Future of Computing: Beyond the Edge

December 5, 2019

18-213

Emily Ruppel

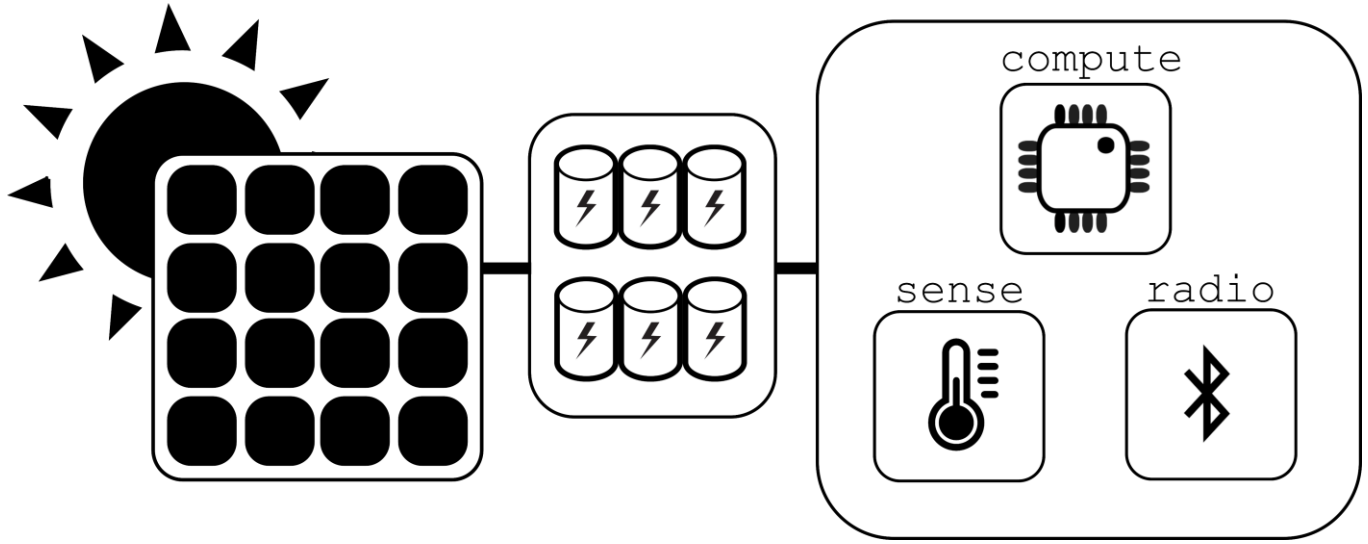
No batteries –> new environments

- Harsh, difficult-to-access environments
- Maintenance is expensive or intrusive

https://www.nasa.gov/mission_pages/Glory/multimedia/HighRes_Glory_Still03.html
https://www.sciencedaily.com/releases/2014/01/140120090428.htm
https://www.flickr.com/photos/michaelfoleyphotography/4978157199/

