A Simpler, Safer Programming and Execution Model for Intermittent Systems

Presenters
Atul Shree: 2012CS50282
Bhargav Reddy: 2012CS50301

About the Paper

Authors : Brandon Lucia @ Carnegie Mellon University, USA
Benjamin Ransford @ University of Washington, USA

Conference : ACM SIGPLAN Notices, 2015

Image Source

Energy Harvesting

process by which energy is derived from external sources

(e.g., solar power, thermal energy, wind energy)

captured, and stored for small, wireless autonomous devices, like those used in wearable electronics.

Energy Harvesting technologies

Ampymove : movement

Piezzoelectricity : mechanical stress

Powerwalking : sole power

https://www.theguardian.com/media-network/2015/jun/04/energy-harvesting-future-mobile-charging

Intermittent Computing

Energy Harvesting Devices(EHDs) are computer systems that operate intermittently, only as environmental energy is available.

Key technology for implantable medical devices, IoT applications, and nano-satellites etc..

http://ee.princeton.edu/events/programming-and-system-support-reliable-intermittent-computing

Intermittent Computing

Energy Harvesting Devices(EHDs) are computer systems that operate intermittently, only as environmental energy is available.

Example : RFID tags.

http://ee.princeton.edu/events/programming-and-system-support-reliable-intermittent-computing

Intermittent Computing

An intermittent execution of a program is composed of periods of sequential execution interrupted by reboots.

A key difference between an intermittent execution and a continuous one is that a reboot is not the end of an intermittent execution.

http://ee.princeton.edu/events/programming-and-system-support-reliable-intermittent-computing

Intermittent Computing

Effects of Rebooting :

- All volatile memory is cleared, and control returns to the entry point of main().
- Nonvolatile state retains its contents across reboots.

http://ee.princeton.edu/events/programming-and-system-support-reliable-intermittent-computing

Memory Inconsistancy

- Breaks the assumption of a continuous energy supply prevalent in existing computation paradigms

Memory Inconsistancy

- Breaks the assumption of a continuous energy supply prevalent in existing computation paradigms
- Intermittent systems could lead to failed states not possible in continuous execution.

Reasons:

- The loss of program state in RAM(volatile memory) and registers

Memory Consistancy Issues

Intermittent systems could lead to failed states not possible in continuous execution.

Reasons:

- The loss of program state in RAM(volatile memory) and registers

Solution :

- Store the complete program state in flash.

Solution : Incremental Checkpointing

Optimal checkpointing solution should be able to precisely identify modified RAM locations and only update those in the secondary storage

Incremental Checkpointing

Optimal checkpointing solution should be able to precisely identify modified RAM locations and only update those in the secondary storage

Observation :
program state is only changed by a few, well defined statements in the program, such as assignment, increment, shift operations, and function calls and returns.

Incremental Checkpointing

Optimal checkpointing solution should be able to precisely identify modified RAM locations and only update those in the secondary storage

Observation :
program state is only changed by a few, well defined statements in the program, such as assignment, increment, shift operations, and function calls and returns.

Solution :

- Special functions to insert before making any state-changes

Incremental Checkpointing

Optimal checkpointing solution should be able to precisely identify modified RAM locations and only update those in the secondary storage

Observation :
program state is only changed by a few, well defined statements in the program, such as assignment, increment, shift operations, and function calls and returns.

Solution :

- Special functions to insert before making any state-changes
- Event to Variable Mapping based on static program analysis

Memory Consistancy Issues

Intermittent systems could lead to failed states not possible in continuous execution.

Reasons:

- The loss of program state in RAM(volatile memory) and registers

Memory Consistancy Issues

Intermittent systems could lead to failed states not possible in continuous execution.

