Programming Challenges of Intermittent Energy-harvesting Devices
Brandon Lucia | Assistant Professor, CMU ECE
ABSTRACT Research Group - https://intermittent.systems
with PhD Students: Alexei Colin, Kiwan Maeng, Emily Ruppel, Vignesh Balaji
Graham Gobieski, Milijana Surbatovich
MS Students: Preeti Murthy, Amanda Marano, Neil Ryan, Devon White, Chris Meredith
BS Students: Graham Harvey, Mark McElwaine, Hannah Tomio
Industry Collaborators: Zac Manchester (Harvard/KickSat)
Alanson Sample (Disney Research Pittsburgh)
Emerging Hardware/Software Devices Everywhere

Wearables, sensors, “Internet of Things”, tiny satellites, medical implants, environmental monitoring, security, computational art, interactive clothing, smart furniture, smart carpets, smart toilet paper

Brandon Lucia - Carnegie Mellon University - Challenges of Intermittent Computing
Energy harvesting untethers devices

- Energy source
- Empty space
- Energy receiver (antenna)
- Energy buffer (capacitor)
- Computer (microcontroller)
...but energy is **intermittent**—and as a result—software becomes **inherently unreliable**.
Designing for Intermittence
System Support for Reliable Intermittence

Debugging Intermittence
Toolchain Support for Understanding Intermittence
Running on intermittent energy
Running on intermittent energy
Running on intermittent energy

Available Energy vs Time

“Death Line”

Not charging, dead.
Running on intermittent energy

Available Energy vs. Time

Charging, dead.

RF Available!
Running on intermittent energy
Available Energy

Time

Computer shuts down

Computer reboots

“Life Line”

“Death Line”
The Intermittent Execution Model

[PLDI '15]
The Intermittent Execution Model

**Goal:** Run programs that take longer than one green box
```c
void main(void){
    for( i = 1 .. 10)
        append()
}

void append(){
    sz++
    buf[sz] = 'a'
}
```
```c
void main(void){
    for( i = 1 .. 10)
        append()
    }

void append(){
    sz++
    buf[sz] = 'a'
}
```
void main(void){
    for( i = 1 .. 10)
        append()
}

void append(){
    sz++
    buf[sz] = 'a'
}

for( i = 1 )
append()
for( i = 2 )
append()
```c
void main(void){
    for( i = 1 .. 10)
        append()
    for( i = 2 )
        append()
    for( i = 3 )
        append()
    sz++
    buf[sz] = 'a'
    sz++
    buf[sz] = 'a'
    sz++
    buf[sz] = 'a'
}

void append(){
    sz++
    buf[sz] = 'a'
}
```
We can model intermittence as a control-flow problem.
Control-flow Graph

append() -> main()

for( i = …)

<loop>

call append()

buf[sz] = 'a'
sz++

end of main()

Failure Induces Implicit Control-flow

Back in time!

Intermittent Programming Challenge:
Implicit control-flow “back in time” on reboot.
Mixture of Volatile & Non-volatile State

volatile memory (e.g., DRAM, SRAM registers)

Access latencies growing similar w/ new technology

non-volatile memory (e.g., Flash, FRAM)
Reboots **clear** volatile state and **preserve** non-volatile state
Prior work (e.g., Mementos) overlooked the need to version non-volatile memory with checkpoints.

Use NV memory to capture values of registers, stack, and global variables.

Control-flow Graph

main()

append()

buf[sz] = 'a'

for (i = …) {
  call append()
}

<loop>

Checkpoint Volatile State

end of main()
Intermittent Programming Challenge:
Implicit control-flow “back in time” on reboot.

Checkpoint introduce more complex implicit control-flow
The Big Idea

The way we think about programming does not match the intermittent execution model.
Challenge: *Intermittence Bugs*

Software that is **correct** with continuous power can be **incorrect** due to intermittent execution especially when we combine non-volatile and volatile memory!

```c
void main(void){
  for( i = 1 .. 10 )
    append();
}

void append(){
  sz++
  buf[sz] = 'a'
}

for( i = 1 )
  append()
for( i = 2 )
  append()
for( i = 3 )
  append()
  sz++
  buf[sz] = 'a'
  sz++
  buf[sz] = 'a'
  sz++
  buf[sz] = 'a'

for( i = 1 )
  append()
  sz++
  buf[sz] = 'a'
for( i = 2 )
  append()
  sz++
  buf[sz] = 'a'
```
Intermittence Bug: Out-of-thin-air Values

[MSPC ’14; PLDI ’15]
**Intermittence Bug: Out-of-thin-air Values**

[MSPC ’14; PLDI ’15]

```
main()
    append()
    for( i = …)
    call append()
<loop>
end of main()
```

```
buf[0] = 'a'
sz++
for( i = 1)
call append()
buf[sz] = 'a'
```

**Intermittence Bug:**

Out-of-thin-air Values

[MSPC ’14; PLDI ’15]
Intermittence Bug: Out-of-thin-air Values

Error: 'a' is appended to buf twice on the i = 1 iteration
Intermittence Bug: Out-of-thin-air Values

[MSPC ’14; PLDI ’15]

Stuck in an implicit loop for $i = 1$
Intermittence Bug: Out-of-thin-air Values

"Out of thin air" value not permitted by any continuous execution!