Computing beyond the edge

Battery-free
Energy Harvesting
Intermittently Powered

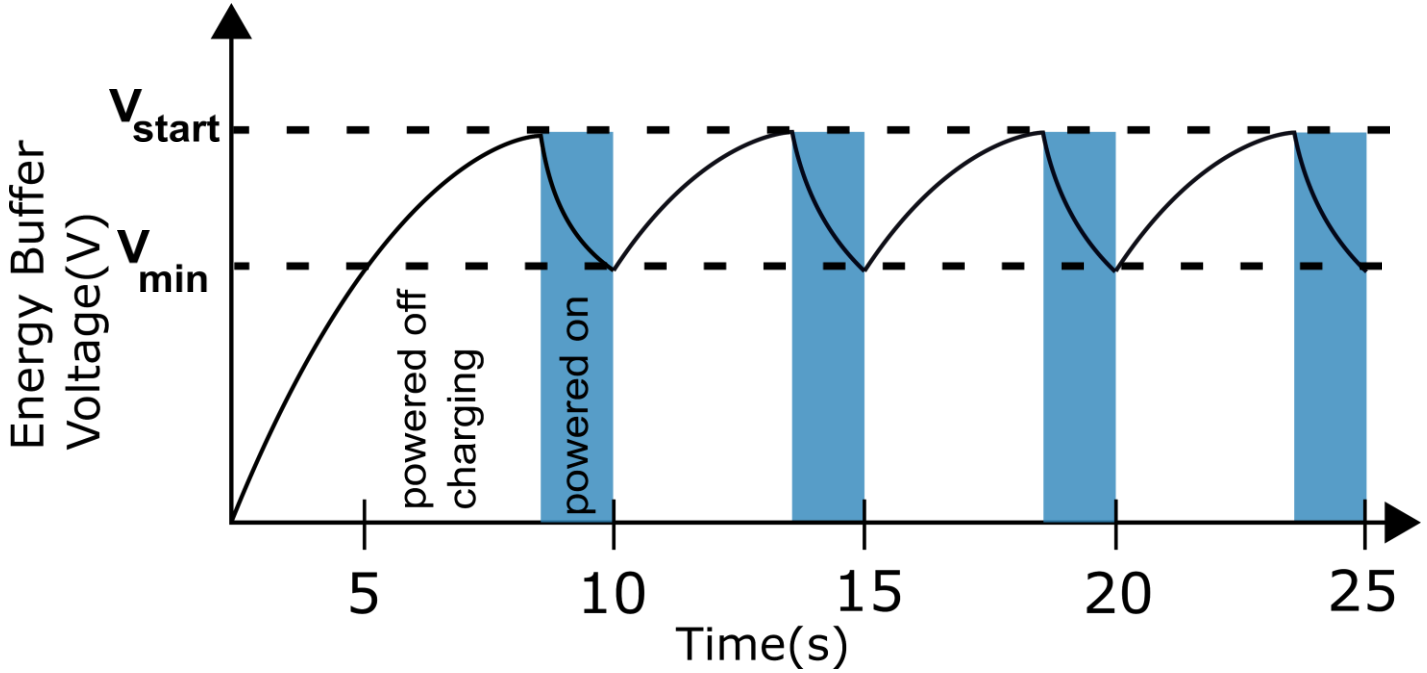


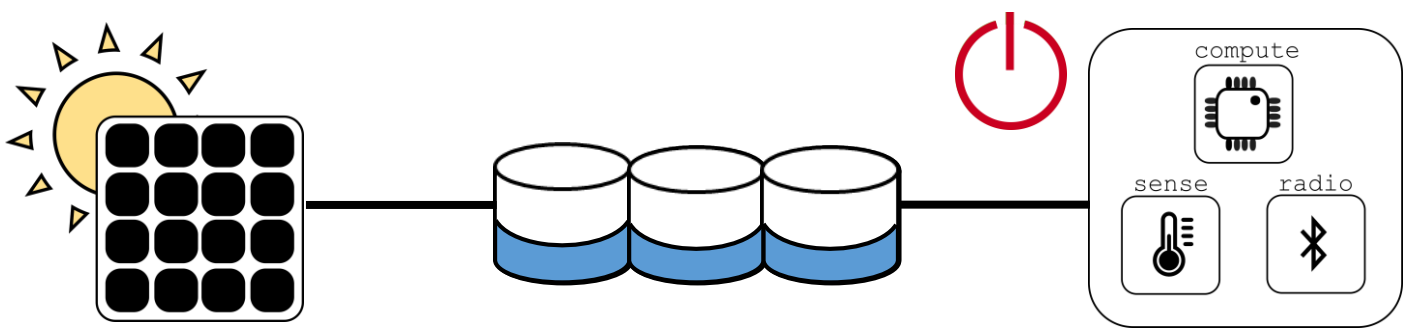
Harvester

Energy Buffer

MCU & peripherals

No batteries –> intermittent power supply





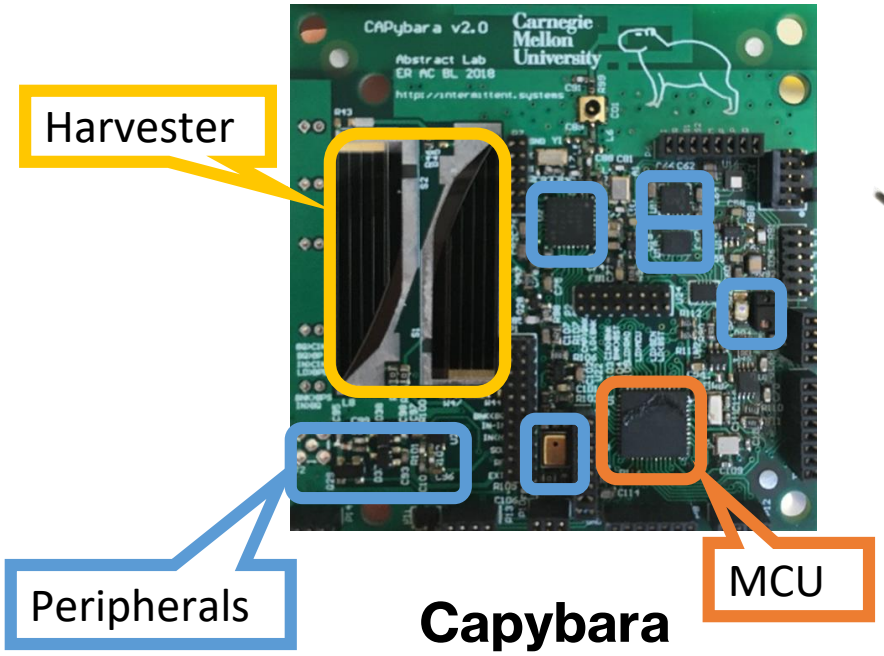
Outline

- Batteryless Device Basics
- Keeping Persistent Memory Consistent
- Task-Based Programming Models
- Asynchronous Energy Demands
- Capybara Platform
- Summary

Outline

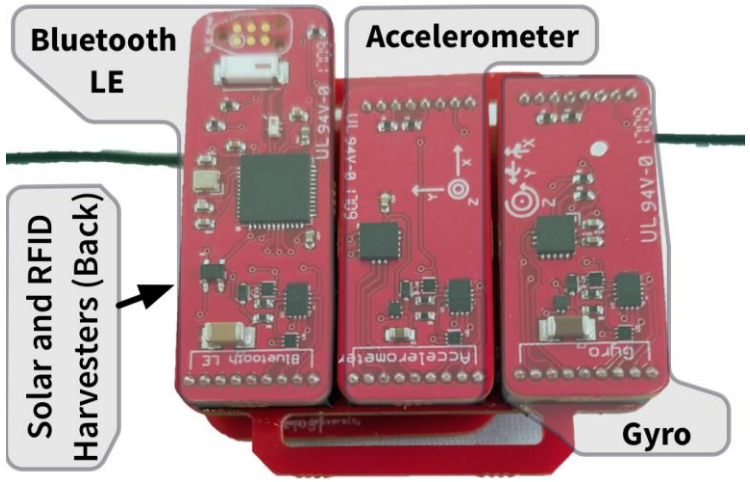
- Batteryless Device Basics
- Keeping Persistent Memory Consistent
- Task-Based Programming Models
- Asynchronous Energy Demands
- Capybara Platform
- Summary

