Mementos

System Support for Long-Running Computations on RFID-Scale Devices

Benjamin Ransford*, UMass Amherst
Jacob Sorber, Dartmouth College
Kevin Fu, UMass Amherst

http://spqr.cs.umass.edu/mementos

ASPLOS XVI — March 8, 2011
Ubiquitous Computing

“... the most powerful things are those that are effectively invisible in use.”

— Mark Weiser
(PARC, 1988)

Problem:
Batteryless invisible computer ⇒ constant reboots
Baby Steps Toward Ubicomp

1. Take Emerging Platform
Baby Steps Toward Ubicomp

1. Take Emerging Platform

2. Add Robustness Mechanism
Baby Steps Toward Ubicomp

1. Take Emerging Platform
2. Add Robustness Mechanism
3. Provide Simulation Tools
RFID-Scale Devices

1. Emerging Platform

Radio (RF) harvester

Energy buffer (capacitor)

Reprogrammable microcontroller (~1 MHz) w/ on-chip flash

magnified 10x
RFID-Scale Devices

1. Emerging Platform

- Radio (RF) harvester
- Energy buffer (capacitor)
- Reprogrammable microcontroller (~1 MHz) w/ on-chip flash

Moo WISP: Hong Zhang
RFID-Scale Devices

1. Emerging Platform

Radio (RF) harvester

Energy buffer (capacitor)

Reprogrammable microcontroller (~1 MHz) w/ on-chip flash

Fills quickly, low capacity

Moo WISP: Hong Zhang
RFID-Scale Devices

1. Emerging Platform

Radio (RF) harvester

Energy buffer (capacitor)

Reprogrammable microcontroller (~1 MHz) w/ on-chip flash

Frequent reboots

Fills quickly, low capacity
Mementos: System Support for Long-Running Computation on RFID-Scale Devices

Benjamin Ransford
Department of Computer Science
University of Massachusetts Amherst
ransford@cs.umass.edu

Jacob Serber
Institute for Security, Technology, and Society
Dartmouth College
jacob.serber@dartmouth.edu

Kevin Fe
Department of Computer Science
University of Massachusetts Amherst
kfe@cs.umass.edu

Abstract

Traditionally powered computing devices such as RFID tags, kinetic energy harvesters, and smart cards typically rely on programs that complete a task under tight time constraints before energy starvation leads to complete loss of volatile memory. Mementos is a software system that transforms general-purpose programs into interruptible computations that are run from frequent power losses by automatic, energy-aware state checkpointing. Mementos compiles a collection of optimization passes for the LLVM compiler that transform programs to enable energy-efficient execution. It maps components of a program to execute within the energy budget of the device. It computes an integer state checkpointing interval and uses this interval to schedule checkpoints. It restarts the program from the last checkpoint when power is restored. It uses an energy-aware system call interface for energy measurement while managing state checkpoints stored for power loss recovery. It supports automatic and manual reinitialization of software and data in a trace-driven simulator of an energy-aware state checkpointing system for those devices that are implemented for the MFC30 family of microcontrollers, and a trace-driven simulator of an energy-aware state checkpointing system for those devices that are implemented for the MFC90 family of microcontrollers.

Categories and Subject Descriptors: C.3 [Special-purpose and Application-based Systems]: Real-time and embedded systems

General Terms: Design, Experimentation

Keywords: Mementos, RFID-Scale Devices, Computational RFID, Energy-aware Checkpointing

1. Introduction

Demand for low-power, easily deployable computers has driven the development of general-purpose microcontroller-powered devices that lack both batteries and wired power, operating exclusively on energy harvested from remote supplies or the environment. Such devices range from computational RFID [9] to microcontroller-based devices that harvest RF from readers and communicate via RFID protocols to general-purpose batteryless sensor devices [5].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM Reference Format


Computing under transient power conditions is a challenge. Traditionally powered RFID tags use simple state machines instead of supporting general-purpose computation. Contactless smart cards perform more complicated special-purpose computations (e.g., cardholder authentication), however, they offer no execution guarantees, and instead rely on the user to provide the needed RF power for a sufficient period of time. When energy consumption outpaces energy harvesting, these computations fail and must restart from scratch, when adequate energy becomes available.