Application failures can depend solely on the availability of wireless power and capacitor behavior!
Intermittence Bug: Structural Inconsistency

Reboot violates atomicity of
\[ sz++; \text{buf}[sz] = 'a' \]
causing update to wrong entry in \text{buf}.

Corrupt buffer state impossible in a continuous execution!
**Intermittence Bug:** I/O Atomicity & Timeliness

[Colin OOPSLA ’16, Maeng OOPSLA ’17, Hester et al SenSys ‘17]

**Checkpoint between** I/O operations that should execute together leaves I/O inconsistent.

Inconsistent data read at different moments in time

Useless, stale data read a long time ago
**Intermittence Bug:** Livelihood, Non-termination, and Energy-Provisioning

[Colin, et al. CASES ’15, ASPLOS ’16]

**Energy cost** atomic span exceeds energy capacity

Code that is **correct** with continuous power reboots **indefinitely** on intermittent power.

---

For $i = 1$

- Call hicost()
- Call foo()
- Call bar()

If energy cost exceeds full capacitor charge:

*never completes 1 iteration*

---

Affected code need not even modify NV storage!

---

Energy cost atomic span exceeds energy capacity

---

Affected code need not even modify NV storage!
**Intermittence "Bug":** Reactive vs. Burst Operation

Different operations have different energy requirements because of different hardware activation.

If \( \text{temp} > T \)

**Need sampling coverage** for sensing, long **burst** for BLE, **reactive** operation for asynchronous events.
Reliable Intermittent Execution

Intermittent Programming Models
Dino
Arbitrary Task-based Intermittent Programming [PLDI’16]

Chain
Task-based Intermittence w/ Static Data Duplication [OOPSLA’16]

Alpaca
Task-based Intermittence w/ Dynamic Privatization [OOPSLA’17]
Tasks have \textit{atomic} all-or-nothing semantics.

No \textit{implicit} control-flow!

\textit{Task boundaries} selectively \textbf{checkpoint} volatile and \textbf{non-volatile state}

Task-atomic Execution in DINO

\[\text{[PLDI '15]}\]
for( i = 1)
call append()
buf[sz] = 'a'
sz++

for( i = 2)
call append()
sz++
buf[sz] = 'a'

DINO Eliminates OoTA Values
[PLDI '15]
int NV_Array[1000000];
\textbf{task\_boundary}
for( i = 0; i < 1000000; i++){
    NV_Array[i] = i;
}\textbf{task\_boundary}

\textbf{Key Question in DINO:}
What non-volatile data must we checkpoint?
int NV_Array[1000000];

for( i = 0; i < 1000000; i++ ){
    NV_Array[i] = i;
}

task_boundary

Insight: No versioning! “future” write not observed in “past”

Selective Versioning Makes Tasks Idempotent
int NV_Array[1000000];

**task_boundary**

for (i = 0; i < 1000000; i++) {
    NV_Array[i] ++;
}

**task_boundary**

**Insight:** Must version data! Task reads & writes same NV data

Selective Versioning Makes Tasks Idempotent
Ratchet inserts checkpoints
To break WAR dependences

```
for ( i = ... )
    t = NV_Array[i];
    t++;
    <checkpoint>
    NV_Array[i] = t;
```

**What to checkpoint here?**
**Assumptions about memory?**

**Insight:** Write-After-Read dependences violate idempotence

---

**Alternative Scheme: Idempotent Checkpointing**

Ratchet [Hicks 2016]
int NV_Array[1000000];

```c
task_boundary
i = rand(1000000);
NV_Array[i]++;
```

```c
}
task_boundary
```

Fundamental limitation of checkpointing: Checkpoints must keep volatile and possibly updated NV state consistent across reboots.
Checkpointing Overhead

Task Size Limitation
int NV_Array[1000000];

i = rand(1000000);

NV_Array[i]++;

}

Compiler cannot know i in advance! Must version all of NV_Array.