Reasons:

- The loss of program state in RAM(volatile memory) and registers
- Intermittent execution could lead to partially executed code and repeated code that result in consistency violations

Memory Consistancy Issues

Source Code

```
NV int len = -1
NV char buf[]="";
main () {
append();
}
append () {
r1 = len;
r1 = r1 + 1;
len = r1;
buf[len] = 'a';
}
```

Continuous Execution

```
main()
    append()
        r1 = len
        r1 = r1+1;
        len = r1
        buf[len] = 'a'
            ..
            ..
            ..
```

Memory Consistancy Issues

Source Code

```
NV int len = -1
NV char buf[]="";
main () {
append();
}
append () {
r1 = len;
r1 = r1 + 1;
len = r1;
buf[len] = 'a';
}
```

Intermittent Execution

```
main()
        append()
                r1 = len
                r1 = r1+1;
                len = r1
                reboot
main()
        append()
                r1 = len
                r1 = r1+1;

                        ..
                        ..
                        ..
```

Memory Consistancy Issues

Source Code

```
NV int len = -1
NV char buf[]="";
main () {
append();
}
append () {
r1 = len;
r1 = r1 + 1;
len = r1;
buf[len] = 'a';
}
```

Dynamic Checkpointing

```
main()
        append()
                r1 = len
________________________   -> checkpoint
                r1 = r1+1;
                len = r1
                reboot


                r1 = r1+1;
                len = r1
                buf[len] = 'a'
```

DINO (Death is not an option)

Contributions of the paper :

- Intermittent Execution Model (Two ways)
    - Concurrency
    - Control flow
- DINO Programming and Execution Model
- Evaluation of a working prototype of DINO
    - Including a compiler and runtime system for embedded energy-harvesting platforms

Intermittent Computing

Effects of Rebooting :

- All volatile memory is cleared, and control returns to the entry point of main().
- Nonvolatile state retains its contents across reboots.

http://ee.princeton.edu/events/programming-and-system-support-reliable-intermittent-computing

Memory

- Flash (nonvolatile)
- DRAM (volatile)
- SRAM (volatile)
- FRAM (nonvolatile)

Operating system implications of fast, cheap, non-volatile memory. In Workshop on Hot Topics in Operating Systems (HotOS), May 2011.

Memory

- **Flash** (nonvolatile)
- DRAM (volatile)
- SRAM (volatile)
- FRAM (nonvolatile)

Non-volatile memory but write speed are much slower than RAM.

Operating system implications of fast, cheap, non-volatile memory. In Workshop on Hot Topics in Operating Systems (HotOS), May 2011.

Memory

- Flash (nonvolatile)
- **DRAM** (volatile)
- SRAM (volatile)
- FRAM (nonvolatile)

Volatile memory, low cost. Used for Main memory or Graphics memory

Operating system implications of fast, cheap, non-volatile memory. In Workshop on Hot Topics in Operating Systems (HotOS), May 2011.

Memory

- Flash (nonvolatile)
- DRAM (volatile)
- **SRAM** (volatile)
- FRAM (nonvolatile)

Volatile memory, costly and faster. Used as caches.

Operating system implications of fast, cheap, non-volatile memory. In Workshop on Hot Topics in Operating Systems (HotOS), May 2011.

Memory

- Flash (nonvolatile)
- DRAM (volatile)
- SRAM (volatile)
- **FRAM** (nonvolatile)

FerroElectric RAM (FRAM), is a relatively new technology.

Non-volatile memory, speeds are comparable to RAM.

Operating system implications of fast, cheap, non-volatile memory. In Workshop on Hot Topics in Operating Systems (HotOS), May 2011.

Dynamic Checkpointing

Dynamic analysis determines when to copy the execution context—registers and some parts of volatile memory—to a reserved area in non-volatile memory.

Problems:

- Does not know where the execution restarts

Dynamic Checkpointing

Dynamic analysis determines when to copy the execution context—registers and some parts of volatile memory—to a reserved area in non-volatile memory.

Problems:

- Limited to only volatile memory
- Does not know where the execution restarts

Solution :

a programmer or static program analysis has to guess where execution will resume after a reboot,

Dynamic Checkpointing

Dynamic analysis determines when to copy the execution context—registers and some parts of volatile memory—to a reserved area in non-volatile memory.