Numerous batteryless platforms exist



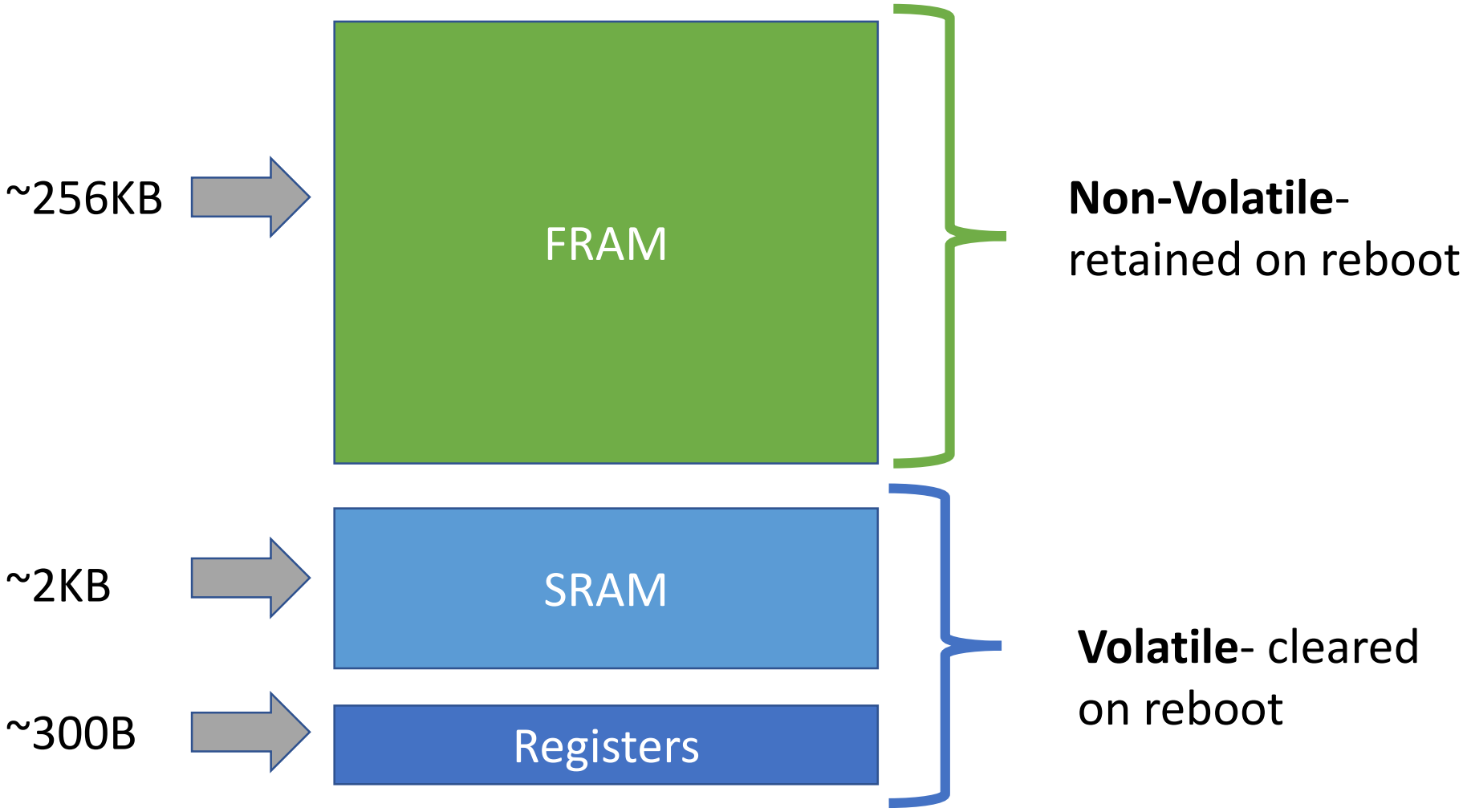
Capybara

WISP
University of Washington

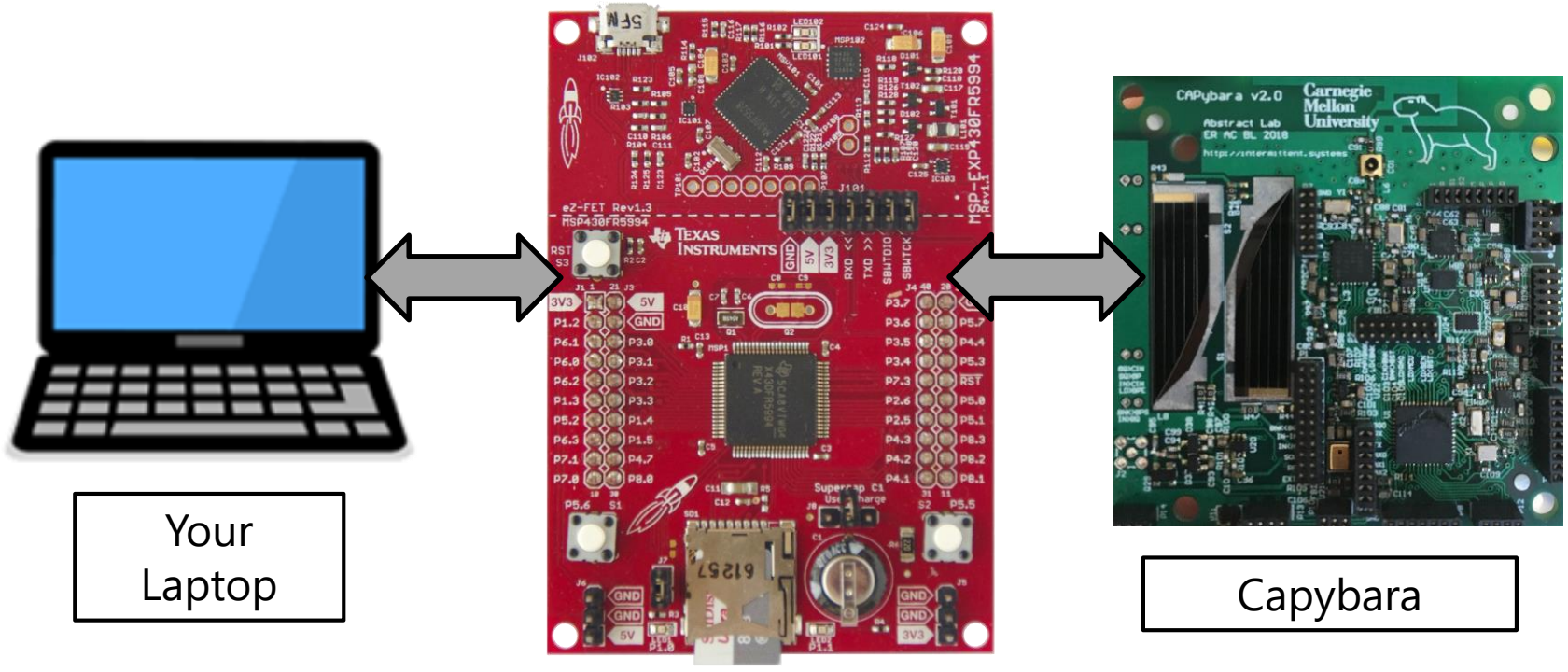


Flicker
Clemson University

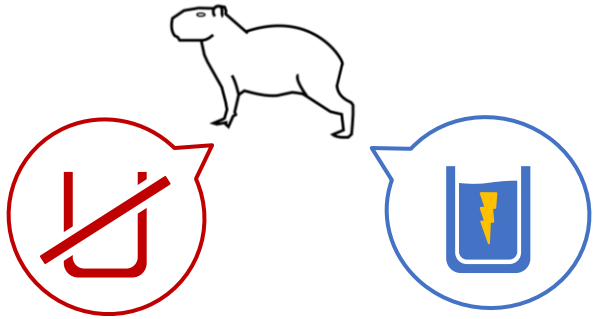
Batteryless devices typically have a hybrid memory hierarchy



Programming a batteryless sensor



Programs may not finish

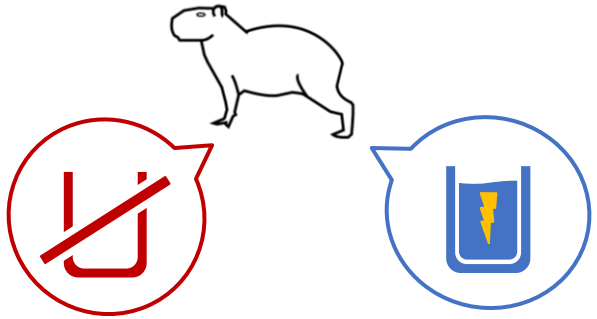


```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

```
count++
buf[count] = accel()
Power fail
```

Execution Time

Programs may not finish



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

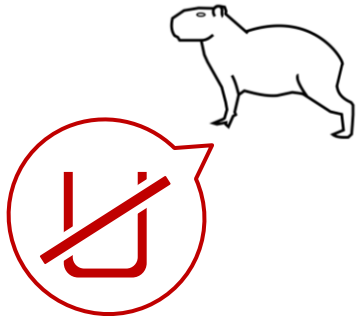
```
count++
buf[count] = accel()
```
Power fail

```
count++;
buf[count] = accel()
```
Power fail

.
.
.

Execution Time

Programs may not finish



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

```
count++
buf[count] = accel()
```
Power fail

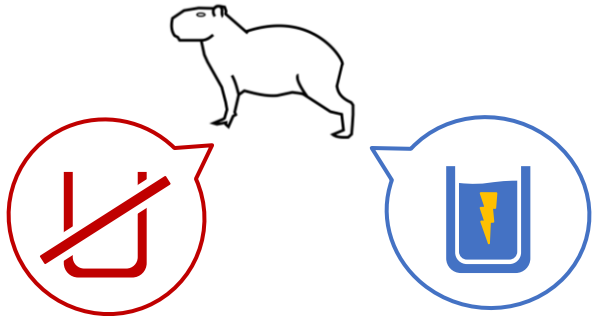
```
count++;
buf[count] = accel()
```
Power fail

. . .

Execution Time

Need to latch execution state periodically!

Programs may not finish



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

Execute with checkpoints