Programming Challenge:
"How much checkpointing overhead will I have?"

breaks the compiler abstraction boundary
Programmer explicitly defines tasks and task-flow graph in code

Channels *statically multiversion data* separating inputs from outputs

Tasks exchange data via abstract *channels of non-volatile memory*

**Chain:** Static data multi-versioning

[Colin, et al OOPSLA ’16]
**Chain: Static data multi-versioning**

[Colin, et al OOPSLA ’16]

 SELF-Channels allow a future task instance to consume updates from past instances

More complex task structures possible to support complex dataflow
origin task Sense()
int s = sensor()
ChOut { S <- s }, CmpAvg
NextTask CmpAvg

Task Alert()
int s = ChIn S, Sense
int cnt = ChIn Cnt, self
report(s, cnt);
ChOut { Cnt <- cnt+1 }, self
NextTask Sense

Task CmpAvg()
int s = ChIn S, Sense
int head = ChIn HEAD, self
for(int i=0; i < 5; i++){
    sum += ChIn a[i], self
}

avg = sum / 5
ChOut { a[head] <- s }, self
head = (head+1)%5
ChOut { HEAD <- head }, self
if( s > avg*2 ){
    ChOut { S <- s }, Alert
    NextTask Alert
}
NextTask Sense

Chain: Static data multi-versioning
[Colin, et al OOPSLA ’16]
**Alpaca: Tasks with Dynamic Privatization**

[Maeng, et al OOPSLA ’17]

**Function-as-task** means no need to checkpoint stack & regs

**Alpaca selectively privatizes task-shared variables**

```java
origin task Sense()
    S = read_sensor()
NextTask CmpAvg
}

task Alert()
    report(S, Cnt);
    Cnt++
NextTask Sense
}

task CmpAvg()
    for(int i=0; i < 5; i++)
        sum += A[i]
    avg = sum / 5
    A[head] = S
    head = (head+1)%5
    if( S > avg*2 )
        NextTask Alert
    NextTask Sense
}

Shared S, Cnt, A[5], head
```
origin task Sense() {
    S = read_sensor()
    NextTask CmpAvg }

task Alert() {
    priv_Cnt = Cnt
    report(S, Cnt);
    priv_Cnt++
    2PCommit(Cnt) && NextTask Sense }

Shared S, Cnt, A[5], head

Alpaca: Tasks with Dynamic Privatization

[Maeng, et al OOPSLA ’17]
**Mayfly: Timely Intermittent Execution**

[Hester, et al SenSys '17]

New idea: Timely Execution requiring stale inputs to expire

Requires intermittence-safe time-keeping (remanance timekeeping)
Mementos \cite{Ransford:2011:IT:1998019.1998021}

Dino

Ratchet \cite{Hicks:2016:AJ:2897765.2897798}

Chain

Alpaca

Mayfly \cite{Hester:2017:359:1584514.2897765}

Intermittence Bug: Out-of-thin-air Values

Intermittence Bug: Structural Inconsistency

Intermittence Bug: IO Atomicity/Timelines

Intermittence Bug: Livelock/Progress

Intermittence Bug: Reactivity/Bursts

Hardware assumptions?
Prototype Implementations

DINO, Chain, Alpaca
Programming Model

Task-aware Compiler

Runtime Library

Task, Channel Definitions

Task Data-flow Analysis
Link to Task Runtime
Channels, Versioning, Privatization

DINO: Checkpoint & Recovery
Chain: Task Management
Alpaca: Privatization/Commit
Energy-harvesting Platform Prototypes

Applications

Activity Recognition (accelerometer+ML)
MIDI Natural User Interface
Cold-chain Equipment Monitor
Multi-granular Sensor Log

Key Result:
Applications suffer errors & failures without our system support for tasks
The diagram shows a bar chart comparing the normalized run time for different algorithms. The x-axis represents the algorithms: CEM, AR, RSA, CF, BF, and BC. The y-axis represents the normalized run time, with lower values indicating better performance. The chart includes data for three categories: Alpaca, Chain, and DINO.
Use Alpaca, Chain and DINO for your research!
http://intermittent.systems | http://github.com/CMUAbstract
https://cmuabstract.github.io/alpaca-landing-page/

Alpaca
A programming language for writing reliable software for intermittent, energy-harvesting computer systems.

Download VM (10.67GB)
VM contains Alpaca and other previous state-of-the-art systems (Chain, DINO)
UserID: reviewer PW: review.

Pull source code from Github
https://github.com/CMUAbstract/alpaca-oopsla2017
Debugging Intermittence
Toolchain Support for Understanding Intermittence
Intermittence Bugs Lead to Application Errors

“Out of thin air” value not permitted by any continuous execution!

sz & buf are corrupted by the reboot.

Reboot… …bug!