Problems:

- Does not know where the execution restarts
- Volatile memory does not remain consistant across reboots
    - I/O operations to nonvolatile memory does not remain consistant across reboots

Memory Consistancy Issues

Source Code

```
NV int len = -1
NV char buf[]="";
main () {
append();
}
append () {
r1 = len;
r1 = r1 + 1;
len = r1;
buf[len] = 'a';
}
```

Dynamic Checkpointing

```
main()
        append()
                r1 = len
________________________   -> checkpoint
                r1 = r1+1;
                len = r1
                reboot

                r1 = len
                r1 = r1+1;
                len = r1
                buf[len] = 'a'
```

Memory Consistancy Issues

Non-volatile vs volatile memory:

The ISAs or compiler do not distinguish between writes to volatile an non-volatile memory for a programmer.

DINO (Death is not an option)

Contributions of the paper :

- Intermittent Execution Model (Two ways)
    - Concurrency
    - Control flow
- DINO Programming and Execution Model
- Evaluation of a working prototype of DINO
    - Including a compiler and runtime system for embedded energy-harvesting platforms

DINO (Death is not an option)

Contributions of the paper :

- Intermittent Execution Model (Two ways)
  - **Concurrency**
  - Control flow
- DINO Programming and Execution Model
- Evaluation of a working prototype of DINO
  - Including a compiler and runtime system for embedded energy-harvesting platforms

Intermittance as Concurrency

Reboot : Can be compared to the executing period being pre-empted and a pre-empting period begins executing.

Instance1

```
____________-> checkpoint
r1 = len
r1 = r1+1;
len = r1
reboot
```

Instance 2 (cont. after reboot)

```
____________ -> checkpoint
r1 = len
r1 = r1+1;
len = r1
reboot

..
..
..
```

Concurrency Model

Atomic Violations :

Atomicity : if code outside the sequence—i.e., code executing after a reboot—cannot observe its effects until all operations in the sequence complete

Isolation : operations cannot observe results from operations that are not part of the sequence

Data Inconsistancy

Atomicity : if code outside the sequence—i.e., code executing after a reboot—cannot observe its effects until all operations in the sequence complete

Instance1

____________-> checkpoint
r1 = len
r1 = r1+1;
len = r1
reboot

Instance 2 (cont. after reboot)

____________ -> checkpoint
r1 = **len**
r1 = r1+1;
len = r1
reboot

..
..
..

Data Inconsistancy

Isolation : operations cannot observe results from operations that are not part of the sequence

Instance1

______________-> checkpoint
r1 = len
r1 = r1+1;
len = r1
reboot

Instance 2 (cont. after reboot)