```
count++
buf[count] = accel()
```
Power fail

```
count++;
buf[count] = accel()
```
Power fail

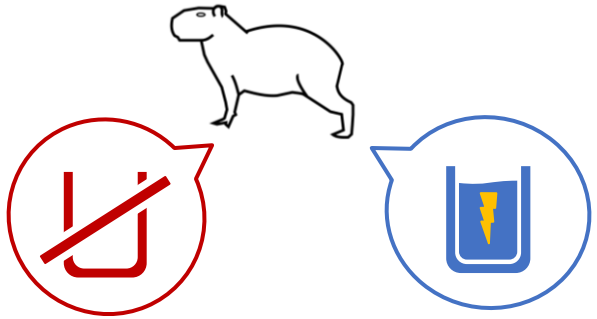
. . .

Execution Time

```
count++
```
Checkpoint
```
buf[count] = accel()
```
Power fail

Need to latch execution state periodically!

Programs may not finish



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

Execute with checkpoints

```
count++
buf[count] = accel()
```
Power fail

```
count++;
buf[count] = accel()
```
Power fail

. . .

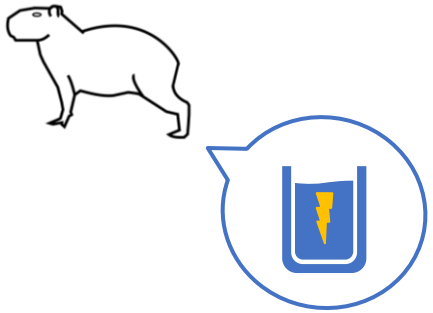
Execution Time

```
count++
```
Checkpoint
```
buf[count] = accel()
```
Power fail

```
buf[count] = accel()
avg = sum(buf)/count
```
Checkpoint
```
transmit-
```
Power fail

Need to latch execution state periodically!

Programs may not finish



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

Execute with checkpoints

```
count++
buf[count] = accel()
```
Power fail

```
count++;
buf[count] = accel()
```
Power fail

. . .

Need to latch execution state periodically!

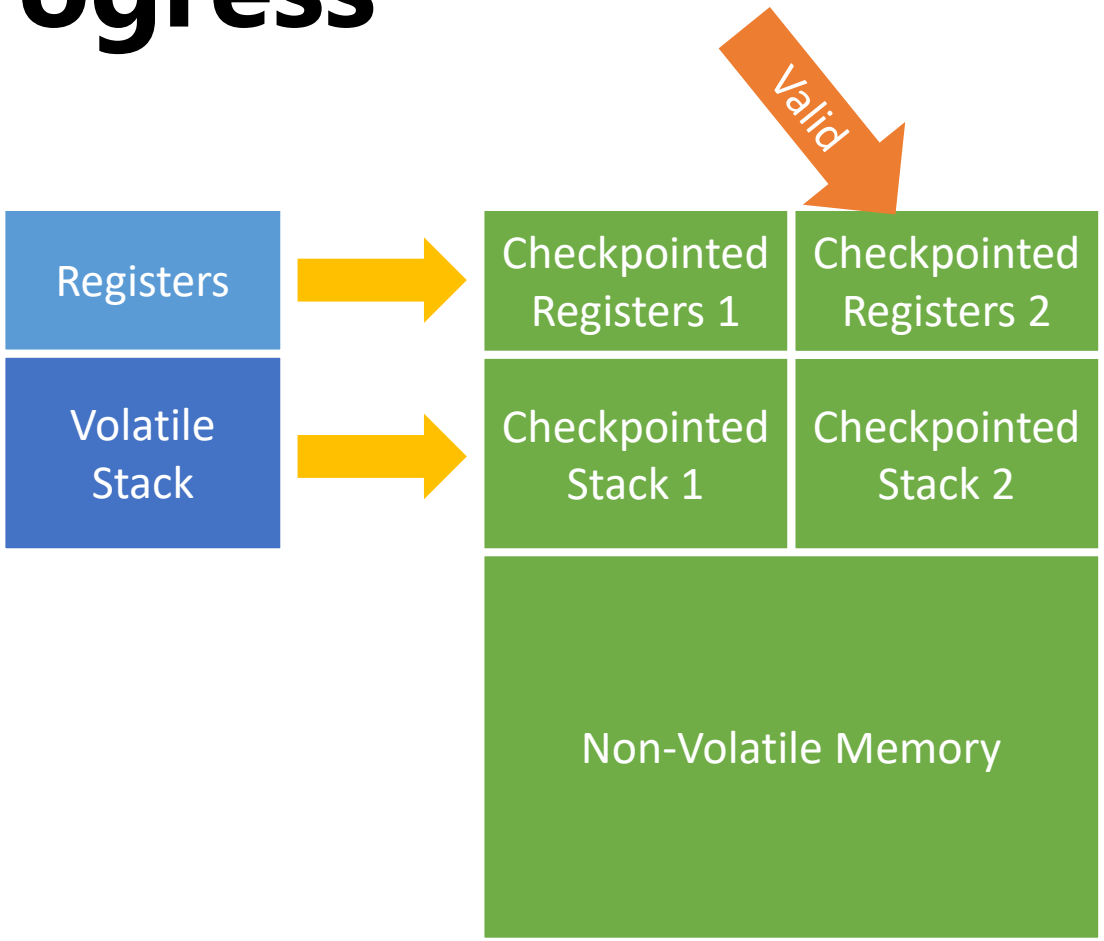
Execution Time

```
count++
```
Checkpoint
```
buf[count] = accel()
```
Power fail

```
buf[count] = accel()
avg = sum(buf)/count
```
Checkpoint
```
transmit-
```
Power fail

```
transmit(avg)
```

Checkpointing allows the program to retain progress

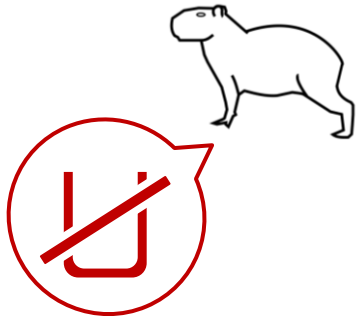


This strategy does not manage non-volatile memory correctly!

Outline

- Batteryless Device Basics
- Keeping Persistent Memory Consistent
- Task-Based Programming Models
- Asynchronous Energy Demands
- Capybara Platform
- Summary

Memory can become inconsistent



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

NV Memory

count | 0 |

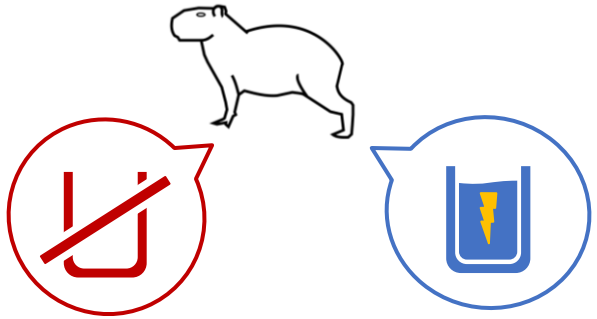
buf | 7 | 0 | 0 |

