Peripheral State Persistence for Transiently Powered Systems

Gautier Berthou, Tristan Delizy, Kevin Marquet, Tanguy Risset, Guillaume Salagnac

CEA-INRIA NVRAM Workshop

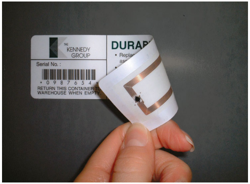
May 30th 2017

Context: *Transiently Powered Systems*

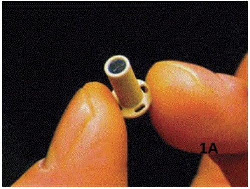
Internet of Tiny Things

- Internet of Things ▶ networked embedded systems
- no battery ▶ must harvest power from the environment

smart cards



RFID tags



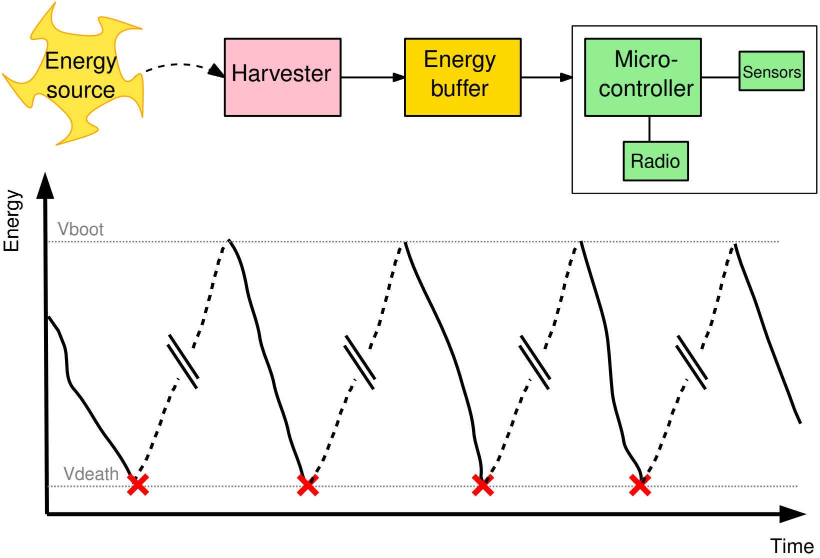
wearable sensors

Today: ad hoc HW+SW codesign ▶ high engineering costs

Expectation: a user-friendly “Internet of Tiny Things App Store”

▶ wearable computing, home automation, environment monitoring, parking assistance, supply chain control...

Transient power = frequent power failures



Problem statement: how to run code despite constant reboots ?

SW baseline: bare-metal application programming

Program=app+libs+drivers

```
ISR deviceA_interrupt()
{ ... }
ISR deviceB_interrupt()
{ ... }

void main(void){
   hardware_init();
   ...
   __enable_interrupts();
   for(;;){
      task1_step();
      tast2_step();
      ...
      __low_power_mode();
   }
}
```

Very little RAM space

- only one stack
- no multithreading

Must boot quickly

- code executes in-place
- app+libs+drivers intermixed

▶ Typical software architecture

- no OS support
- giant loop + interrupt routines

Application must run to completion within one “power window”

Not a solution: Non-Volatile Random Access Memory

NVRAM aka Storage-Class Memory (SCM)

- retains data when not powered
- byte-addressable
- low latency / low power (vs Flash)

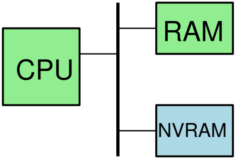
Many emerging technologies

- 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

```
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

```
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;

...
```

Lucia & Ransford. *Nonvolatile Memory is a Broken Time Machine.* MSPC 2014

The Broken Time Machine problem

Source program

```
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;
}
```

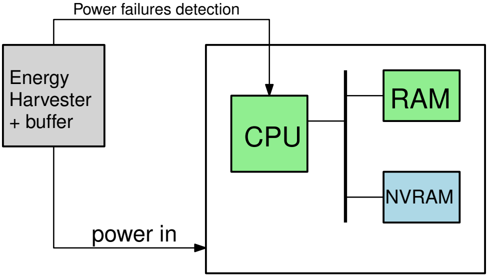
Dynamic execution

```
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

...

