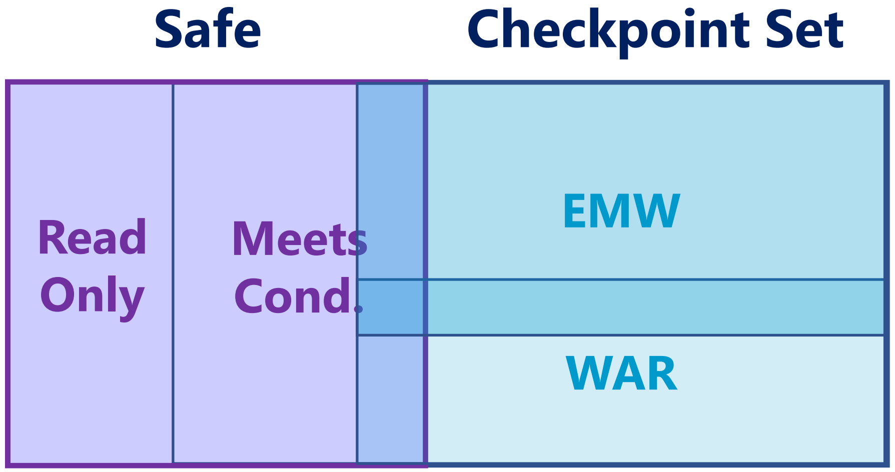
```
x := input()
If x > 5:
      y := 1
Else z := 1
```

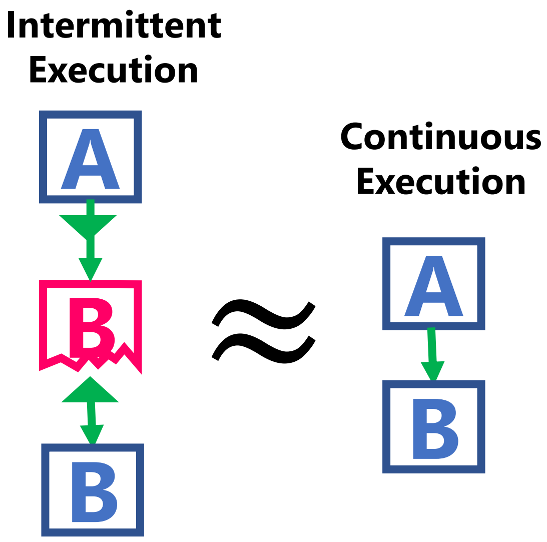


Exclusive May-Write (EMW) set: may-writes minus must-writes

Correctness Theorem

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





How to reason precisely about intermittent execution?

Define a model language and system state (simple, but should include key features)

Define how executing commands changes the state

Show that no matter what command executes, the state
of the intermittent execution is related to a continuous execution

Define a model language and system state

- Programs are made of:
  - Commands c ::= ι | $\iota; c$ | if e then c_1 else c_2
  - Instructions ι ::= … | x := e | **checkpoint(ω)** | **reboot**
  - Expressions e ::= x |v (e.g., int, bool) | $e_1 \oplus e_2$

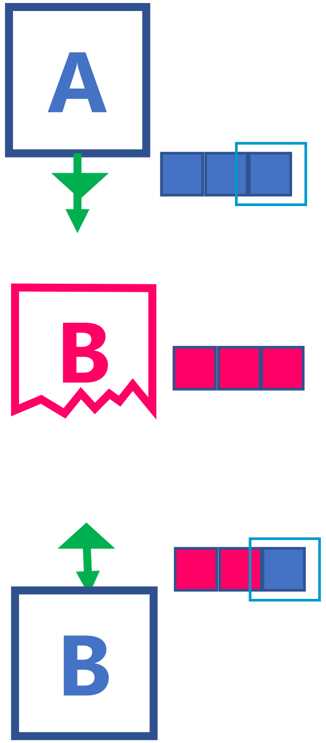
Particular to intermittence

Intermittent execution state: (κ, N, V, c)

- κ is a record of the last checkpoint
- N is non-volatile memory, a map of variables to values (x → 3)
- V is volatile memory
- C is the command to execute

Define how commands change state

Executing a command transitions a system from one state to another



Defines what happens at a checkpoint

$$\text{Checkpoint}\ \frac{}{(\kappa, N, V, \boldsymbol{checkpoint}(\boldsymbol{\omega}); c) \Rightarrow ((N|_{\omega}, V, c), N, V, c)}$$

List of locations to save

Copy only locations in the list

$$\text{Powerfail}\ \frac{}{(\kappa, N\ , V, c) \Rightarrow (\kappa, N, \emptyset, reboot)}$$

$$\text{Reboot}\ \frac{\kappa = (N'\ , V, c)}{(\kappa, N, \emptyset, reboot) \Rightarrow (\kappa, N\ \ \lhd N', V, c)}$$

… do this for all commands.

Prove the theorem

- An intermittent execution is a sequence of state transitions
- Show that after any transition, all memory locations either match the memory of the continuous execution or meet the conditions

Take-aways

- To build interesting applications, intermittent systems need to be robust to power failures of arbitrary position and duration
- One challenge is that inputs cause bugs generally not handled by existing systems
- Formalizing system behaviour and correctness definitions allow us to prove if a system is correct or not