INTOS: Persistent Embedded OS and Language Support for Multi-threaded Intermittent Computing

Yilun Wu[*]
Wenjie Xiong[‡]

Byounguk Min[†]
Changhee Jung[†]

Mohannad Ismail[‡]
Dongyoon Lee[*]

[*] Stony Brook University

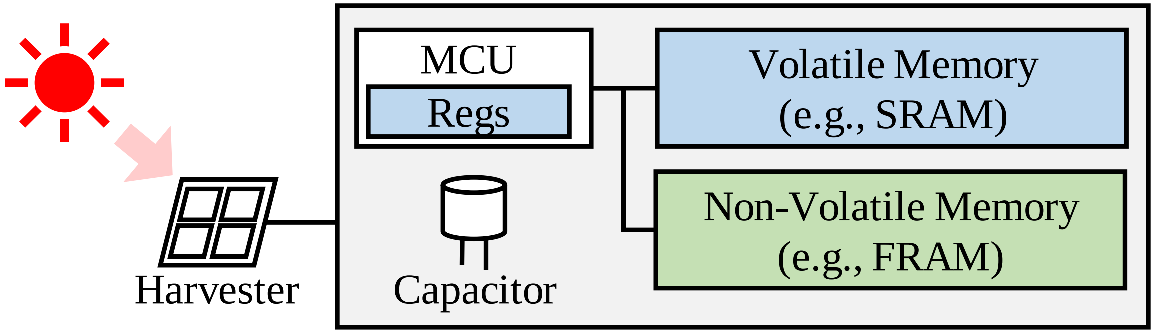
[‡] VIRGINIA TECH.

[†] PURDUE UNIVERSITY®

Background: Intermittent Computing

Energy harvesting system

(e.g. env. sensors, nano satellites…)

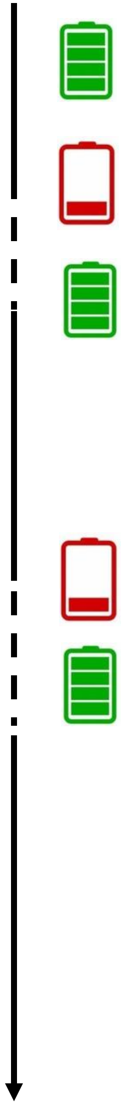


On Crash ⚡ Regs + SRAM states are lost

After recharging 🔋, system reboots

=> Need crash consistency

Intermittent program execution



```
fn task_recognize(model: Model) {
  let q = sys_create_queue();
  let stats = sys_alloc();
```

Crash ⚡

```
while(...) {
    let reading =sys_read(SENSOR);
    let window = [0; 3]
    init_window(&reading)
    transform(&mut window)
```

Crash ⚡

```
    let feature = featurize(&window)
    let class = classify(&feature,
&model)
    stats[class] += 1

    sys_queue_send(q, class, TIME_OUT);
  }
}
```