With the low-power microcontrollers (MFC30, MFC90) programs deployable devices can perform computation and sensing under RFID-scale energy constraints; however, these MFCs consume more power than conventional RFID circuits, and energy consumption can easily surpass harvesting, resulting in frequent power loss. Today, programs that run CPU-intensive operations like cryptography or time-sensitive applications are power constrained and painstakingly hard-coded to complete within a short time window (often under 100 ms) [5, 9]. The resilience and power of RFID-scale devices can be dramatically improved if designers can confidently write programs without being limited by power failures.

Mementos is a software system that enables long-running computations to cope with low-power events by combining complete-time instrumentation and time-aware energy-aware state checkpointing. At compile time, Mementos installs function calls that estimate available energy. At run time, Mementos monitors power losses and, when appropriate, saves programs in nonvolatile memory. After a failure, program state is restored and execution continues rather than restarting from scratch.

This paper makes the following contributions: (1) An energy-aware state checkpointing system that sparsely programs execution when the remaining time is not sufficient for execution. The system is implemented for the MFC90 family of microcontrollers, requires no hardware modifications to existing devices, and operates automatically at run time without user intervention. (2) A suite of compile-time optimization passes that inserts energy-checkpoints into programs. The optimization passes employ three different instrumentation strategies that factor different program structures. (3) A trace-driven simulator to evaluate the behavior of programs on transiently powered RFID-scale devices. The simulator, modeled after a prototype hardware device with an off-the-shelf microcontroller, takes energy- and state-loss events as input and simulates power loss events during runs. We evaluate the simulator’s accuracy and Mementos’s performance under simulation in Section 5.

In the side this discovers it’s orientation, it would be expected for them to have an effect on the system’s perception of the world. However, simple changes in lighting or distance can be misleading. Ransford’s paper at ASPLOS XVI...
Robustness Under RF Harvesting

- Typical approach: constrain the problem
- **Mementos**: relax constraints to make general-purpose computation feasible
Unpredictable Energy Morass

Voltage

Infinite energy; constant voltage

Time
Unpredictable Energy Morass

Infinite energy; constant voltage

Voltage vs. Time

Voltage

(40 seconds)

Time
Mementos Approach

- Checkpoint when failure appears imminent
- Spread computation across reboots
Mementos Approach

- Checkpoint when failure appears imminent
- Spread computation across reboots
Mementos Approach

- Checkpoint when failure appears imminent
- Spread computation across reboots
Mementos Approach

• Checkpoint when failure appears imminent
• Spread computation across reboots
Mementos Approach

- Checkpoint when failure appears imminent
- Spread computation across reboots
Running Example: CRC

- Compute CRC16-CCITT checksum over 2 KB data
- Tight nested loops
- 575,000 CPU cycles ~ 575 ms
Running Example: CRC

- Compute CRC16-CCITT checksum over 2 KB data
- Tight nested loops
- Reboots every $O(100)$ ms!
- 575,000 CPU cycles $\sim 575$ ms
How to Use Mementos

Programmer | Mementos (our contributions)

- Write C code
- Choose params
How to Use Mementos

**Programmer**
- Write C code
- Choose params

**Mementos (our contributions)**
- Instrument w/ energy checks (via LLVM passes)
- Simulate program
How to Use Mementos

Programmer

- Write C code
- Choose params

Mementos (our contributions)

- Instrument w/ energy checks (via LLVM passes)
- Simulate program
- Suggest params
How to Use Mementos

Programmer

Write C code

Choose params

Mementos (our contributions)

Instrument w/ energy checks (via LLVM passes)

Simulate program

Suggest params
Choosing Parameters (1/2)

1) Instrumentation strategy

Programmer

Write C code

Choose params
Choosing Parameters (1/2)