```
count++

buf[count] = accel()
```

Execution Time

Memory can become inconsistent



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

NV Memory

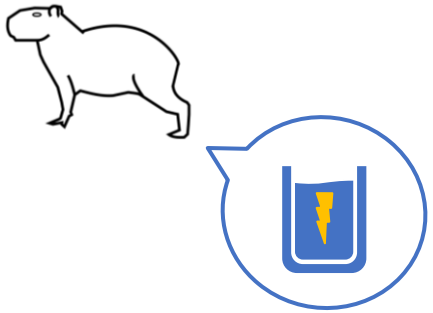
count 1

buf 7 0 0

```
count++
Power fail
buf[count] = accel()
```

Execution Time

Memory can become inconsistent



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

NV Memory

count 1

buf 7 0 0

```
count++
```
Power fail
```
buf[count] = accel()
```

count 2

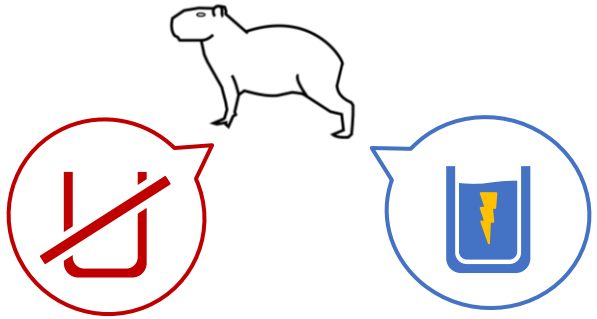
buf 7 0 9

```
count++;
buf[count] = accel()
```

Execution Time

Need to track or prevent potentially inconsistent variables!

Memory can become inconsistent



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

Make working copy of inconsistent variables!

NV Memory

count 1

buf 7 0 0

```
count++
Power fail
buf[count] = accel()
```

count 2

buf 7 0 9

```
count++;
buf[count] = accel()
```

Execution Time

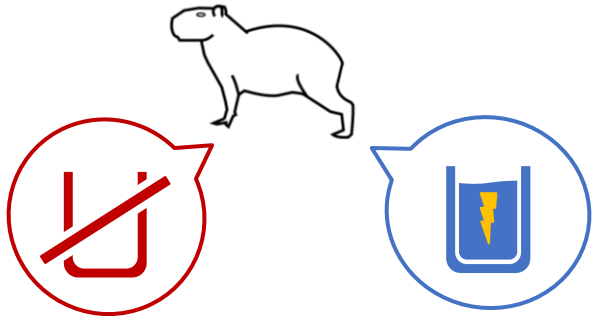
count 1

buf 7 0 0

```
count++
Power fail
buf[count] = accel()
```

Need to track or prevent potentially inconsistent variables!

Memory can become inconsistent



```
int process() {
    count++;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

Make working copy of inconsistent variables!

NV Memory

count 1

buf 7 0 0

```
count++
Power fail
buf[count] = accel()
```

count 2

buf 7 0 9

```
count++;
buf[count] = accel()
```

Execution Time

count 1

buf 7 0 0

```
count++
Power fail
buf[count] = accel()
```

count 1

buf 7 9 0

```
count++;
buf[count] = accel()
```

Need to track or prevent potentially inconsistent variables!

Common Intermittent Execution Models

Checkpoints

```
int main() {
    temp = count + 1;
    count = temp;
    buf[count] = accel();
    avg = sum(buf)/count;
    transmit(avg);
}
```

Voltage, time, program structure, WARs

Easy to use, but arbitrarily placed checkpoints can break I/O

Tasks

```
task sample() {
    count++;
    buf[count] = accel();
    task_transition(compute)
}
```

All-or-nothing semantics, logging for NVM accesses

```
task compute() {
    avg = sum(buf)/count;
    transmit(avg);
}
```

More programmer involvement, but much more control

B. Ransford, J. Sorber, and K. Fu. *Mementos: System support for long-running computation on RFID-scale devices.* ASPLOS, 2011.

J. Hester, K. Storer, and J. Sorber. *Timely execution on intermittently powered batteryless sensors.* SenSys, 2017.

D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli and L. Benini. *Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. IEEE ESL,* 2015.

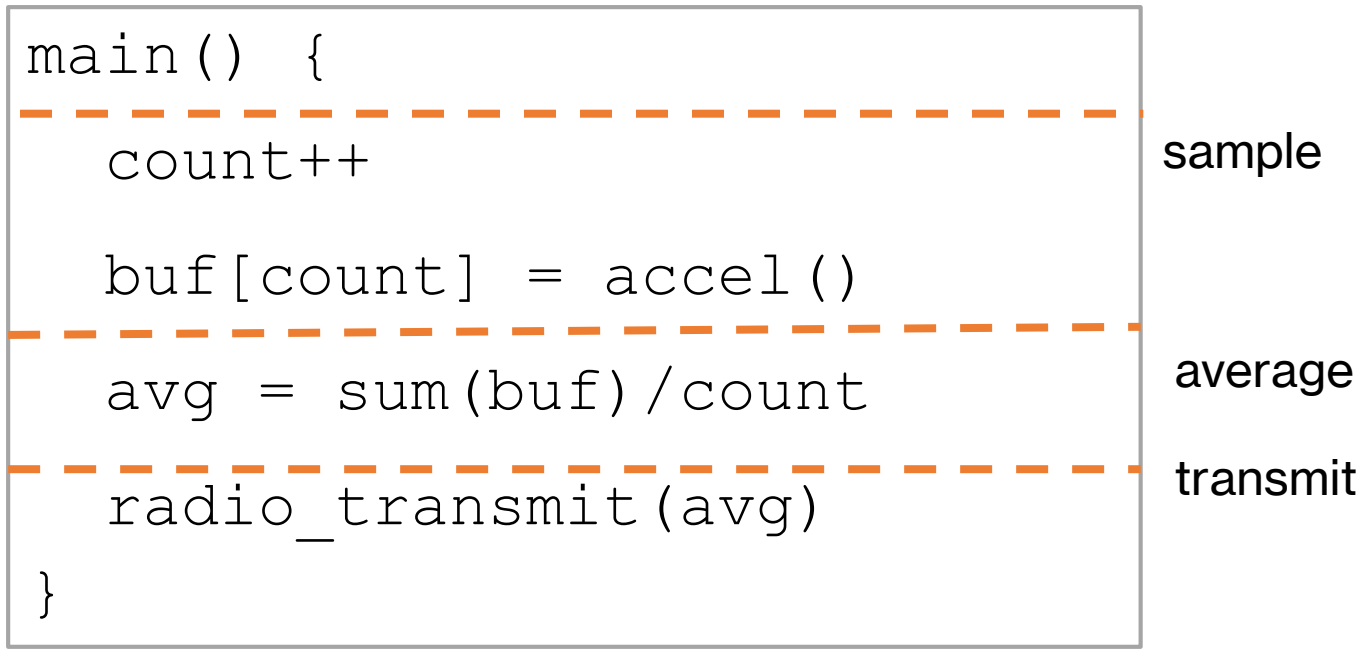
B. Lucia and B. Ransford. *A simpler, safer programming and execution model for intermittent systems.* PLDI, 2015.