No Embedded OS for Intermittent Computing

- **Embedded OS**

  e.g., threads, queues, semaphores, events, software timers

  + Improved MCU utilization => better energy utilization

  + Improved HW multiplexing

  + Easier programming for async multi-tasking

- **Existing embedded OSes**

  e.g., FreeRTOS, Tock

  – NOT crash safe

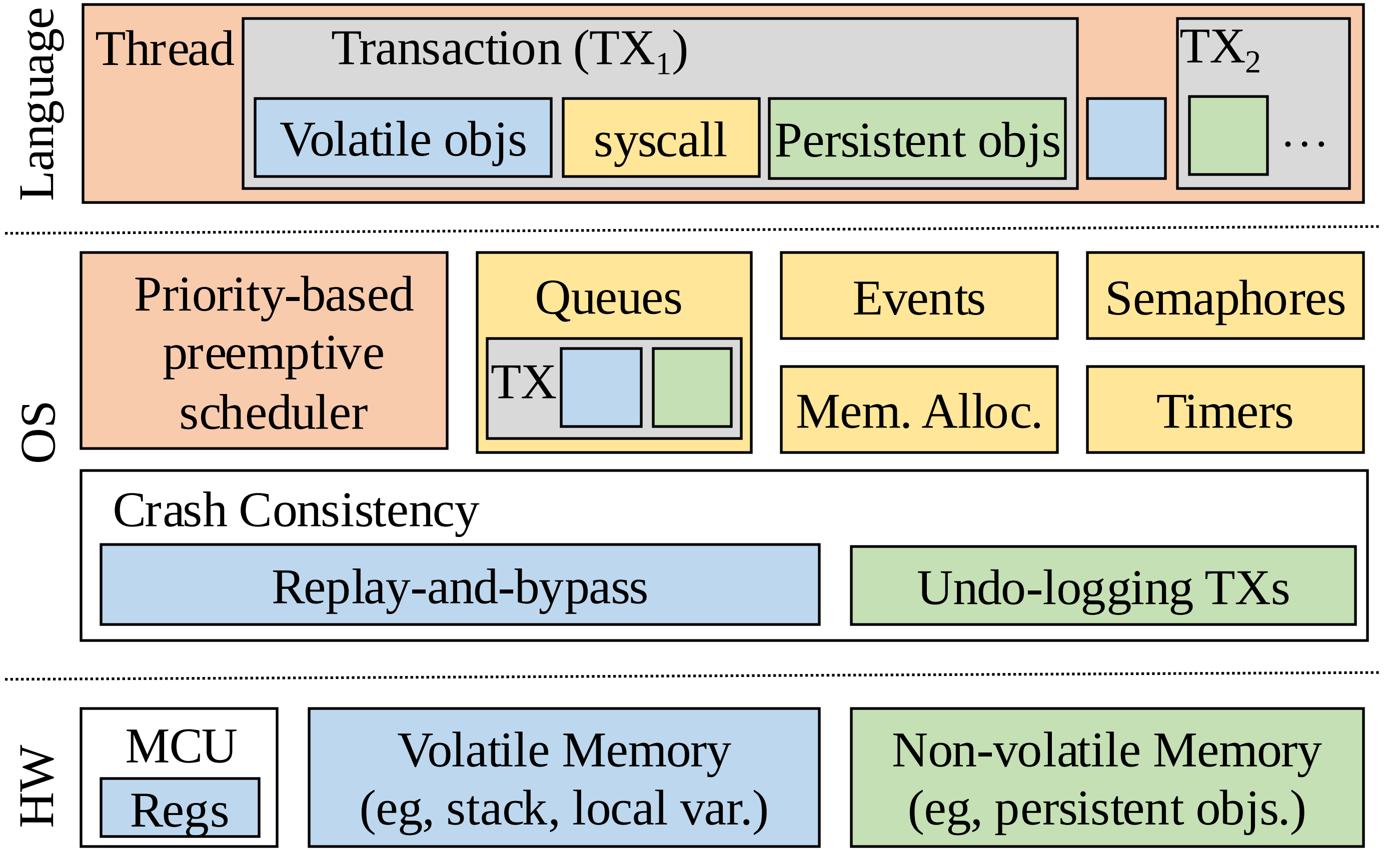
Prior Crash Consistency Solutions for Embedded OS?

- **Idempotent processing**: e.g., Ratchet [OSDI'16]
  - + Transparent
  - – NVM only. slow. less energy efficient

- **Micro-continuation**: e.g., Immortal Thread [OSDI'22]
  - + (Almost) transparent. some threading support
  - – NVM only. slow. less energy efficient

- **Manual task-decomposition**: e.g., Alpaca [OOPSLA'17]
  - + Good performance
  - – Huge manual efforts

Overview of IntOS



- **Threads** (multithreading)
- **OS**
  - queues
  - events
  - semaphores, and more
- **HW**
  - volatile registers
  - volatile memory
  - non-volatile memory
- **Crash Consistency**
  - Transactions NVM
  - Replay-and-bypass

