Peripheral State Persistence
for
Transiently Powered Systems

Tristan Delizy, Kevin Marquet, Guillaume Salagnac

LSC seminar, nov 2016
Context: the Internet of Things

IoT = Embedded Systems + Networking

<table>
<thead>
<tr>
<th></th>
<th>count / user</th>
<th>battery life</th>
</tr>
</thead>
<tbody>
<tr>
<td>wired</td>
<td>+</td>
<td>∞</td>
</tr>
<tr>
<td>wearable</td>
<td>+</td>
<td>days</td>
</tr>
<tr>
<td>set-and-forget</td>
<td>+</td>
<td>years</td>
</tr>
<tr>
<td>permanent</td>
<td>+ + + + + + +</td>
<td>decades</td>
</tr>
<tr>
<td>batteryless</td>
<td>+ + + + + + +</td>
<td>∞</td>
</tr>
</tbody>
</table>

Outline

Introduction

Related Work
- Transiently Powered Systems
- Non-Volatile Random Access Memory
- Intermittent Execution

Peripheral State Persistence
- Peripheral State Volatility Problem
- Peripheral Access Atomicity Problem

Experimental Results

Conclusion and Perspectives
Problem statement: how to run code despite constant reboots?
Non-Volatile Random Access Memory

**NVRAM** aka Storage-Class Memory (SCM)
- retains data when not powered
- byte-addressable
- low latency / low power (vs Flash)

Many **emerging** technologies
- FeRAM, RRAM, MRAM, STT-RAM, CBRAM, PCM...
but no **Universal Memory** in sight (yet ?)
- problems: endurance, write latency / energy, bit errors...

- typical architecture = SRAM+NVRAM
The Broken Time Machine problem

Source program

```c
NONVOLATILE int len = -1;
NONVOLATILE char buf[MAX];

void main(void){
    for(;;){
        append('a');
        ...
    }
}

void append(char c){
    register int reg = len;
    reg = reg + 1;
    len = reg;
    buf[len] = c;
}
```

Dynamic execution

```c
main()
    append('a')
    reg = len;
    reg = reg + 1;
    len = reg;
    buf[len] = 'a';

main()
    append('a')
    reg = len;

main()
    append('a')
    reg = len;
    reg = reg + 1;
    len = reg;
    buf[len] = c;
...```

Lucia & Ransford. **Nonvolatile Memory is a Broken Time Machine.** In *MSPC 2014: ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2014
The Broken Time Machine problem

Source program

```c
NONVOLATILE int len = -1;
NONVOLATILE char buf[MAX];

void main(void){
    for(;;){
        append('a');
        ...
    }
}

void append(char c){
    register int reg = len;
    reg = reg + 1;
    len = reg;
    buf[len] = c;
}
```

Dynamic execution

```c
main()
    append('a')
    reg = len;
    reg = reg + 1;
    len = reg;
    buf[len] = 'a';
    power failure + reboot

main()
    append('a')
    reg = len;
    power failure + reboot

main()
    append('a')
    reg = len;
    reg = reg + 1;
    len = reg;
    power failure + reboot

...```

Lucia & Ransford. **Nonvolatile Memory is a Broken Time Machine.** In *MSPC 2014: ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2014
Classical solution: program checkpointing

Principles:
- Keep execution volatile
- Anticipate power failures
- Save state to non-volatile memory
- Restore state at next boot
Program checkpointing for TPS

[Ransford et al '13]

CPUTRARM NOR Flash

[Jayakumar et al '14]

CPU FeRAM

[Lucia & Ransford '15]

CPU

[F]RAM

[Lucia & Ransford '15]

CPU FeRAM

[NAND Flash]

[Bhatti & Mottola '16]

CPU RAM

FC

[9/27]
Outline

Introduction

Related Work
  Transiently Powered Systems
  Non-Volatile Random Access Memory
  Intermittent Execution

Peripheral State Persistence
  Peripheral State Volatility Problem
  Peripheral Access Atomicity Problem

Experimental Results

Conclusion and Perspectives
Making peripherals persistent too?

Program checkpointing is not enough:

- indirect access
- hardware initialization procedures
- atomicity of each access
The Peripheral State Volatility Problem

Source program

```c
void main(void){
    sensor_init();
    radio_init(myconfig);

    for(;;){
        v = sensor_read();
        radio_send(v);
        ...
    }
}
```

Problem: reloading memory will not restore device state
Our approach (some refactoring required)

In each driver:

- add a `restore()` method
  - similar to `init()`
  - + maybe a switch/case
  - makes use of already existing functions

- maintain a **device context**
  - describing some “restore()-able” state
  - will be included in checkpoint image
When should device contexts be saved to NVRAM?
Capturing state of nested drivers

sys_funA() → drv_funA() → drv_funB() → signal(B) → signal(A) → commit

mark A as dirty
mark B as dirty

commit A&B device contexts to NVRAM
The Peripheral Access Atomicity Problem

Source program

```c
void main(void){
    sensor_init();
    radio_init(myconfig);

    for(;;){
        v = sensor_read();
        radio_send(v);
        ...
    }
}
```

**Problem:** resuming execution in the middle of a hardware access does not always make sense.
Our approach: make "syscalls" atomic

