Surviving Peripheral Failures in Embedded Systems

Rebecca Smith and Scott Rixner
Motivation

● Embedded systems are ubiquitous

● They interact with the real world via sensors and actuators

● These peripherals can fail asynchronously
Contributions & Outline

● Phoenix Peripheral Recovery System
  1. **Insights** into embedded system recovery
  2. **Procedure** for recovering from peripheral failures
  3. **Mechanisms** implementing this procedure
  4. **Evaluation** on microbenchmarks and applications
Owl

- An embedded run-time system and development toolchain which provides:
  - **Productivity**: Python interpreter, interactive prompt
  - **Hardware access**: two native function interfaces

- Available at [embeddedpython.org](http://embeddedpython.org)
Insights: Embedded Systems

1. External Peripheral State
   - External state must be restored
   - Phoenix logs all peripheral accesses and handles each one individually during recovery

2. Space Constraints
   - Microcontrollers have extremely limited memory
   - Phoenix only logs memory that has been changed

3. Time Constraints
   - Embedded systems are event-driven
   - Phoenix minimizes the latency of recovery
Insights: Peripherals

1. Peripherals affect the external state in four different ways
   - **Stateless**: no state
   - **Ephemeral**: temporary state
   - **Persistent**: state determined by a single write
   - **Historical**: state determined by multiple writes
Insights: Peripherals

2. Peripherals do not operate in isolation
   a. P1 depends on P2 if P2 failing results in P1 not having its intended effect on the external state
      ○ e.g. autonomous car: motor and servo
Insights: Peripherals

3. Not all peripheral accesses can be replayed
   a. Re-executing accesses to peripherals that depended on the failed peripheral is mandatory
   b. Re-executing accesses to other peripherals may be incorrect
      ○ **Rematerialize** = skip during re-execution, restoring the old value instead
4. Restoring persistent state takes extra steps
   A. Put P in a safe state during recovery
   B. Restore P’s last state during re-execution
      - If P is in the redo set, restore:
        what: initial state at point of failed access
        when: before re-execution
      - Otherwise, restore:
        what: final re-materialized state
        when: after re-execution
Recovery Procedure

1. Rollback to the point of failure
   - **Goal**: Restore the internal program state

2. Recovery of the failed peripheral
   - **Goal**: Restore system functionality

3. Redo mode execution
   - **Goal**: Restore the external peripheral state
Example

1 # Run the motor
2 speed = 100
3 motor.run(speed)
4 SD.write("set motor to 100")
5 speed += 100
6
7 # Turn the wheels
8 servo.set_servo(-1)
9 ...
Example

1 # Run the motor
2 speed = 100
3 motor.run(speed)
4 SD.write("set motor to 100")
5 speed += 100

6

7 # Turn the wheels
8 servo.set_servo(-1)

detect motor failure

9 ...
Example

```python
1 # Run the motor
2 speed = 100
3 motor.run(speed)
4 SD.write("set motor to 100")
5 speed += 100

# Turn the wheels
6 detect motor
7 servo.set_servo(-1)
8 ...
9 put motor, servo in safe state
```
Example

```python
1 # Run the motor
2 speed = 100
3 motor.run(speed)
4 SD.write("set motor to 100")
5 speed += 100

7 # Turn the wheels
8 servo.set_servo(-1)
9 ...
```
Example

1 # Run the motor
2 speed = 100
3 motor.run(speed)
4 SD.write("set motor to 100")
5 speed += 100
6
7 # Turn the wheels
8 servo.set_servo(-1)
9 ...

recover motor
Example

1  # Run the motor
2  speed = 100
3  motor.run(speed)
4  SD.write("set motor to 100")
5  speed += 100
6
7  # Turn the wheels
8  servo.set_servo(-1)
9  ...

(no last states)
Example

```python
1    # Run the motor
2    speed = 100
3    motor.run(speed)
4    SD.write("set motor to 100")
5    speed += 100
6
7    # Turn the wheels
8    servo.set_servo(-1)
9    ...
```
Example

1 # Run the motor
2 speed = 100
3 motor.run(speed)
4 rematerialize SD.write("set motor to 100")
5 speed += 100
6
7 # Turn the wheels
8 servo.set_servo(-1)
9 ...