1) Instrumentation strategy

Programmer

Write C code

Choose params

```
unsigned short crc16_ccitt(volatile unsigned char *data, unsigned short crc_16)
{
    register unsigned short i, j;
    unsigned short crc_16;

    crc_16 = 0xFFFFu; // Equivalent Preset to 0x1D0F
    for (i=0; i<n; i++) {
        crc_16 ^= data[i] << 8;
        for (j = 0; j < 8; ++j) {
            if (crc_16 & 0x8000) {
                crc_16 <<= 1;
                crc_16 ^= 0x1021; // (CCITT) x16 + x12 + x5 + 1
            } else {
                crc_16 <<= 1;
            }
        }
    }

    return(crc_16^0xFFFFu);
}
```

```
crc-vanilla.c (/opt/mementos/src/mementos/samples) - Vim
```

```
/home/tools/xx/2016/0001
```

```
/home/tools/xx/2016/0001
```
Choosing Parameters (1/2)

Programmer

Write C code

Choose params

1) Instrumentation strategy

```
unsigned short crc16_ccitt(volatile unsigned char *data, unsigned short *crc16)
{
    register unsigned short i, j;
    unsigned short crc_16 = 0xFFFFU; // Equivalent Preset to 0x1D0F
    for (i=0; i<n; i++) {
        crc_16 ^= data[i] << 8;
        for (j = 0; j < 8; ++j) {
            if (crc_16 & 0x8000) {
                crc_16 <<= 1;
                crc_16 ^= 0x1021; // (CCITT) x16 + x12 + x5 + 1
            } else {
                crc_16 <<= 1;
            }
        }
    }
    return(crc_16^0xFFFFU);
}
```

Checkpoint?
Choosing Parameters (1/2)

1) Instrumentation strategy

Programmer

Write C code

Choose params

```
unsigned short crc16_ccitt(volatile unsigned char *data, unsigned short crc_16)
{
    register unsigned short i, j;
    unsigned short crc_16;

    crc_16 = 0xFFFFu; // Equivalent Preset to 0x1D0F
    for (i=0; i<n; i++) {
        crc_16 ^= data[i] << 8;
        for (j = 0; j < 8; ++j) {
            if (crc_16 & 0x8000) {
                crc_16 <<= 1;
                crc_16 ^= 0x1021; // (CCITT) x16 + x12 + x5 + 1
            } else {
                crc_16 <<= 1;
            }
        }
    }
    return(crc_16^0xFFFFu);
}
```

Checkpoint? Checkpoint?
Choosing Parameters (2/2)

2) Checkpoint threshold $V_{\text{thresh}}$

Programmer

- Write C code

Choose params
Choosing Parameters (2/2)

2) Checkpoint threshold $V_{\text{thresh}}$

Programmer

Choose params

Write C code

DEATH

Voltage (V)

Time (ms)
Choosing Parameters (2/2)

2) Checkpoint threshold $V_{\text{thresh}}$

Programmer

- Write C code
- Choose params

Diagram:
- Time (ms) on x-axis
- Voltage (V) on y-axis
- DEATH at 3.6 V
- CKPT at 2.0 V
Choosing Parameters (2/2)

2) Checkpoint threshold $V_{\text{thresh}}$

Programmer

- Write C code
- Choose params

```
Write C code
Choose params
```

```
DEATH
```

```
Wasted!
```
Choosing Parameters (2/2)

2) Checkpoint threshold $V_{\text{thresh}}$

Programmer

Write C code

Choose params

![Diagram showing Voltage (V) over Time (ms) with DEATH and CKPT markers.](image-url)
Choosing Parameters (2/2)

2) Checkpoint threshold $V_{\text{thresh}}$

Programmer

- Write C code
- Choose params

![Graph showing voltage over time with DEATH and CKPT markers.](image-url)
Choosing Parameters (2/2)

2) Checkpoint threshold $V_{\text{thresh}}$

Programmer

Write C code

Choose params

Voltage (V)

