Timely Execution on Intermittently Powered Batteryless Sensors

Josiah Hester
Kevin Storer @ UCI
Jacob Sorber @ Clemson

SenSys’17—November 7, 2017
Batteryless Sensing Applications

Infrastructure
- Pipelines
- Bridges
- Roads

Buildings
- Occupancy
- Energy Monitor

Wearables
- Clothing
- Jewelry
- Implants

Wildlife Tracking
- Small animal
- Endangered

Extreme Locales
- Deep Sea
- High Altitude
- Space
Traditional Energy View
RFID-Scale Energy

100ms – 1s

Capacitor Voltage

Energy

Full 80% 60% 40% 20% Empty

RFID-Scale Energy
Intermittent Computing

As small as possible
  • Minimal energy storage (Cap)
  • Harvest energy (RF, Solar)

Run when you can
  • Frequent failures
  • Erratic supply
Research Space

Platform and Hardware
- WISP
- Flicker

Testing and Debugging
- Ekho
- EDB

Language and Runtime
- Dewdrop, Mementos, DINO
- Hibernus, Chain, HarvOS...

+ Mayfly (This Work)
Intermittent Programming

Power failures make coding tough

- limited functionality
- no knowledge of time
- no way to control data

We want sophisticated sensing.

There are challenges…
Intermittent Execution

A typical energy harvesting voltage trace...

1. Energy harvesting
2. Execution of program
3. Power failure
4. RAM and clocks cleared

Charge

Volts

Time (s)
Greenhouse Monitoring

Computation

Solar Energy Harvesting

Humidity and Wetness Sensors

Energy Storage
Intermittent Execution

```c
NV int t,l,w,m

main() {
    while(1)
        t = temp()
        l = light()
        m = moist()
        w = wet(t,l)
        send(w)
        sleep()
}
```

Source
Intermittent Execution

NV int t,l,w,m

main() {
    while(1)
        t = temp()
        l = light()
        m = moist()
        w = wet(t,l)
        send(w)
        sleep()
    <repeat>
}

main() {
    while(1)
        t = temp()
        l = light()
        m = moist()
        w = wet(t,l)
        send(w)
        sleep()
        while(1)
            :
            :
        <repeat>
}

main() while(1)
    t = temp()

Reboot

main() while(1)
    t = temp()
    l = light()

Reboot

main() while(1)
    t = temp()
Checkpoints

Source

```c
NV int t,l,m,w
main() {
    while(1)
    t = temp()
    l = light()
    m = moist()
    w = wet(t,l)
    send(w)
    sleep()
}
```

Intermittent

```c
main()
while(1)
t = temp()
```

```
main()
while(1)
t = temp()
l = light()
m = moist()
w = wet(t,l)
send(w)
sleep()
```

Checkpoints

```c
main()
while(1)
t = temp()
```
State of the Art

Existing techniques handle checkpoints and transactions

- checkpointing saves progress
  Mementos, Hibernus, QuickRecall, HarvOS

- memory consistency
  DINO, Ratchet
Limitations

1. Current programming models ignore the relationship between time and data.

2. The code does not map to actual program behavior.
1. NV int w, t, l, m
2. main() {
3.     while(1)
4.         t = temp()
5.         l = light()
6.         m = moist()
7.         w = wet(t, l, m)
8.         send(w)
9.         sleep(1)
10. }

Source               Execution

w[n]=wet(t,l,m)

~0s

How Long?
Limitations

Source

1. NV int w, t, l, m
2. main() {
3.   while(1)
4.     t = temp()
5.     l = light()
6.     m = moist()
7.     w=wet(t,l,m)
8.     send(w)
9.     sleep(1)
10. }

Execution

How does the runtime manage time?

- w[n]=wet(t,l,m)
- How Long?
- 1 day
- send(w)
Limitations

1. Current programming models ignore the relationship between time and data.

2. The code does not map to actual program behavior.
Limitations

1. NV int w, t, l, m
2. main() {
3.     while(1)
4.         t = temp()
5.         l = light()
6.         m = moist()
7.         w = wet(t, l, m)
8.         send(w)
9.         sleep(1)
10. }

Source

Execution does not match intent
Limitations

1. NV int w, t, l, m
2. main() {
3.     while(1)
4.         t = temp()
5.         l = light()
6.         m = moist()
7.         w = wet(t,l,m)
8.         send(w)
9.     sleep(1)
10. }

Source

Intermittence forces complexity
Mayfly

1. Coordination language–built on C

2. Relates time and data.

3. Code maps to program behavior.
// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int lht, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge dependencies
light -> wet {expires(10s),}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10, 30m)}
SYNTAX

// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int lht, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge dependencies
light -> wet {expires(10s),}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10, 30m)}
// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int lht, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge dependencies
light -> wet {expires(10s),}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10, 30m)}
// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int lht, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge dependencies
light -> wet {expires(10s),}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10, 30m)}
Definition. The amount of time data can sit on an edge.
Syntax

\[ \text{misd}(\text{time}) \]

**minimum-inter-sample-delay**

**Definition.** The amount of time before more data is useful.
Syntax

collect(quantity)

**Definition.** Gather a set of data coming out of the task.
// Optional global policies
{: scheduling_policy(FINISH_FLOW); :}