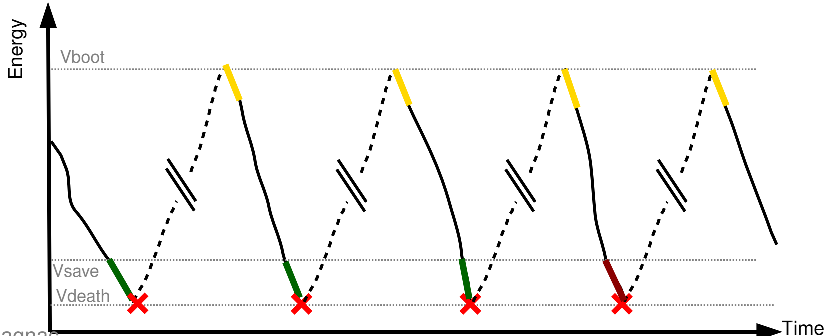
Lucia & Ransford. *Nonvolatile Memory is a Broken Time Machine.* MSPC 2014

State of the art: program checkpointing



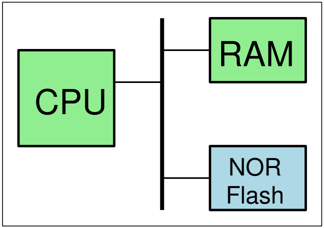
Idea: add some “OS” code to

- Anticipate power failures
- Save program state to a non-volatile memory
- Restore state at next boot

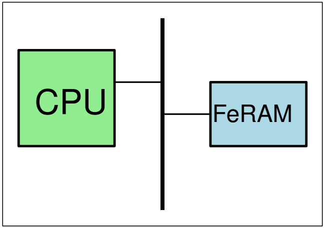


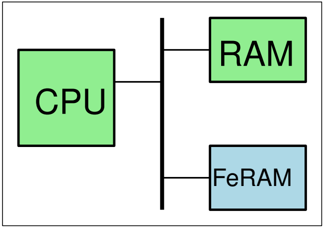
Checkpointing for Transiently Powered Systems

[Ransford *et al* '13]

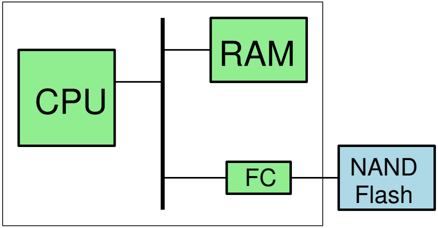


[Jayakumar *et al* '14]
[Lucia & Ransford '15]





[Ait Aoudia *et al* '14]



[Bhatti & Mottola '16]

Outline

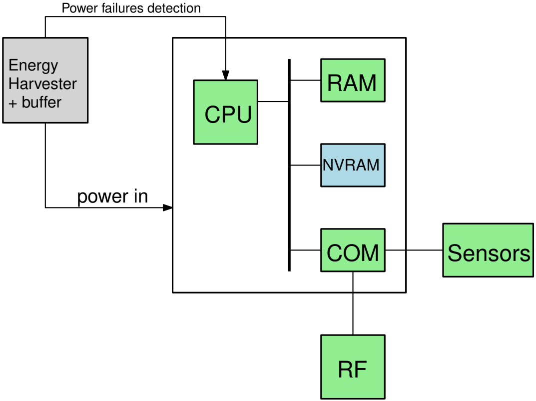
Introduction: Context and State of the Art
- Transiently Powered Systems
- Non-Volatile Random Access Memory
- Intermittent Execution

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:

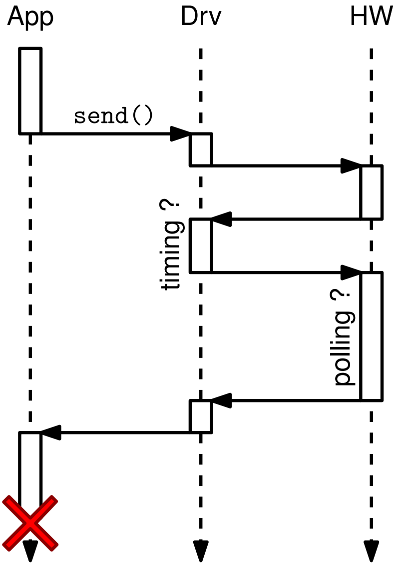
- indirect access
- hardware initialization procedures
- atomicity of each access