On syscall entry:
- backup arguments + syscall id
- switch to auxiliary volatile stack

On syscall exit:
- clear arguments + syscall id
- commit device contexts
- switch back to main stack

▶ Interrupted syscalls get retried and not just resumed.
Outline

Introduction

Related Work
  Transiently Powered Systems
  Non-Volatile Random Access Memory
  Intermittent Execution

Peripheral State Persistence
  Peripheral State Volatility Problem
  Peripheral Access Atomicity Problem

Experimental Results

Conclusion and Perspectives
Prototype Implementation

- **MSP430RF5739**: 16-bit CPU 24MHz, 1kB SRAM, 15kB FRAM 8MHz
- **CC2500**: 2.4 GHz transciever, 64B packets
Evaluation

Benchmark programs
- RSA encryption
- Diode counter
- Sense and aggregate
- Sense and send

Evaluation metrics
- Duration of shortest usable lifecycle
- Execution overhead
Results: RSA Encryption

- 2.79 ms
- 2%
Results: Sense and Send

Yield vs. T_on (ms)

9.40 ms
Results: detail of the boot sequence

- Hardware boot: 1.24 ms
- App state restoration: 45 µs
- Device context restoration: 27 µs
- Peripheral state restoration: 1.17 ms
- Next checkpoint initialization: 30 µs

Diagram showing the boot sequence with timelines for each step.
## Results: detail of syscall overhead

<table>
<thead>
<tr>
<th>Function</th>
<th>Overhead</th>
<th>Overhead Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>led toggle</td>
<td>3.4 µs, 1.6 µs, 17.8 µs</td>
<td>x 14.25</td>
</tr>
<tr>
<td>ADC read</td>
<td>2.4 µs, 75.2 µs, 17.6 µs</td>
<td>x 1.27</td>
</tr>
<tr>
<td>radio sleep</td>
<td>2.4 µs, 23 µs, 29.2 µs</td>
<td>x 2.37</td>
</tr>
<tr>
<td>radio wake up</td>
<td>2.4 µs, 428 µs, 31.4 µs</td>
<td>x 1.08</td>
</tr>
<tr>
<td>radio send</td>
<td>2.4 µs, 3.4 ms, 20.6 µs</td>
<td>x 1.01</td>
</tr>
</tbody>
</table>
Outline

Introduction

Related Work
  Transiently Powered Systems
  Non-Volatile Random Access Memory
  Intermittent Execution

Peripheral State Persistence
  Peripheral State Volatility Problem
  Peripheral Access Atomicity Problem

Experimental Results

Conclusion and Perspectives
Conclusion and Perspectives

**Transiently Powered Systems with NVRAM**
- Broken Time Machine problem ➤ use checkpointing
- but checkpointing doesn’t work for peripherals

**Peripheral State Persistence**
(with some help from the driver)
- **Volutility**: device contexts + `restore()` methods
- **Atomicity** of “syscalls”: retry VS resume

**Future Work**
- programming abstractions for transient power ?
  - passing of time ; networking ; interrupts
- look at other combinations of hypotheses
  - e.g. battery+NVRAM
  - e.g. managed runtime
Merci de votre attention

Questions ?
Backup Slides

**Embedded Systems**
- ST23ZL48 microcontroller

**Energy Harvesting**
- Typical power ranges of ambient sources

**Non-Volatile Memory**
- Comparison of NVRAM technologies
The ST23ZL48 product is a serial access microcontroller specially designed for secure smartcard applications.

It is based on an enhanced STMicroelectronics 8/16-bit CPU core offering 16 Mbytes of linear addressing space. It is manufactured using an advanced highly reliable ST CMOS EEPROM technology.

Moreover, an ISO 7816-3 EMV-compliant asynchronous receiver transmitter (IART) communication peripheral is available.

**Features:**
- 16-bits CPU (27MHz)
- 8kB RAM
- 300kB ROM
- 48kB EEPROM

Typical power ranges of ambient sources

Comparison of NVRAM technologies

<table>
<thead>
<tr>
<th></th>
<th>SRAM</th>
<th>FLASH</th>
<th>PCM</th>
<th>MRAM</th>
<th>RRAM</th>
<th>FRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maturity</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>–</td>
<td>–/–</td>
<td>++</td>
</tr>
<tr>
<td>Density</td>
<td>–/–</td>
<td>++</td>
<td>++</td>
<td>–</td>
<td>+</td>
<td>n/a</td>
</tr>
<tr>
<td>Scalability</td>
<td>++</td>
<td>+</td>
<td>++</td>
<td>+/–</td>
<td>+</td>
<td>–/–</td>
</tr>
<tr>
<td>Endurance</td>
<td>++</td>
<td>–/–</td>
<td>–</td>
<td>+/–</td>
<td>–/–</td>
<td>++</td>
</tr>
<tr>
<td>Leakage power</td>
<td>–/–</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Read latency</td>
<td>++</td>
<td>–</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Read energy</td>
<td>++</td>
<td>–/–</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>–</td>
</tr>
<tr>
<td>Write latency</td>
<td>++</td>
<td>–/–</td>
<td>–/–</td>
<td>+/–</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Write energy</td>
<td>++</td>
<td>–/–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>