J. Van Der Woude and M. Hicks. *Intermittent computation without hardware support or programmer intervention.* OSDI, 2016.

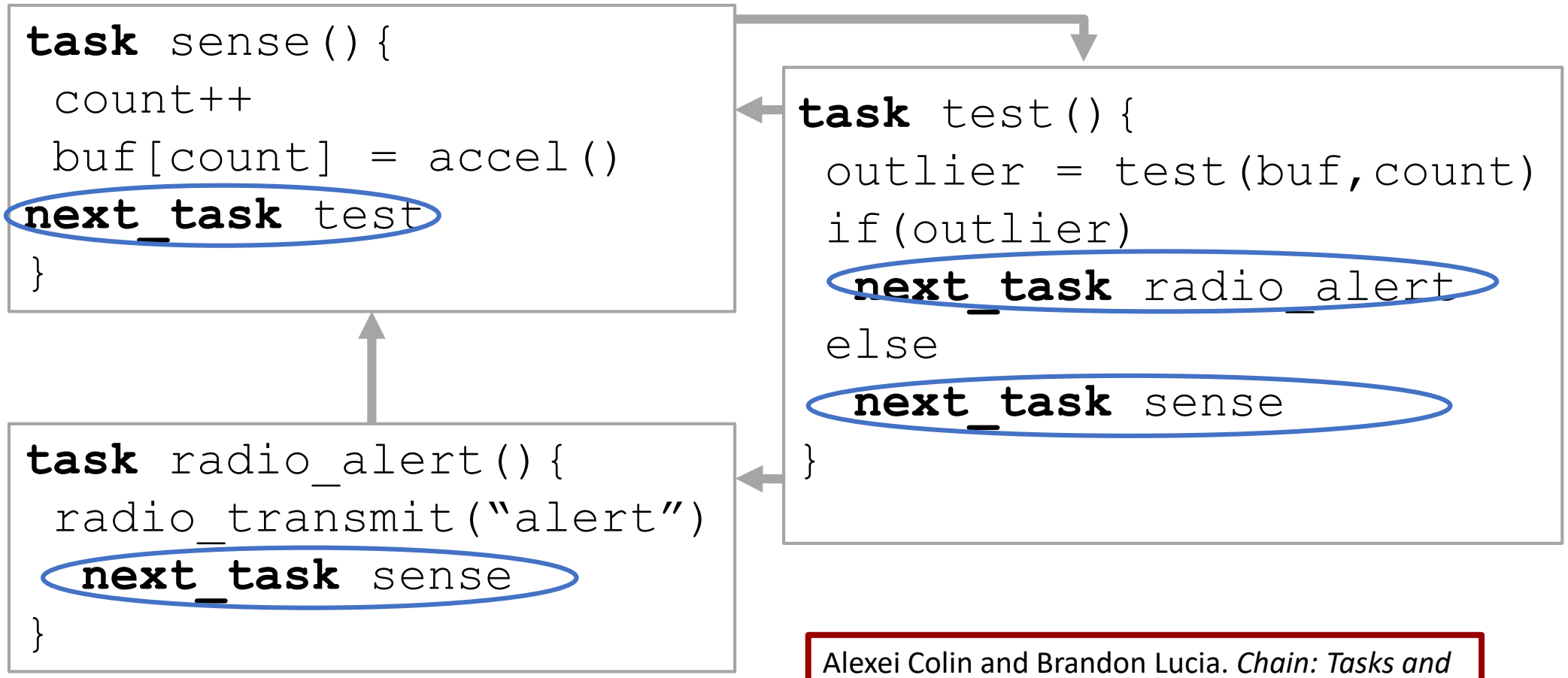
Outline

- Batteryless Device Basics
- Keeping Persistent Memory Consistent
- Task-Based Programming Models
- Asynchronous Energy Demands
- Capybara Platform
- More Challenges
- Summary

Code can be decomposed into tasks



Task control flow is explicit



Alexei Colin and Brandon Lucia. *Chain: Tasks and Channels for Reliable Intermittent Programs. OOPSLA.* 2016.

Data is transferred using shared variables

```
TASK_SHARED(...)
TASK_
```

Only access that the runtime systems needs to handle

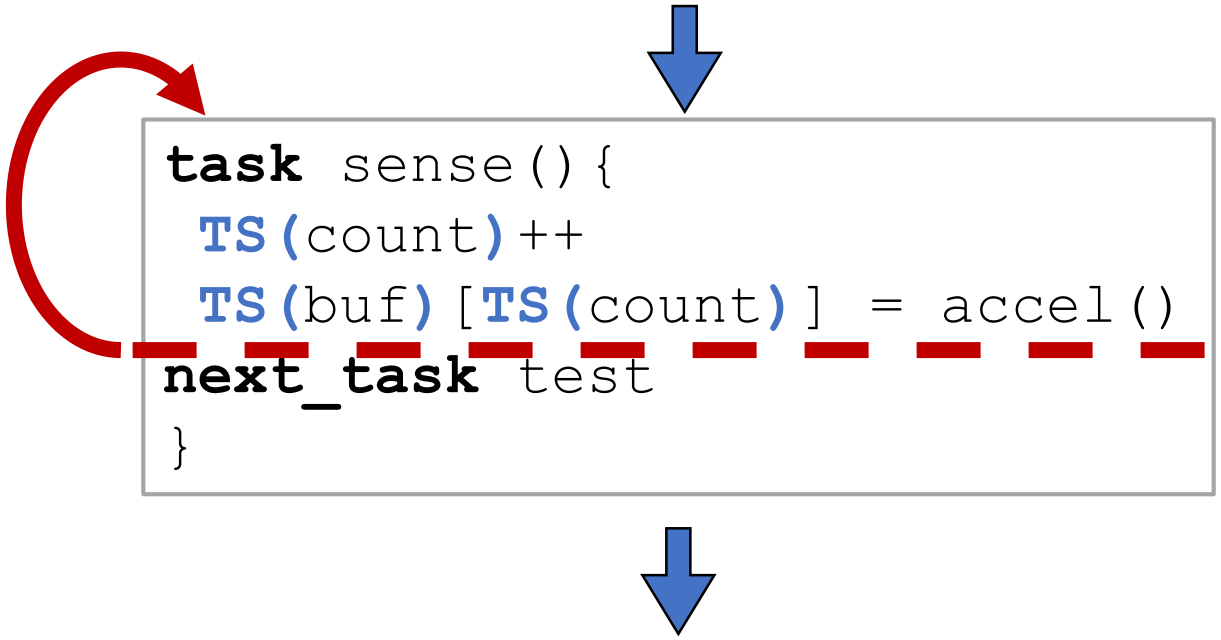
```
task se  e(){
 TS(count)++
 TS(buf)[TS(count)] = accel()
next_task test
}
```

```
task test(){
 outlier = test(TS(buf),TS(count))
 if(outlier)
  next_task radio_alert
 else
  next_task sense
}
```

```
task radio_alert(){
  radio_transmit("alert")
  next_task sense
}
```

Kiwan Maeng and Brandon Lucia. *Alpaca: Intermittent Execution Without Checkpoints.* OOPSLA. 2017.