The Peripheral State Volatility Problem

Application code

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

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

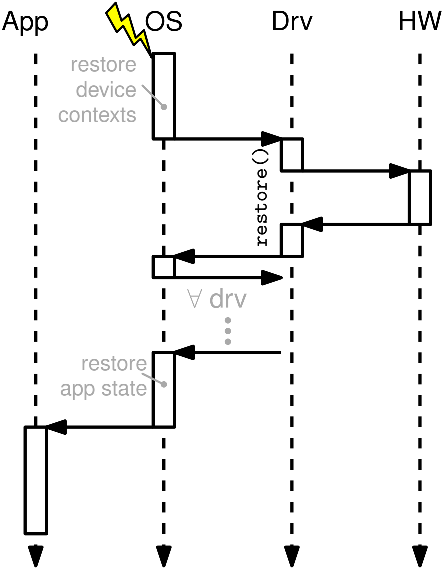


Problem 1: reloading memory will not restore device state

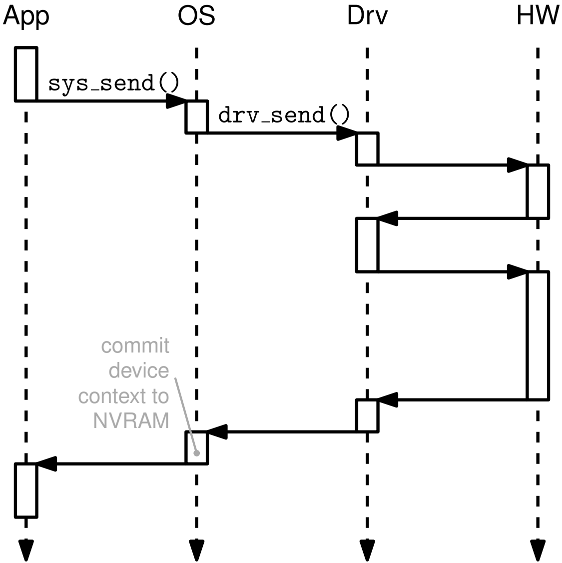
Our approach (some refactoring required)

In each driver:

- add a `restore()` method
  - similar to `init()` + maybe a switch/case
  - makes use of already existing functions
- maintain a device context
  - struct describing a "restore()-able" state
  - will be included in checkpoint image



When should device contexts be saved to NVRAM ?

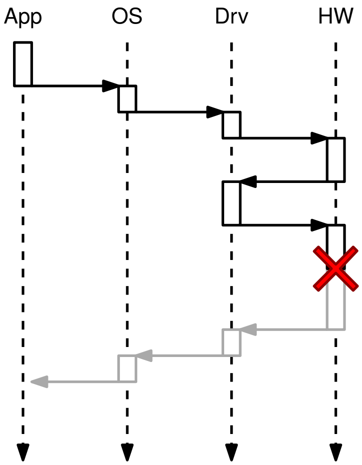


The Peripheral Access Atomicity Problem

Application code

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

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



Problem 2: resuming execution in the middle of a hardware access does not always make sense

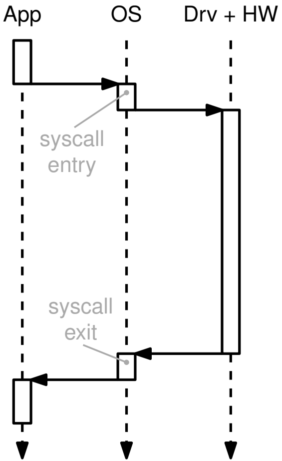
Our approach: make "syscalls" atomic

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

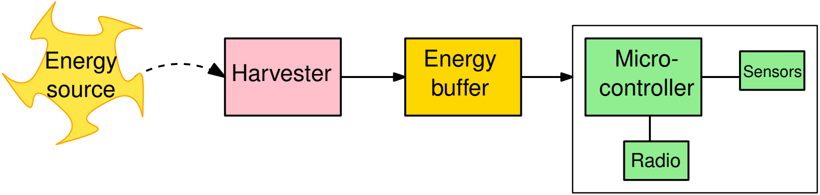
Introduction: Context and State of the Art
- Transiently Powered Systems
- Non-Volatile Random Access Memory
- Intermittent Execution