    Regs SRAM

Transactions

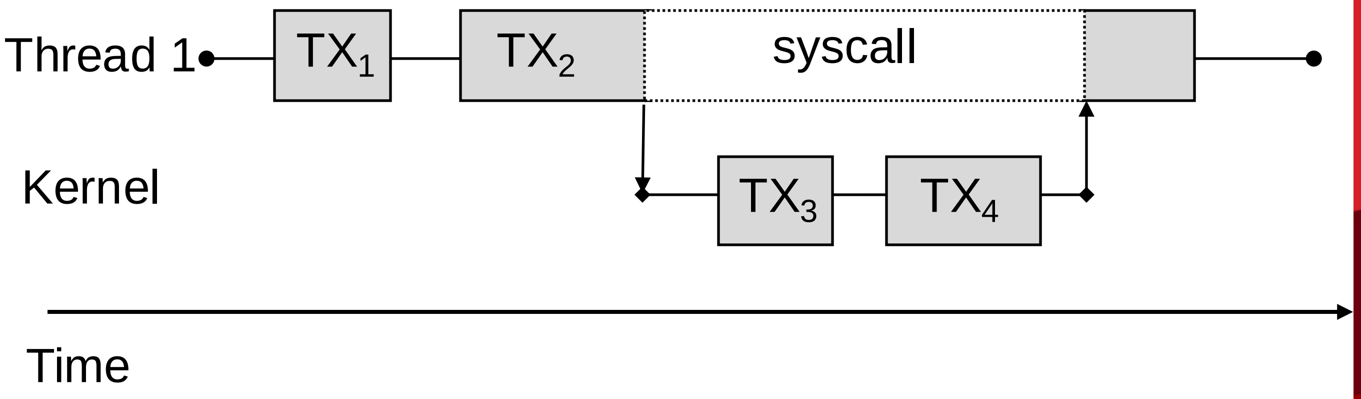
```
fn task_recognize(model: Model) {
  let q,stats = transaction::run(|j,t|
    q = sys_create_queue(Q_SZ);
    let stats = PBox::new(…);
    return (q,stats);
  );

  // while loop removed for simplicity
  transaction::run(|j,t| {
    let reading = read(SENSOR);
    let window = [0; 3]
    init_window(&reading)
    let feature = featurize(&window)
    let class = classify(&feature, &model)
    // automatic Undo-logging
    let stats_ref = stats.as_mut(j);
    *stats_ref[class] += 1
    sys_queue_send(q, class, TIME_OUT);
  });
}
```