// Task definitions
temp() -> (int temperature)
light() -> (int light)
moist() -> (int moisture)
wet (int tmp, int light, int mstr) -> (int wet)
send (int[] leaf_wetness) -> ()

// Data flows
temp -> wet -> send
light -> wet
moist -> wet

// Edge constraints
light -> wet {expires(10s)}
temp -> wet {expires(1m)}
moist -> wet {expires(2m)}
wet -> send {expires(4m), collect(10)}
Scheduling

1. Statically defined task schedule
2. Near zero energy overhead
3. Handles transactions, timekeeping
Scheduling Loop

// Pre: Sort tasks by priority
while(1)
{
    // What's the next thing to do?
t = next_task();
    // Maintain ordered stack
    if(constraints_satisfied())
        execute(t);
}

Approximation is the intent
Compilation

**Step 1:** Parse program and verify syntax. Check for common errors.

**Step 2:** Generate task graphs from code.

**Step 3:** Check for logic errors. Annotate graph(s) with constraints.

**Step 4:** Take policy information and task graphs to generate sensing schedule.

**Step 5:** Compile and link with runtime libraries and user code.

**Step 6:** Install firmware on device.
Implementation

1. Manage timekeeping
2. Manage execution state
3. Manage data on edges
Timekeeping

Capacitor or SRAM as Hourglass

- timekeeping across power failures
- fine grained and reliable

Persistent Clocks for Batteryless Sensing Devices, Josiah Hester et al. ACM TECS, 2016
Memory Model

Memory Map

- Task-local variables
- Edge Data
- Code
- Time
- Current Task ID

SRAM

FRAM

Active Mode (μA)

0 225 450 675 900

SRAM
FRAM
Memory Model

Memory Map

- Task-local variables
- Edge Data
- Code
- Time
- Current Task ID

SRAM

FRAM

Task 1

collect(n)

Task 2

Stack

x
y
z
Memory Model

Memory Map

- Task-local variables
- Edge Data
- Code
- Time
- Current Task ID
- SRAM
- FRAM

User managed hardware consistency

Mayfly managed hardware consistency

Gyro

Task 1

Task 2

collect(n)
Applications

Cold-chain Equipment Monitoring
Logs temperature at a steady rate to monitor environment sensitive equipment, artifacts or people.

Sit-to-stand Recognition
Gathers acceleration readings and roughly classifies stand up and sit down movement.

Time sensitive data.
1. Experimental Setup: WISP, RFID Reader + Custom Timekeeper PCB

2. Comparison against Chain + DINO

3. Experiments: Data Utility, Memory, Overhead, Usability
Data Utility

**Chain Activity Recognition – execution trace**

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Data</th>
<th>Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 20</td>
<td>Expired</td>
<td></td>
</tr>
<tr>
<td>20 - 40</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>40 - 60</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>60 - 80</td>
<td>Expired</td>
<td></td>
</tr>
</tbody>
</table>

**Mayfly Activity Recognition – execution trace**

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Data</th>
<th>Acceleration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 10</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>10 - 20</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td>20 -</td>
<td>Valid</td>
<td></td>
</tr>
</tbody>
</table>

Data discarded by runtime

Mayfly computes on useful data.
Memory Footprint

Non-volatile memory usage (KB) of each app.

- Mayfly
- Chain
- DINO

Non-volatile memory usage (bytes) of each Mayfly app.

Task Data

Low memory footprint.
## Overhead

### Mayfly Runtime Initialization

- **Harvest** → **Power-on** → **RTC Hard Reboot** → **All Rollback** → **RTC Soft Start** → **Task Rollback** → **Get Time**

### Execution Time

<table>
<thead>
<tr>
<th>Init function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-on from brown-out</td>
<td>1.0 ms</td>
</tr>
<tr>
<td>RTC hard reboot</td>
<td>708.5 ms</td>
</tr>
<tr>
<td>RTC soft start</td>
<td>144.6 μs</td>
</tr>
<tr>
<td>Rollback all Data</td>
<td>23.6 μs</td>
</tr>
<tr>
<td>Single task rollback</td>
<td>4.6 μs</td>
</tr>
<tr>
<td>Get time (seconds, minutes, hours)</td>
<td>246.4 μs</td>
</tr>
<tr>
<td>Get time (seconds)</td>
<td>80.6 μs</td>
</tr>
</tbody>
</table>

### Scheduler function

- Process constraints for single task: 4.5 μs
- Scheduler can’t find task to execute: 56.3 μs
- Task finished, commit results: 7.0 μs

*Needs improvement*
User Study

1. Reduce program writing time?

2. Increase understanding of intermittence?

3. Will programmers use Mayfly?

11 participants, 2.5hrs, 3 coding challenges
Future Work

1. Triggered Tasks

2. Dynamic Scheduling

3. Lightweight Memory Versioning
Mayfly

1. Coordination language—built on C

2. Relates time and data.

3. Code maps to program behavior.

https://github.com/PERSISTLab/
Timely Execution on Intermittently Powered Batteryless Sensors

Josiah Hester
Kevin Storer @ UCI
Jacob Sorber @ Clemson

https://github.com/PERSISTLab/