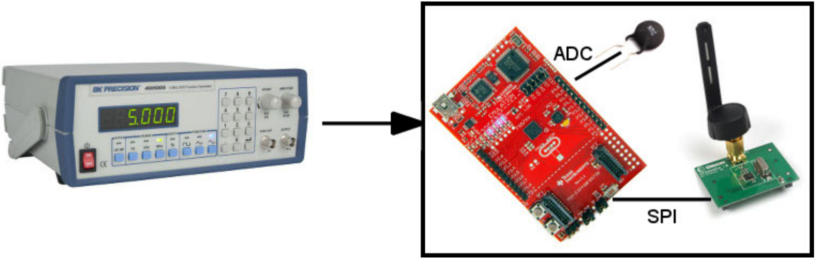
Peripheral State Persistence
- Peripheral State Volatility Problem
- Peripheral Access Atomicity Problem

Experimental Results

Conclusion and Perspectives

Sytare 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

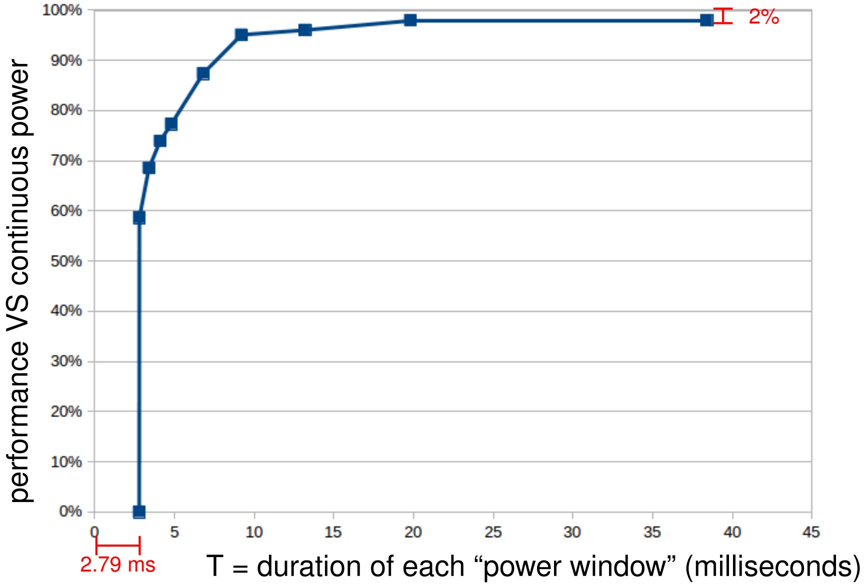
Experimental setup

- Power cycle: ON for a duration T, then OFF (and then repeat)
- Measure performance for various values of T

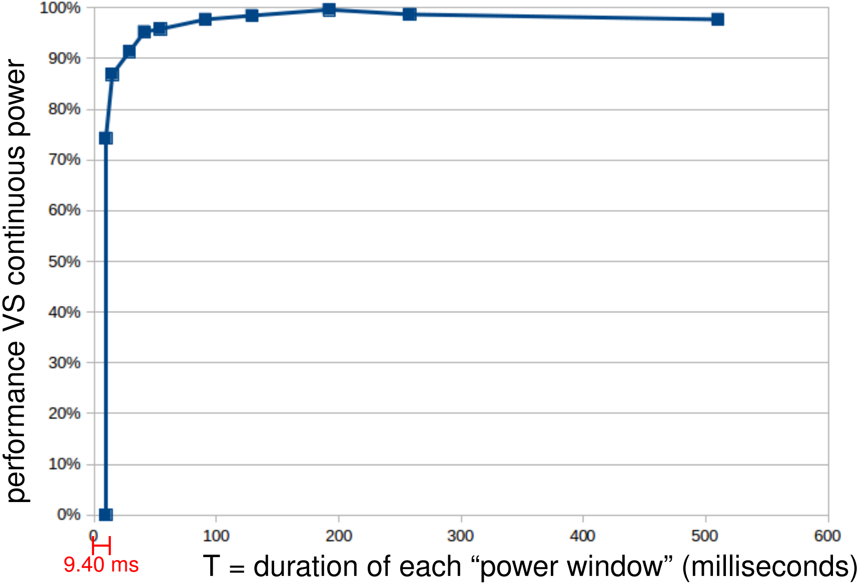
Performance metrics

- Duration of shortest usable power window
- Execution overhead w.r.t. bare-metal baseline