TX_1

TX_2

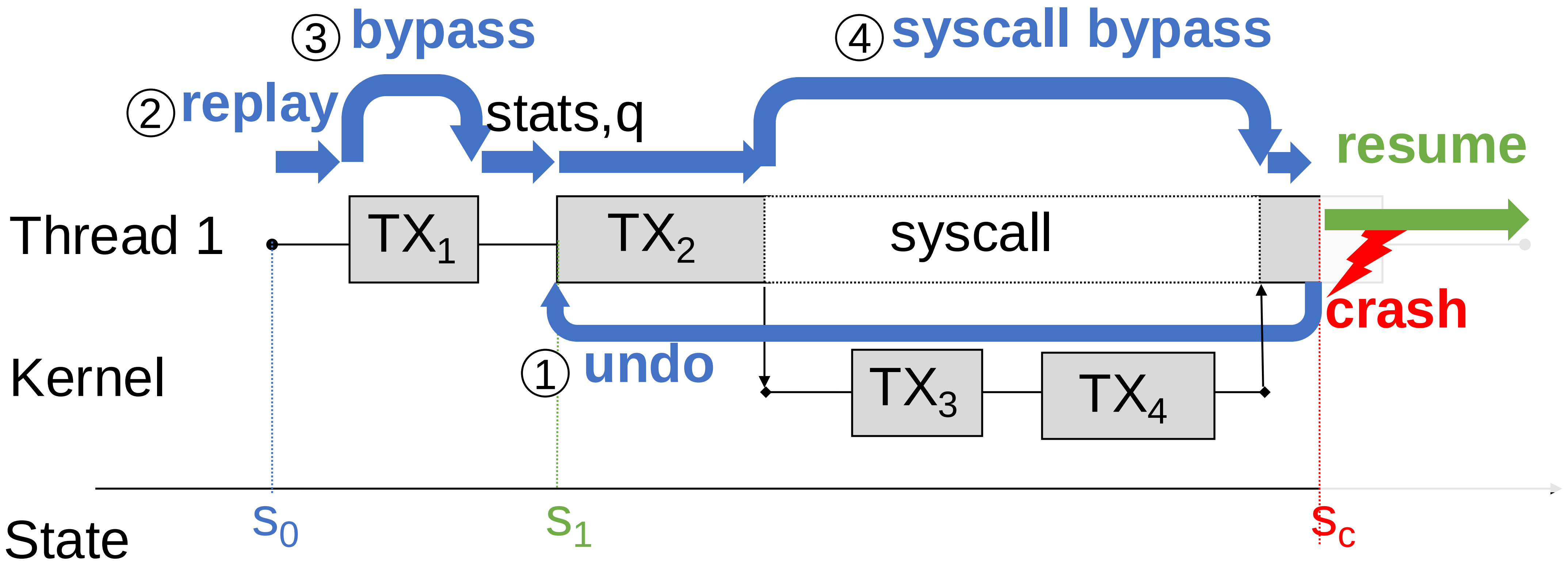
Example task (thread)

A thread includes transactions (TX_1 and TX_2) for persistent objects.



A syscall “sys_queue_send” contains transactions (TX_3 and TX_4) in the kernel

Replay-and-Bypass (single-thread)