______________ -> checkpoint
r1 = **len**
r1 = r1+1;
len = r1
reboot

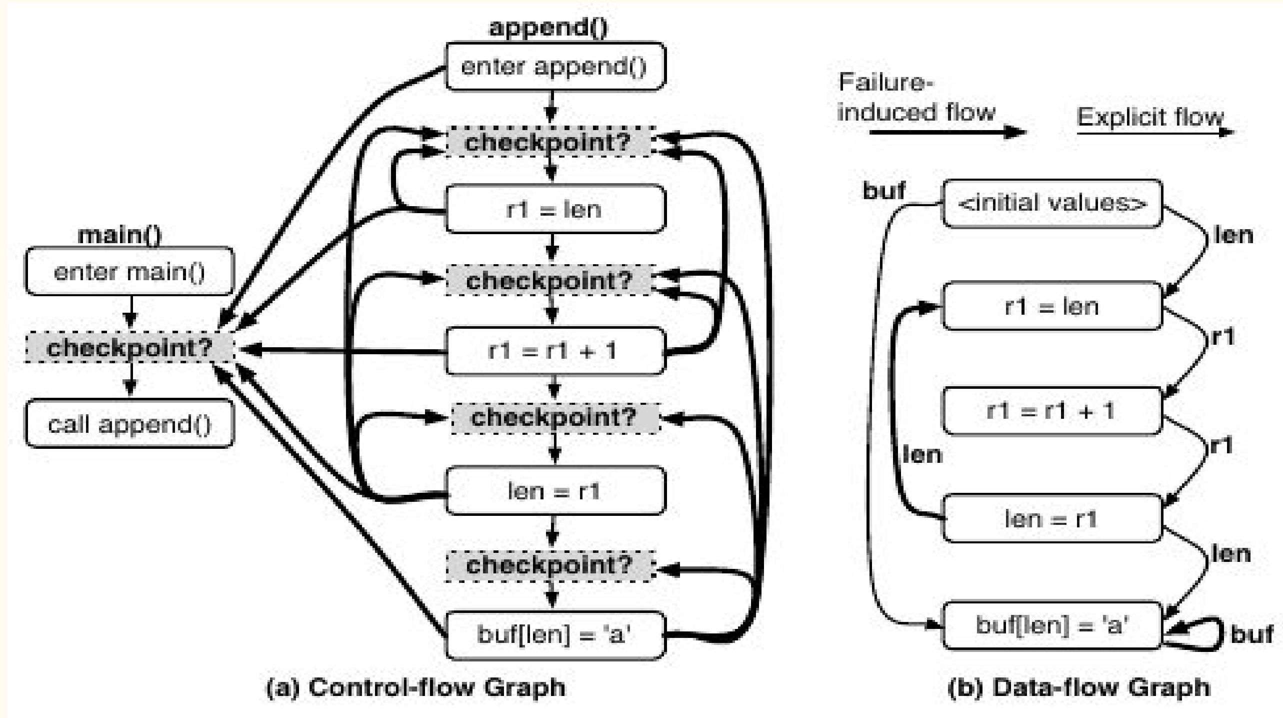
..
..
..

DINO (Death is not an option)

Contributions of the paper :

- Intermittent Execution Model (Two ways)
  - Concurrency
  - **Control flow**
- DINO Programming and Execution Model
- Evaluation of a working prototype of DINO
  - Including a compiler and runtime system for embedded energy-harvesting platforms

Intermittence as Control Flow



(a) Control-flow Graph (b) Data-flow Graph

Intermittence as Control Flow

Idempotent Computaion: A computation is idempotent if it can be repeatedly executed without producing a new result.

$$A \text{ (op) } A = A.$$

If it fails a function is said to be a violating Idempotence.

Intermittence as Control Flow

CFG (Contol Flow Graph)
Node (V) : A code point in the program
Edge (Vreboot, Vresume) : A control transfer from one code point to another across reboots

Intermittence as Control Flow

CFG (Contol Flow Graph)
Node (V) : A code point in the program
Edge (Vreboot, Vresume) : A control transfer from one code point to another across reboots

Number of edges = |V|^2

Intermittence as Control Flow

NV DFG (Non-Volatile Data Flow Graph):

Node (V) : An access or write to a nonvolatile mem. Loc. in the code

Edge (V_w, V_r) : Data written at V_w is read at V_r.

DINO (Death is not an option)

Contributions of the paper :

- Intermittent Execution Model (Two ways)
  - Concurrency
  - Control flow
- DINO **Programming Model** and Execution Model
- Evaluation of a working prototype of DINO
  - Including a compiler and runtime system for embedded energy-harvesting platforms

Programming Model

- Adds several features to the base programming model.
- Programmers can insert task boundaries to sub-divide long running tasks into semantically meaningful shorter tasks.
- Adds task-atomic semantics to C's programming model.