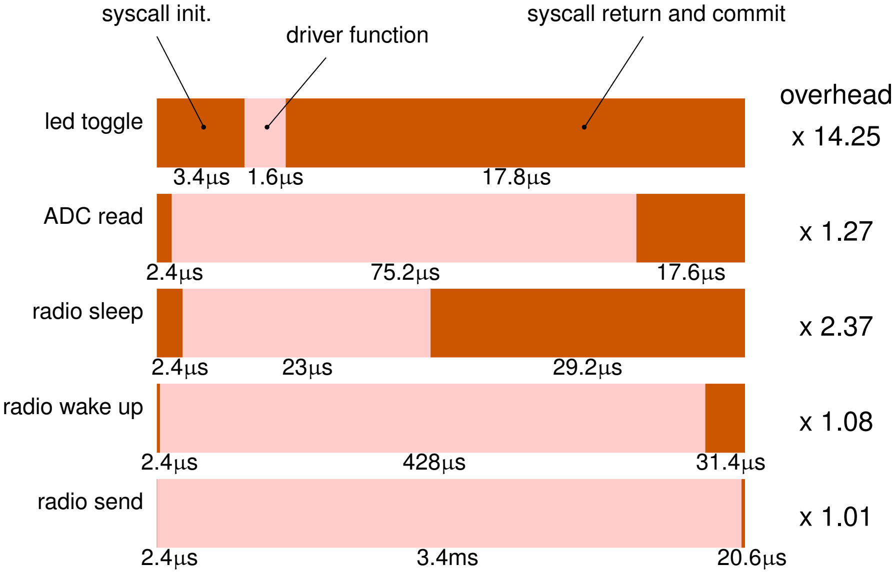
Results: RSA Encryption



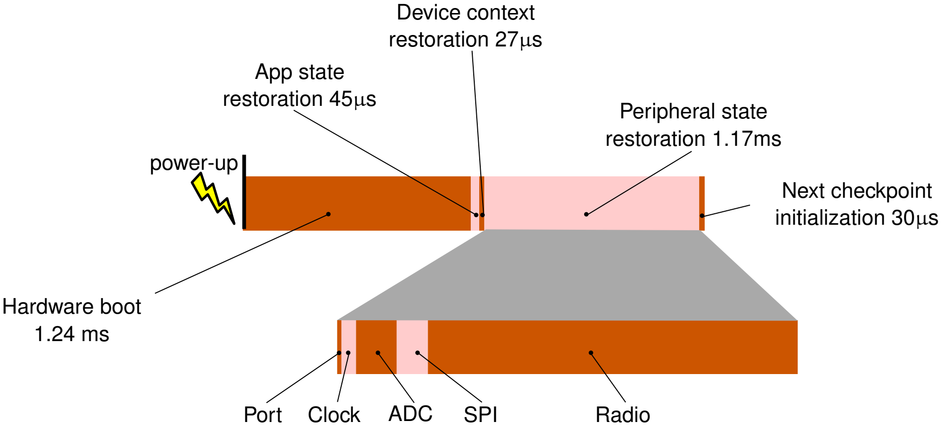
Results: WSN Sense and Send



Results: detail of syscall overhead



Results: detail of the boot sequence



Outline

- Introduction: Context and State of the Art
  - 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

Peripheral State Persistence for Transiently Powered Systems

- Volatility: device contexts + `restore()` methods
- Atomicity of "syscalls": retry VS resume

Perspectives

- programming abstractions for transient power
  - interrupt processing
  - passing of time, delay-tolerance
  - networking stacks and protocols
- other combination of hypotheses
  - NVRAM+battery aka normally-off
  - multiple process management
  - managed runtime