Tasks provide atomic semantics

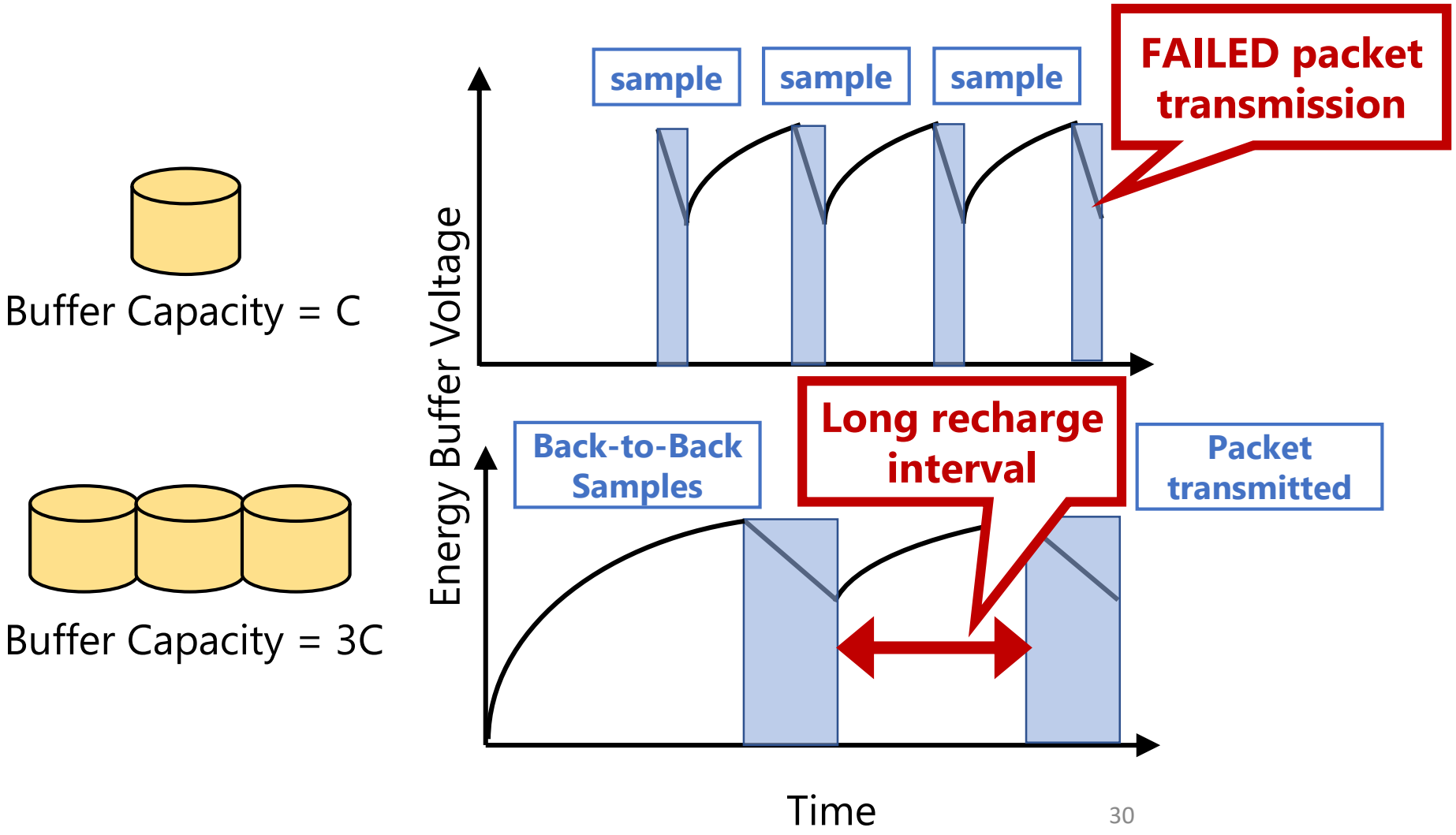


Kiwan Maeng and Brandon Lucia. *Alpaca: Intermittent Execution Without Checkpoints.* OOPSLA. 2017.

Outline

- Batteryless Device Basics
- Keeping Persistent Memory Consistent
- Task-Based Programming Models
- Asynchronous Energy Demands
- Capybara Platform
- Summary

Fixed energy buffers are fixed



Applications may have conflicting constraints

- Temporal constraints
- Energy capacity constraints

Temporal constraint

BOTH

Capacity constraint

```
while(light > CLOSE_OBJECT) {
   light = read_photoresistor();
}
gesture = capture_gesture();
radio_transmit(gesture);
}
```

Outline

- Batteryless Device Basics
- Keeping Persistent Memory Consistent
- Task-Based Programming Models
- Asynchronous Energy Demands
- Capybara Platform
- Summary