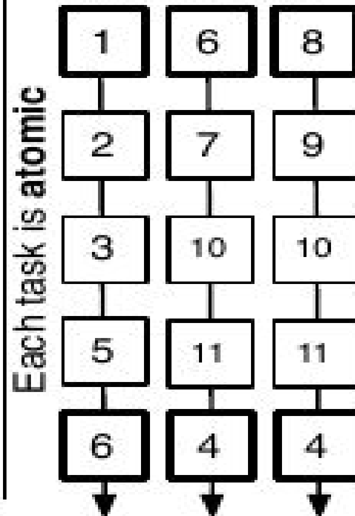
Programming Model

```
void main(){
0:  s = rd_sensor();
1:  DINO_task()
2:  c = classify(s);
3:  upd_stats(c)
4:  DINO_task()}

upd_stats(class c){
5:  if( c == CLASS1 ){
6:    DINO_task();
7:    c1++;
    }else{
8:    DINO_task();
9:    c2++;
    }
10:total++;
11:assert(total==c1+c2)}
```

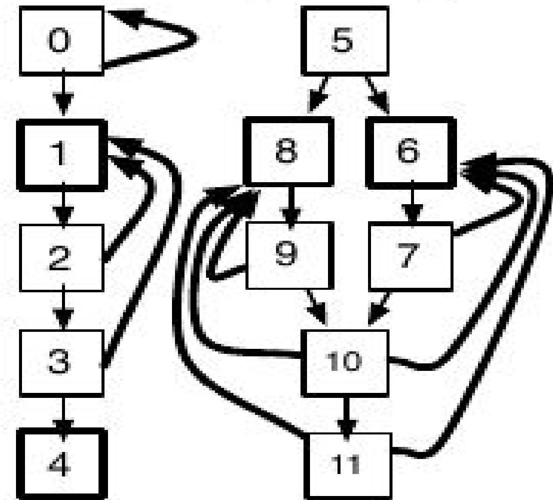
(a) DINO Program

Each task is a path between **boundaries.**



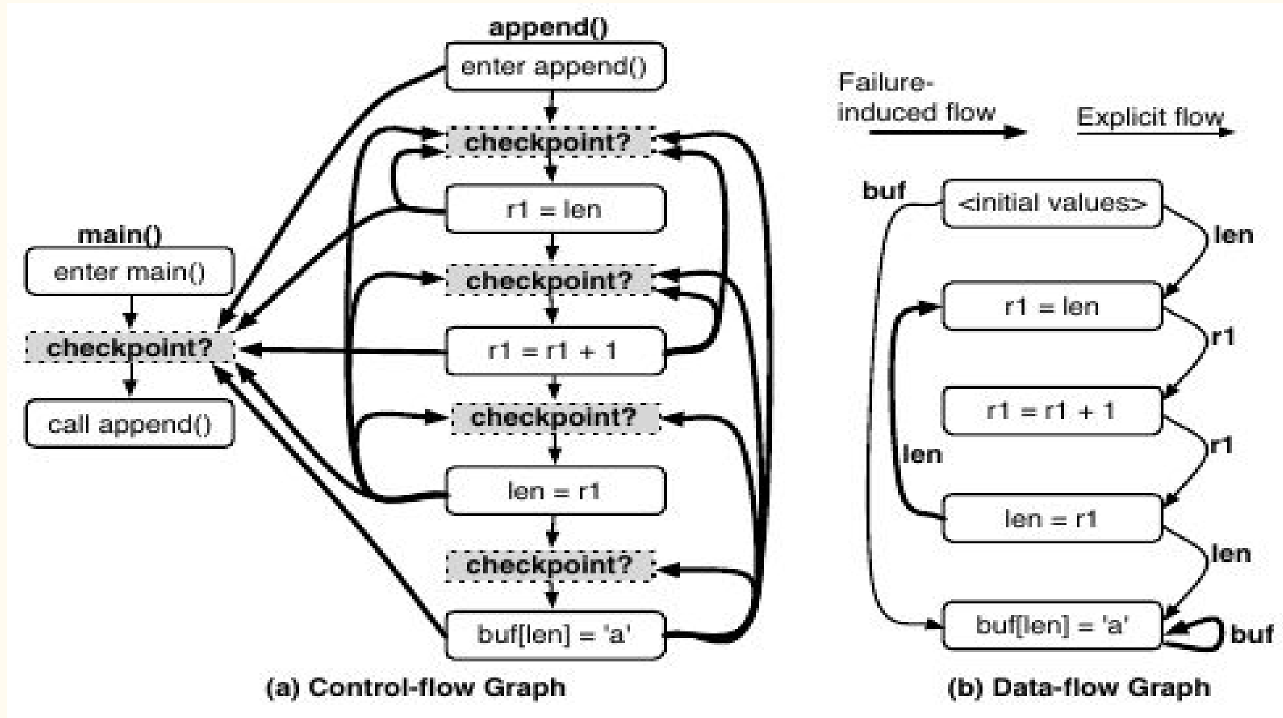
(b) DINO Tasks