User TX Replay Table

ID	Return Val
1	stats, q

Kernel TX Replay Table

ID	Return Val
1	0xABCD
2	0x0

Syscall Replay Table

ID	Return Val
1	0x0

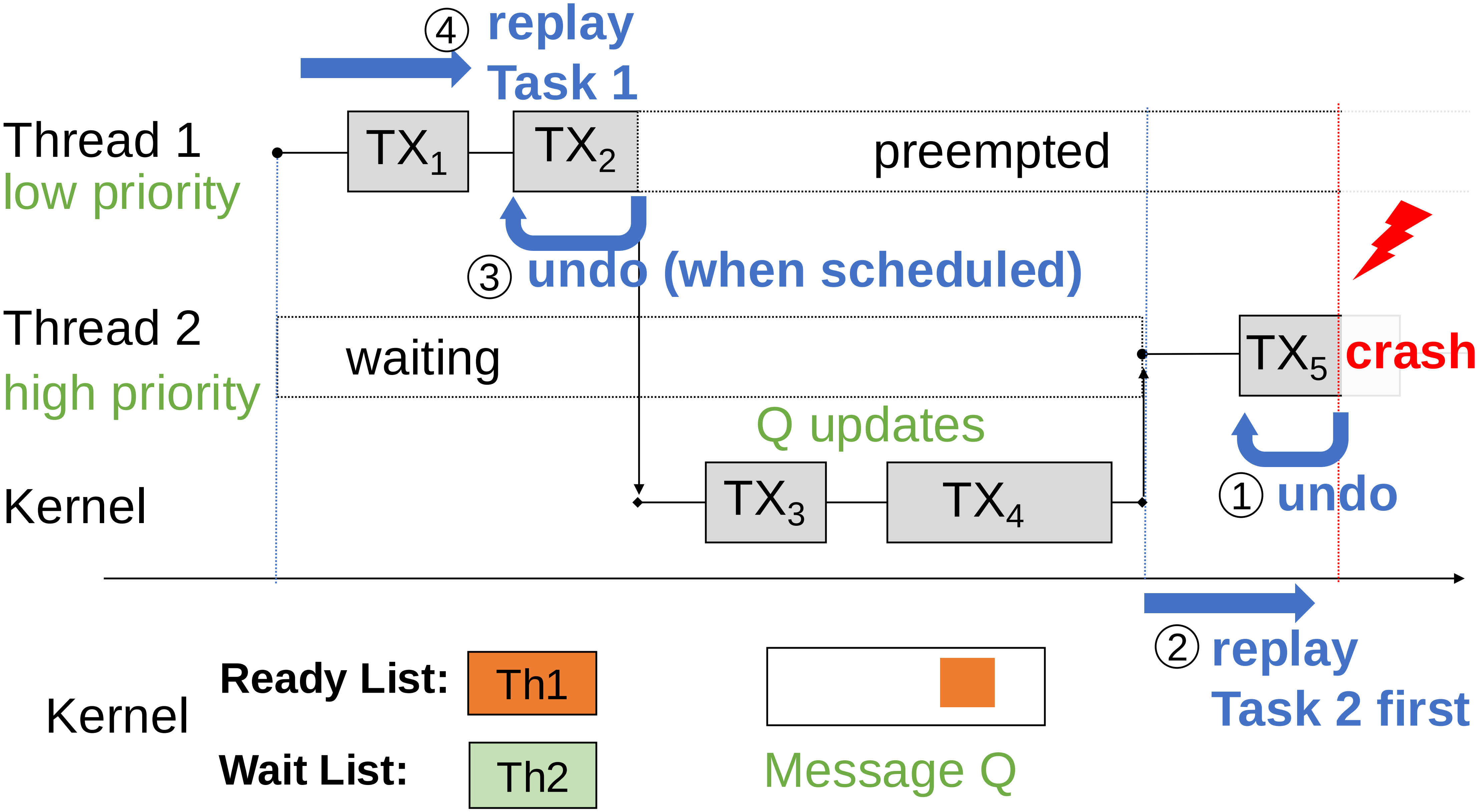
Crash Consistency Support for Multithreading

- **Kernel maintains multiple ready-lists and wait-lists**
  - ⇒ Uses roll-forward crash consistency solution for efficiency (see §7 in paper for detail)

- **Tasks interact through shared kernel objects** (e.g., queue, semaphores)
  - ⇒ Uses kernel-level transactions for crash consistency
  - ⇒ Recovers the interrupted kernel transaction before resuming a user thread

- **Tasks have different priorities**
  - ⇒ Recovers the ready task with highest priority first
  - ⇒ Lazily recovers other tasks (when they are scheduled later)

Multi-thread Crash Consistency



④ replay Task 1

Thread 1 low priority

TX$_1$ TX$_2$ preempted

③ undo (when scheduled)