Time (ms)

DEATH

CKPT
Choosing Parameters (2/2)

2) Checkpoint threshold $V_{\text{thresh}}$

Programmer

Write C code

Choose params

![Graph showing voltage over time with threshold and death points]
Assorted Challenges

Checkpointing isn’t trivial in this context

- No FTL; manage flash ourselves
- Can’t overwrite arbitrary bit patterns in flash memory ➔ tricky checkpoint maintenance

Working on these devices is painful

- Fickle harvesting ➔ runs not reproducible
- Limited visibility into running hardware
Trace-Driven Simulator

- Based on MSPsim — cycle-accurate, open-source MSP430 simulator [EWSN '07]
- We augmented MSPsim with notions of energy (harvester, capacitor, power loss)
Trace-Driven Simulator

- Based on MSPsim — cycle-accurate, open-source MSP430 simulator [EWSN ’07]
- We augmented MSPsim with notions of energy (harvester, capacitor, power loss)
Accurate Energy Simulation

• Simulated capacitor obeys capacitor equations to buffer incoming energy

• Validated with microbenchmarks (all chip modes, all instruction classes)

• Measured MSP430 current down to μA

• Details in paper
Straightforward Simulation

- Simulator input: <executable, voltage trace>
- Output: <# reboots to completion, # CPU cycles, total time, execution trace>
Straightforward Simulation

- Simulator input: <executable, voltage trace>
- Output: <# reboots to completion, # CPU cycles, total time, execution trace>
Simulation with an Oracle

- Checkpoint oracle finds last practicable opportunity by binary search on $V_{\text{thresh}}$
  - Uninstrumented code $\rightarrow$ best-case estimate
  - Final report: lower bound for $V_{\text{thresh}}$
Simulation with an Oracle

- Checkpoint oracle finds last practicable opportunity by binary search on $V_{\text{thresh}}$
  - Uninstrumented code $\rightarrow$ best-case estimate
  - Final report: lower bound for $V_{\text{thresh}}$
Simulation with an Oracle

- Checkpoint oracle finds last practicable opportunity by binary search on $V_{\text{thresh}}$
  - Uninstrumented code $\rightarrow$ best-case estimate
  - Final report: \textit{lower bound} for $V_{\text{thresh}}$

![Graph showing voltage over time with checkpoint markers and death indicator]
Evaluation

• **High-level:** Mementos splits execution in simulation and on hardware

• Focus on CRC example test case

• Baselines:
  - Execution without Mementos
  - Execution against checkpoint oracle
Constant Part of Overhead

- Impact on code memory (NVRAM):
  - 2.4 KB for Mementos library
  - 1 KB reserved checkpoint storage

- Impact on run time:
  - ~0.1 ms per energy check (mostly ADC read)
  - CRC (46 bytes): checkpoint 4 ms, restore 2 ms
Constant Part of Overhead

- Impact on code memory (NVRAM):
  - 2.4 KB for Mementos library
  - 1 KB reserved checkpoint storage
- Impact on run time:
  - ~0.1 ms per energy check (mostly ADC read)
  - CRC (46 bytes): checkpoint 4 ms, restore 2 ms

2 ms boot vs. TinyOS ≥100 ms
CRC Test Case

Uninstrumented, unlimited energy:

575,315 cycles
575 ms
CRC Test Case

Uninstrumented, unlimited energy:

575,315 cycles
575 ms

Oracle:
685,608 cycles
~4,000 ms
14 reboots
$V_{\text{thresh}} \geq 2.35 \text{ V}$

Best execution:
761,983 cycles
6,145 ms
16 reboots
$V_{\text{thresh}} = 2.6 \text{ V}$
CRC Test Case

Uninstrumented, unlimited energy:

575,315 cycles
575 ms

Oracle:

685,608 cycles
~4,000 ms
14 reboots

\[ V_{\text{thresh}} \geq 2.35 \text{ V} \]

Best execution:

761,983 cycles
6,145 ms
16 reboots

\[ V_{\text{thresh}} = 2.6 \text{ V} \]
With and Without Mementos

CRC w/o Mementos: never finishes

CRC 😊 w/ Mementos: 16 reboots

Oracle: 14 reboots
Related Work

- RFID-scale devices
  - Mementos workshop paper [HotPower ’08]
  - Dewdrop scheduler for RFID-scale devices [NSDI 2011]
  - WISP [IEEE TIM ’08] and friends

- Checkpointing
  - Sensornet checkpointing [EWSN ’09]
  - Checkpointing for process migration (Condor [ICDCS ’88], Porch [IEEE Micro ’98])
Extensions

• Dynamic or randomized $V_{\text{thresh}}$ adaptation
• NVRAM technology (PCM? FeRAM?)
• Smarter checkpointing (incremental, LVA...)
• Integrate with asynchronous communications on upcoming RFID-scale prototype (June ’11)
Mementos Conclusion

- Energy-aware checkpoints for computation on batteryless RFID-scale devices
- Tools available today; built on LLVM and MSPsim
- Applications: implantable devices, insect-scale tracking, infrastructure monitoring...
Acknowledgements
Your Homework

Get Mementos, simulator, hardware:
http://spqr.cs.umass.edu/mementos

- What should be moved off-chip to save energy?
- Right combo of RAM & NVRAM? Tiny off-chip NVRAM?
- HW/SW interface for detecting an impending failure?
- Conditional branches predicated on available energy?
- Task scheduling when failure is common case?
- Should we write a SuperTinyOS that expects failure?
- Compile-time optimizations for expected failure?
- How to not repeat non-idempotent actions?
Contingency Slides
\( V_{\text{thresh}} \) Subtleties

- Size, duration of \( \text{CKPT} \) are application dependent
- \( V_{\text{thresh}} \) is a conservative estimate
  - More energy might arrive
  - Choose according to risk tolerance
\( V_{\text{thresh}} \) Subtleties

- Size, duration of \( \text{CKPT} \) are application dependent

- \( V_{\text{thresh}} \) is a conservative estimate
  - More energy might arrive
  - Choose according to risk tolerance

Spoiler: We help the programmer choose \( V_{\text{thresh}} \)
Low-Power Modes

- MSP430 has a variety of low-power modes (~1 µA in LPM3/LPM4) that retain RAM
- Sleeping when energy is low is an optimistic strategy
- We don’t know when or whether energy will return — should Mementos guess?
Applications
Thanks for asking about TinyOS

- Condor, Porch, libckpt — all depend on OS-level or out-of-band facilities
- Sensor OSes (e.g., TinyOS) designed to boot *infrequently* and sip from batteries
  - TinyOS boot: $\geq 100$ ms (too slow)
Thanks for asking about TinyOS

- Condor, Porch, libckpt — all depend on OS-level or out-of-band facilities
- Sensor OSes (e.g., TinyOS) designed to boot *infrequently* and sip from batteries
  - TinyOS boot: $\geq 100$ ms (too slow)

"Existing lightweight OSes *still* too heavy"
CRC Example: Overhead

How much CPU overhead for checks?
Consistently high voltage \( (V > V_{\text{thresh}}) \):

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>CPU Cycles</th>
<th>Mementos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninstrumented</td>
<td>575,315</td>
<td>0%</td>
</tr>
<tr>
<td>Loop latches</td>
<td>619,450</td>
<td>6.9%</td>
</tr>
<tr>
<td>Function returns</td>
<td>577,702</td>
<td>0.2%</td>
</tr>
<tr>
<td>Timer + latches</td>
<td>598,171</td>
<td>3.4%</td>
</tr>
</tbody>
</table>
Why Not Just Add...

- Thin-film battery?
- Deeper charge pump (higher voltage)?
- Tiny dedicated NVRAM?
- Hardware “low energy” interrupt support?
Other Use Cases

Programmer can:

- Disable instrumentation at a function level
- Manually call Mementos routines