main() **upd_stats()**



(c) DINO CFG

Intermittence as Control Flow



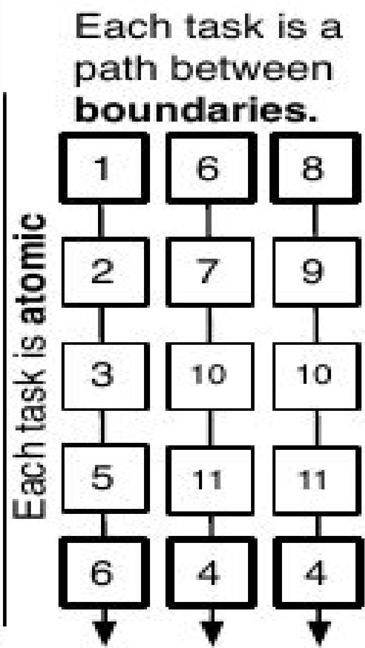
(a) Control-flow Graph (b) Data-flow Graph

Programming Model

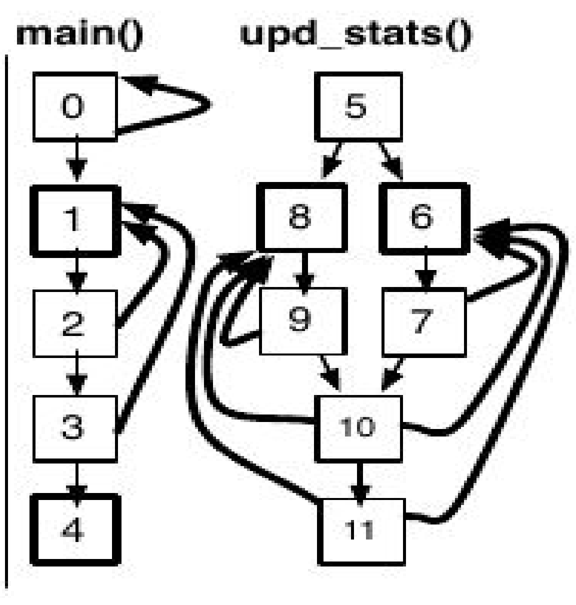
```
void main(){
0:  s = rd_sensor();
1:  DINO_task()
2:  c = classify(s);
3:  upd_stats(c)
4:  DINO_task()}

upd_stats(class c){
5:  if( c == CLASS1 ){
6:    DINO_task();
7:    c1++;
    }else{
8:    DINO_task();
9:    c2++;
    }
10:total++;
11:assert(total==c1+c2)}
```