Example

1  # Run the motor
2  speed  =  100
3  motor.run(speed)
4  SD.write("set motor to 100")
5  redo ➔ speed  +=  100
6
7  # Turn the wheels
8  servo.set_servo(-1)
9  ...
19
Example

1 # Run the motor
2 speed = 100
3 motor.run(speed)
4 SD.write("set motor to 100")
5 speed += 100
6
7 # Turn the wheels
8 redo  servo.set_servo(-1)
9 ...
Example

1  # Run the motor
2   speed = 100
3  motor.run(speed)
4  SD.write("set motor to 100")
5  speed += 100
6
7  # Turn the wheels
8   servo.set_servo(-1)
9     ...

exit redo mode
Mechanisms

- **Run-time system:**
  - Enables and disables checkpointing
  - Logs the internal and external state when checkpointing is enabled
  - Detects success and failure of peripheral accesses
  - Executes the recovery procedure

- **Compiler:**
  - Injects code to enable checkpointing
  - Injects code to track outstanding peripheral accesses
Checkpointing Structures

- **Goal:** Log the internal and external state
  - Store multiple simultaneous checkpoints efficiently

- Stored on a second heap to persist past rollback of the Python heap

- Only used when checkpointing is enabled
  - Populated incrementally as state is changed
  - Freed incrementally as accesses are acked
**Journal**

- **Goal:** Log the internal program state

- One entry per store to the Python heap
  - Heap is set read-only by the MPU
  - Faults are handled by journaling the (memory address, old contents) prior to executing the store

- Implemented in software; could be implemented in hardware for efficiency
Rematerialization Queues

- **Goal**: Log the external peripheral state
- One queue per peripheral
- One entry per access, which stores:
  1. **Rollback point**: current index into the journal
  2. **Rematerialization info**: arguments and return value
Control Flow Queue

● **Goal:** Drive redo mode execution

● Logs control flow during normal execution

● One entry per bytecode

● Exit redo mode if:
  1. Control flow diverges from the original path, or
  2. The point of failure detection is reached again
Example

1. `speed = 100`
2. `motor.run(speed)`
3. `speed += 100`
4. `servo.set_servo(-1)`
5. `...`
Example

1  speed = 100
2  motor.run(speed)
3  speed += 100
4  servo.set_servo(-1)
5  ...

Journal:

**First Slot:** -1
**Next Slot:** 0

**Entries:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example

1    speed = 100
2   motor.run(speed)
3   speed += 100
4  servo.set_servo(-1)
5    ...

Rematerialization Queues:

(Motor)  

Journal:

FIRST SLOT: -1
NEXT SLOT: 0
ENTRIES:

0
1    ...

HEAD:  
JNL IDX: 0
ARGS:   
RETVAL:   

( , )  
100

NONE
Example

1. \texttt{speed} = 100
2. \texttt{motor.run(speed)}
3. \texttt{speed += 100}
4. \texttt{servo.set_servo(-1)}
5. ...

Rematerialization Queues:

(Motor)

\begin{itemize}
  \item \texttt{HEAD:}
  \item JNL IDX: 0
  \item ARGS:
  \item RETVAL:
\end{itemize}

Journal:

\begin{tabular}{|c|c|}
  \hline
  \textbf{First Slot:} & 0 \\
  \textbf{Next Slot:} & 1 \\
  \textbf{Entries:} & \begin{tabular}{|c|c|}
    \hline
    0 & \&speed & 100 \\
    1 & ... & ... \\
    \hline
  \end{tabular} \\
  \hline
\end{tabular}
Example

1. `speed = 100`
2. `motor.run(speed)`
3. `speed += 100`
4. `servo.set_servo(-1)`
5. ...