Thread 2 high priority

waiting

TX$_5$ crash

Q updates

Kernel

TX$_3$ TX$_4$

① undo

② replay Task 2 first

Kernel

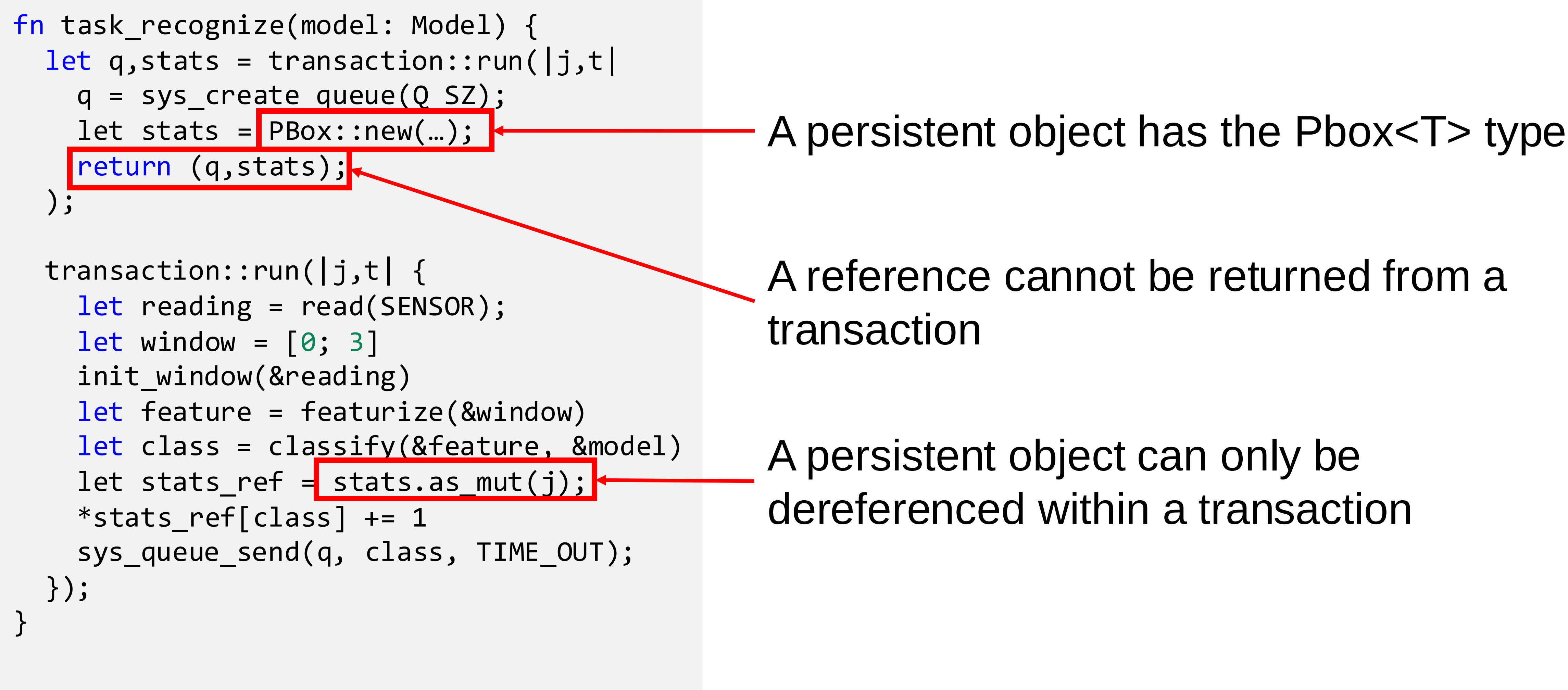
Ready List: Th1

Wait List: Th2

Message Q

Programming Model (enforced by Rust)

Example program



```
fn task_recognize(model: Model) {
  let q,stats = transaction::run(|j,t|
    q = sys_create_queue(Q_SZ);
    let stats = PBox::new(…);
    return (q,stats);
  );

  transaction::run(|j,t| {
    let reading = read(SENSOR);
    let window = [0; 3]
    init_window(&reading)
    let feature = featurize(&window)
    let class = classify(&feature, &model)
    let stats_ref = stats.as_mut(j);
    *stats_ref[class] += 1
    sys_queue_send(q, class, TIME_OUT);
  });
}
```