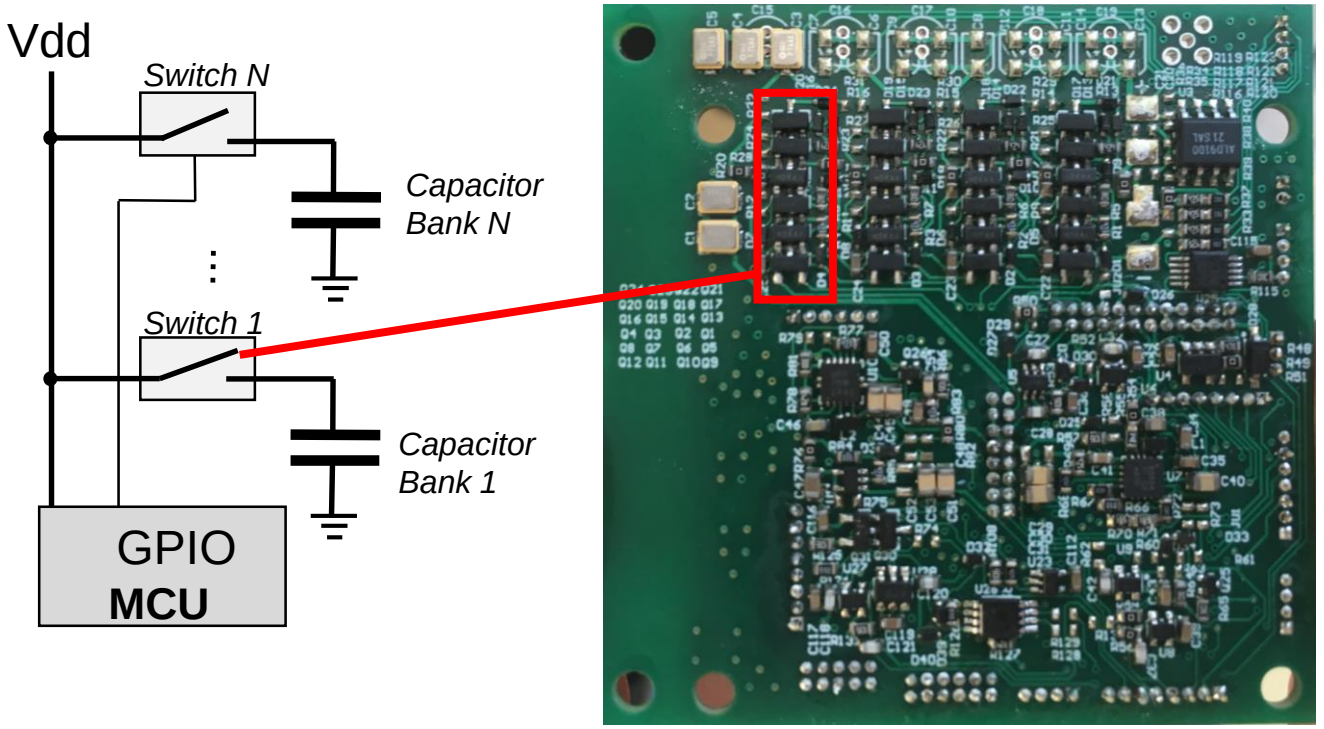
Capybara relies on HW-SW codesign

- Hardware
  - Flexible power system
  - Reconfigurable capacitor bank
- Software
  - Runtime to manage state changes
  - Declarative interface for energy modes

Capybara relies on HW-SW codesign

- Hardware
  - Flexible power system
  - Reconfigurable capacitor bank
- Software
  - Runtime to manage state changes
  - Declarative interface for energy modes

The reconfigurable capacitor bank is controlled with software



Energy modes map to temporal and capacity constraints

Temporal constraint

BOTH

Capacity constraint

```
while(light > CLOSE_OBJECT) {
   light = read_photoresistor();
}
gesture = capture_gesture();
radio_transmit(gesture);
}
```

Tasks are annotated with their energy mode

Preburst:exec=LOW,preburst=HIGH

```
task_sample(){
 light = read_photoresistor()
 if(light < CLOSE_OBJECT)
  TRANSITION_TO(task_gesture)
 else
  TRANSITION_TO(task_sample)
}
```

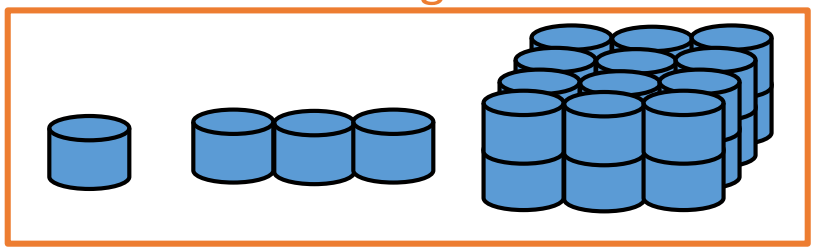
Burst:burst=HIGH

```
task_gesture() {
 TS(gesture) = get_gesture()
 if(TS(gesture) > NONE)
  TRANSITION_TO(task_send)
 else
  TRANSITION_TO(task_sample)
}
```

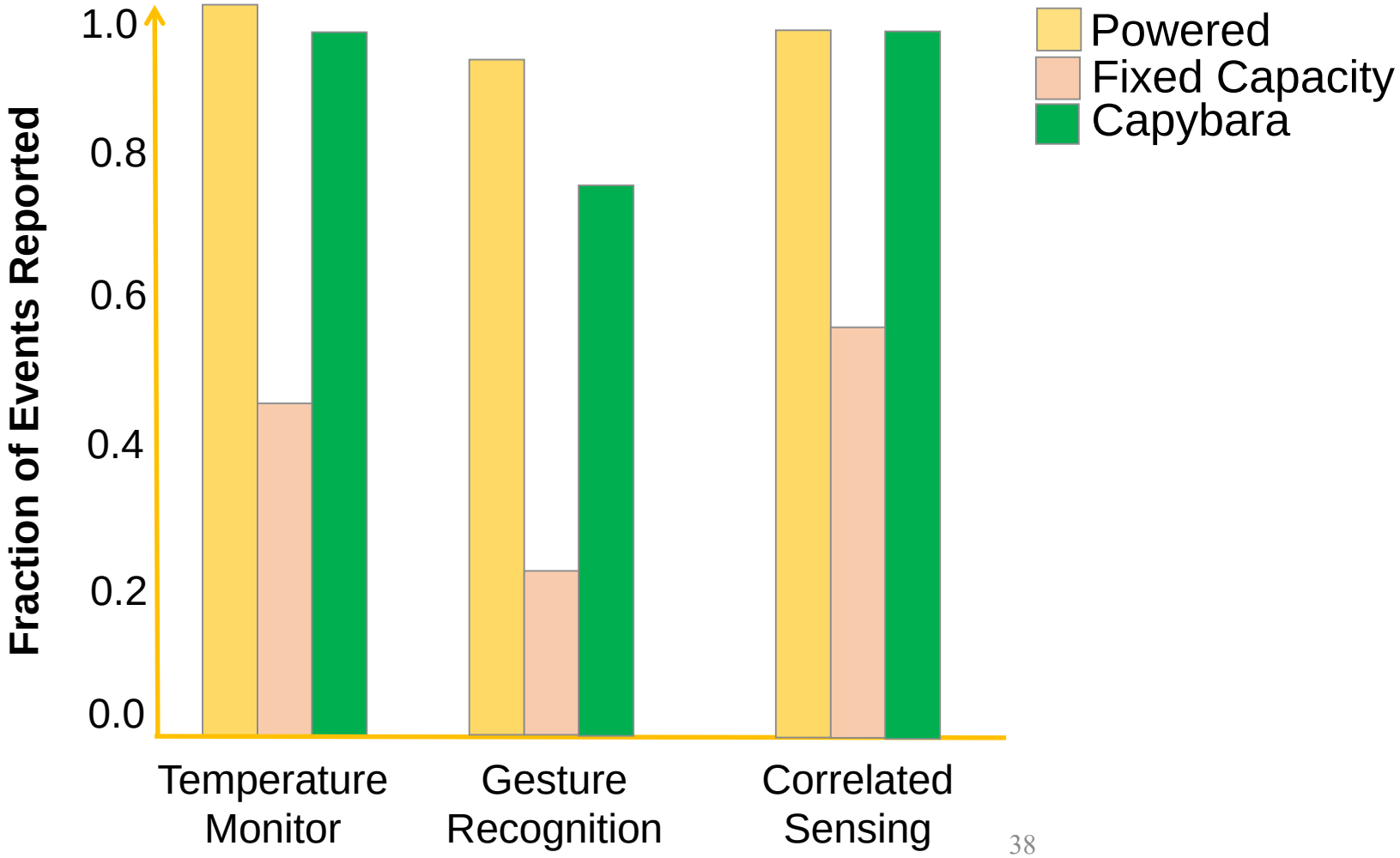
Config:exec=MED

```
task_send() {
 radio_send(TS(gesture))
TRANSITION_TO(task_sample)
}
```

Buffer Configuration



Capybara improves event detection



Programming models simplify handling the physical world beyond the edge

- Intermittent execution can cause persistent memory to become inconsistent
- Task-based programming models provide forward progress and memory consistency
- Fixed-size energy buffers preclude reactive applications with energy capacity constraints
- Capybara's hardware/software codesign enables reactive applications