Rematerialization Queues:

(Motor)

<table>
<thead>
<tr>
<th>HEAD</th>
<th>JNL IDX: 0</th>
<th>ARGS</th>
<th>RETVAL</th>
<th>(, )</th>
<th>100</th>
</tr>
</thead>
</table>

(Servo)

<table>
<thead>
<tr>
<th>HEAD</th>
<th>JNL IDX: 1</th>
<th>ARGS</th>
<th>RETVAL</th>
<th>(, )</th>
<th>-1</th>
</tr>
</thead>
</table>

Journal:

First Slot: 0
Next Slot: 1
Entries:

<table>
<thead>
<tr>
<th>Slot</th>
<th>&amp;speed</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example

1. \texttt{speed = 100}
2. \texttt{motor.run(speed)}
3. \texttt{speed += 100}
4. \texttt{servo.set_servo(-1)}

\[ \rightarrow 5 \ldots \]

Rematerialization Queues:

Journal:

\begin{tabular}{|c|c|}
\hline
\textbf{FIRST SLOT}: & -1 \\
\textbf{NEXT SLOT}: & 1 \\
\hline
\textbf{ENTRIES}: & \\
0 & \&speed & 100 \\
1 & … & … \\
\hline
\end{tabular}
Interrupt Handlers

● **Goal:** Detect failure, acknowledge success

● On **success**, decrement the count of outstanding peripheral accesses

● On **failure**, throw an exception to the interpreter requesting rollback
Compile-time Support

● **Goal:** Identify rollback points
  ○ New `JOURNAL_STORE` bytecode enables checkpointing
  ○ Inserted just before loading arguments to peripheral access function calls

● **Goal:** Track outstanding peripheral accesses
  ○ After each access, code is added to increment the number of outstanding accesses
Application Development

- **Goal:** disentangle peripheral recovery code from application-specific code

- Programmer must follow two simple rules:
  1. Define a Python class for each peripheral
  2. Provide a config file including peripheral metadata
Peripheral Class

- **Goal**: Specify peripheral recovery behavior

- Each peripheral extends one of four classes
  - StatelessPeripheral
  - EphemeralPeripheral
  - PersistentPeripheral
  - HistoricalPeripheral

- Programmer defines functions to support:
  - **Access**: the only C code the programmer must write
  - **Recovery & Restoration**: programmer determines how; system determines when
Example

```python
class Motor(PersistentPeripheral):
    def __init__(self):
        # Initialize primary device
        self.init(PRIMARY)

    def recover(self):
        # Switch to backup device
        self.init(BACKUP)

    def safe_state(self):
        # Stop the motor
        self.set_speed(0)

    def last_state(self, *args):
        # For use by Phoenix only
        native_write(*args)
```
Configuration File

- **Goal**: Specify peripheral metadata
  1. Number of interrupts per peripheral access
  2. Dependencies between peripherals

```plaintext
[dependencies]
motor -> servo
servo -> motor
SD ->
```
Evaluation

● Used the Stellaris LM3S9B92 for evaluation
  ○ 96 KB SRAM, 256 KB flash, 50 MHz

● Microbenchmarks:
  ○ Named in the form <peripherals>_<actions>
    ■ <peripherals> ⊂ {gyro, compass}
    ■ <actions> ⊂ {r, w, c} for {read, write, compute}

● Applications:
  ○ Autonomous RC car (motor, servo, gyro)
  ○ Obstacle tracker (display, range finder)
  ○ Virtual compass (display, compass)
Evaluation: Space

Benchmark Recovery Space, With and Without Failure (bytes)

- gyro_compass_wr
- gyro_wr
- gyro_w
- gyro_r
- compass_wcrc
- compass_wr
- compass_w
- compass_r

Size (bytes)
Evaluation: Space

Application Recovery Space, With and Without Failure (bytes)

- virtual compass
- obstacle tracker
- autonomous car

Size (bytes)

- Without Failure
- With Failure
Evaluation: Time

- Overhead of a single failure: 12–143 ms
- Overhead of a journaled store: 6.2 μs
  - Projected 40.2 ns with hardware journal
- No discernible slowdown on ⅔ applications
  - Virtual compass (intensive accesses)
  - Autonomous RC car (periodic accesses)
  - Obstacle tracker (fixed sleep between accesses)
Conclusions

- Hardware peripherals introduce complex failure scenarios
  - External state impacts the real world
  - Failures occur asynchronously

- Phoenix simplifies handling these failures
  - Incremental checkpointing
  - Precise rollback to the source of the failure
  - Correct recovery of both the internal program state and the external peripheral state