A persistent object has the Pbox<T> type

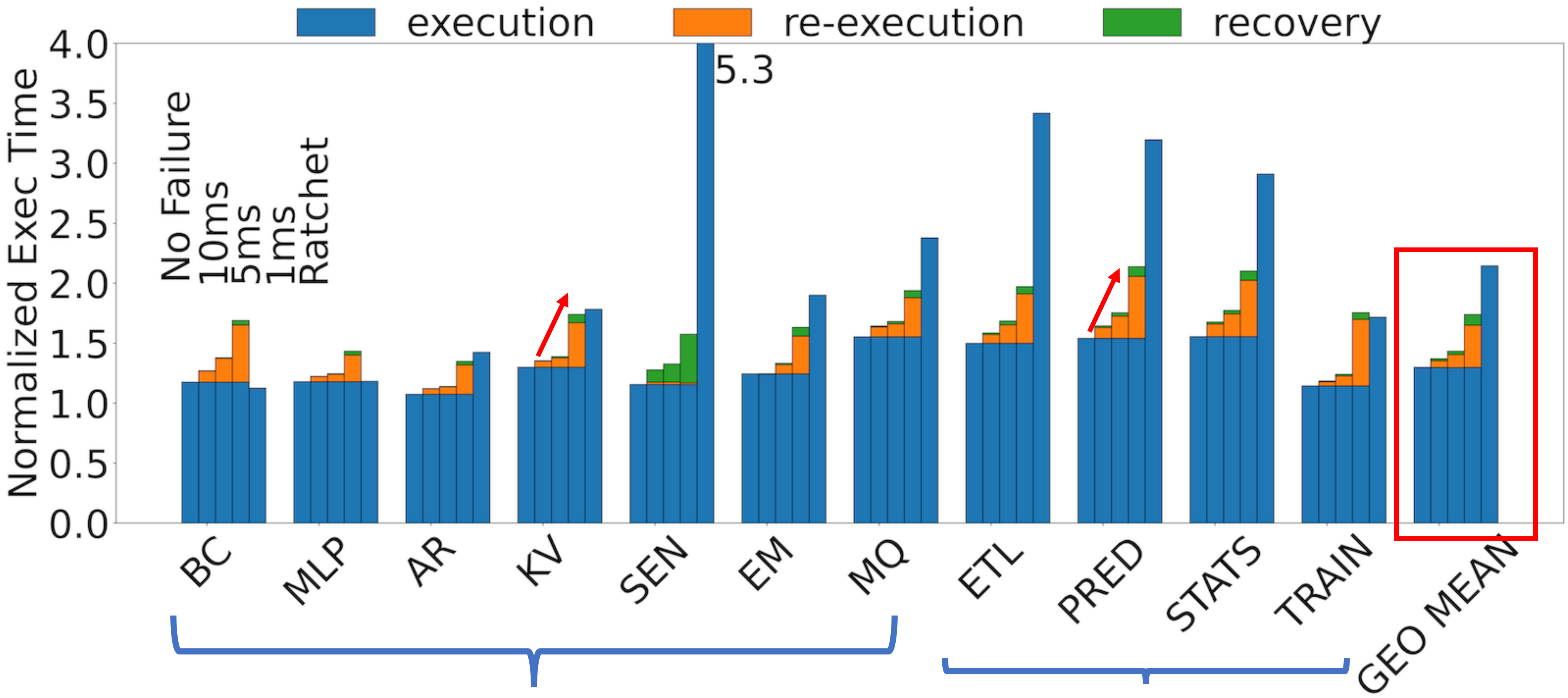
A reference cannot be returned from a transaction

A persistent object can only be dereferenced within a transaction

Evaluation Methodology

- **Benchmarks**
  - Seven micro-benchmark applications (1- 4 tasks per app)
    - Activity Recognition, KV Store, Sensing, Multi-layer Perception, etc.
  - Four RIOTBench applications [CCPE’17] (> 4 tasks per app)
    - IOT data stream processing: e.g., stats, prediction, train, etc.
- **Baseline**
  - Ratchet [OSDI’16] partitions and transforms a program into idempotent regions for crash consistency.
  - Employs FRAM (NVM) only
- **Testbed**
  - MSP430FR5994(MSP FRAM+SRAM)
  - Apollo 4 Blue Plus (ARM, Hybrid Mem)

Evaluation with Power Failures



Micro-benchmark

RIOTBench

Conclusion

- **Functionality:** IntOS is the first embedded (best-effort real time) OS that is crash safe and supports priority-based preemptive multithreading in intermittent computing setting.

- **Efficiency**: IntOS can make progress under frequent power failures at lower runtime and energy overheads than prior works.

- **Safety**: IntOS ensures whole system consistency including both volatile and non-volatile system states using Rust-based type system.