Failure manifests on intermittent power

[Colin, et al. CASES '15, ASPLOS '16, IEEE Micro Top Picks '16]
How Do We Debug *Intermittence Bugs*?

Key Need: Correlate *program events* with the device’s *energy state*.

“What was happening when the power failed that led to weird behavior?”
Energy-interference Precludes Conventional Debuggers

[Colin, et al. CASES ’15, ASPLOS ’16, IEEE Micro Top Picks ’16]

The buggy, “out of thin air” value never shows up with continuous power!

sz & buf are consistent.

Debugger provides power, masks intermittence

No assertions
No break points
No tracing
No interactive debugging
No energy tracing
No debug-time I/O
Energy-interference Complicates Instrumentation

Instrumentation changes energy behavior, can mask corruption, & is oblivious to energy state

sz | 10
buf | 'a' 'a' 'a' 'a' 'a' 'a' 'a' 'a' 'a'
LOG | Need to know what you’re looking for to make LOG useful

Tracing code consumes energy & changes behavior

- printf: UART or other I/O + msg handling code
- LED: up to 5x increase in current draw w/ LED
- Logs: eats NV storage + high access cost

“Energy-interference gap”

[Colin, et al. CASES ’15, ASPLOS ’16, IEEE Micro Top Picks ’16]
Corrupt data is device-internal and does not necessarily affect external signals.

**Oscilloscope:** expensive and limited to external signals

- No assertions
- No break points
- No tracing
- No interactive debugging
- No energy tracing
- No debug-time I/O

Existing Energy-interference-free Tools are Inadequate

[Colin, et al. CASES '15, ASPLOS '16, IEEE Micro Top Picks '16]
EDB: An Energy-interference-free Debugging Toolchain

We built energy-interference-free support for debugging and monitoring intermittent, energy-harvesting computers.

Simultaneously monitor & manipulate program state and energy state.
Monitoring Target Energy & Device State

[Colin, et al. CASES '15, ASPLOS '16, IEEE Micro Top Picks '16]

All monitoring is energy-interference-free

Brandon Lucia - Carnegie Mellon University - Challenges of Intermittent Computing
We charge or discharge the device to compensate for expensive debugging tasks.
Case Study: Expensive Consistency Checks Can Kill You
[Colin, et al. CASES ’15, ASPLOS ’16, IEEE Micro Top Picks ’16]

One list element to check

Many list elements to check

without EDB, checks consume energy

Eventually, checks prevent progress.

```
insert_element()
foreach(elem e){
  assert(e.next.prev == e) }
```
Case Study: EDB Enables Expensive Checks

[Colin, et al. CASES ’15, ASPLOS ’16, IEEE Micro Top Picks ’16]

One list element to check

EDB provides energy for the check only

Many list elements to check

Eliminating energy interference enables expensive checks

```
insert_element()
foreach(elem e){
    assert(e.next.prev == e)
}
```
How Energy-interference-free?

Passive monitoring E interference: 
~0.2% of active power (worst case)

Compensating charge error: ~4.3% of total E (in a 47uF cap)

From the circuits up, EDB is energy-interference-free

[Colin, et al. CASES '15, ASPLOS '16, IEEE Micro Top Picks '16]
Integrating EDB with an application board’s design enables beta testing & diagnostics in a realistic target environment.

EDB can be powered to make test framework reliable.

Integrating EDB Support for Deployed Diagnostics

[Colin, et al. CASES ’15, ASPLOS ’16, IEEE Micro Top Picks ’16]
EDB is freely available to researchers!

Use EDB for your research! Email us for hardware!

The Intermittent Execution Model
Reasoning About Intermittently-powered Devices

Designing for Intermittence
System Support for Reliable Intermittence

Intermittence Debugging
Toolchain Support for Understanding Intermittence
Future Challenges

Intermittent Computing Research Directions
Intermittent Computing:
Verification: Is my program correct?
Key challenge: modeling intermittent behavior

Energy Modeling:
How much energy does my code need?
Key challenge: platform energy varies

Distributed Coordination: Can devices cooperate?
Key challenge: communication & dist. state
Solving the System Challenges of Intermittent Energy-harvesting Devices

Brandon Lucia | Assistant Professor, CMU ECE

ABSTRACT Research Group - https://intermittent.systems

with PhD Students: Alexei Colin, Kiwan Maeng, Emily Ruppel, Vignesh Balaji
Graham Gobieski, Milijana Surbatovich

MS Students: Preeti Murthy, Amanda Marano, Neil Ryan, Devon White, Chris Meredith
BS Students: Graham Harvey, Mark McElwaine, Hannah Tomio

Industry Collaborators: Zac Manchester (Harvard/KickSat)
Alanson Sample (Disney Research Pittsburgh)