(a) DINO Program



(b) DINO Tasks



(c) DINO CFG

Execution Model

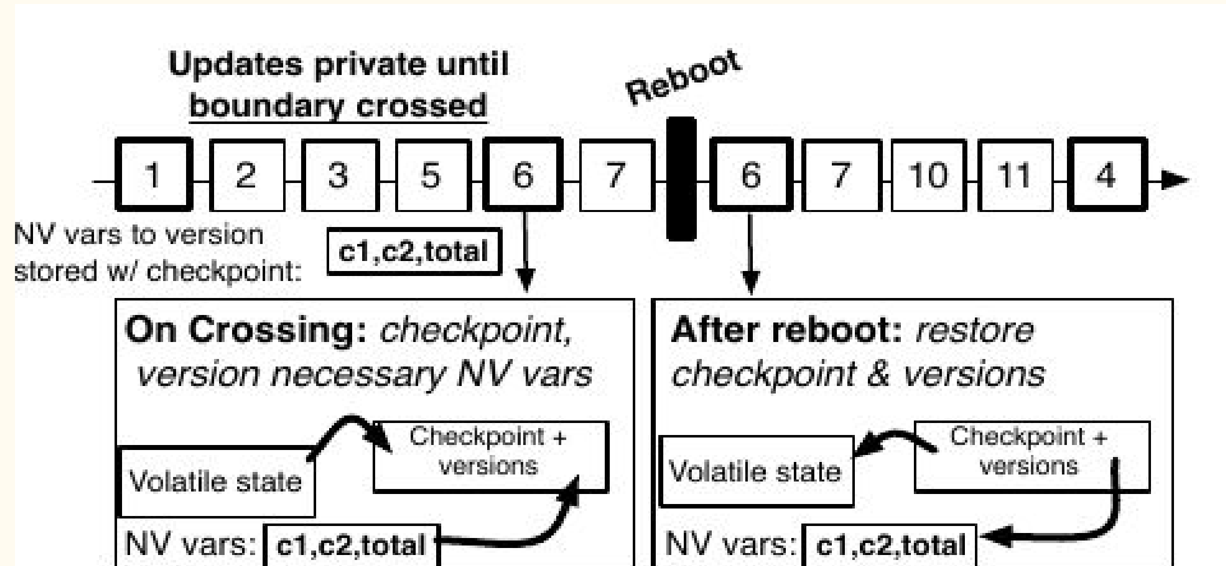
Checkpointing

- DINO has reserved area in nonvolatile memory
- At runtime copies the registers and program stack to the reserved area.

Execution Model

Checkpointing

- DINO has reserved area in nonvolatile memory
- At runtime copies the registers and program stack to the reserved area.



Execution Model

Checkpointing

- DINO has reserved area in nonvolatile memory
- At task boundary at runtime copies the registers and program stack to the reserved area.

Data Versioning

- At task boundaries, we copy non volatile variables to volatile memory
- Later during checkpoint this is copied to checkpointing area.
- At reboot we restore these variables too.
- This prevents the failure induced data flow due to repeated execution of partial code.

Task Boundary Placement

Task Boundary Placement

- Task boundary overhead and failure-recovery cost
- Minimising operations in task with the irrevocable non idempotent I/O

Vs

- Frequency of interference

DINO (Death is not an option)

A programming and Execution model to address the challenges presented

Programmers can insert task boundaries to sub-divide long running tasks into semantically meaningful shorter tasks.

Runtime Overhead: 1.8 -2.7x

Eliminated possible control failure transfer : 5 -9x

Architecture and Implementation

- compiler that analyzes programs and inserts DINO runtime code
- runtime system: implements execution model e.g. checkpointing, data versioning, and recovery

Compiler

- Uses data-flow analysis to identify potentially inconsistent data that must be versioned at a task boundary
- Translates programmer-defined task boundaries into calls into the DINO runtime library
- Analyzes task boundary placement and suggests changes to reduce run-time checkpointing cost

Identifying Potentially Inconsistent Data

- It uses an interprocedural context- and flow-sensitive analysis to find these variables
- For each NV store S to a location LS, it searches backward along all control-flow paths until it hits a “most recent” task boundary on each path
- Between each such task boundary TB & S, the analysis looks for loads from LS that occur before S
-

Compiling Task Boundaries

- Encountering DINO_task(), the DINO compiler first determines the set of nonvolatile variables to version
- Replaces the call to DINO_task() with calls into the DINO runtime library that version the relevant nonvolatile state dino_version() and dino_checkpoint()

Analyzing Task Cost

- first heuristic computes the size of the stack DINO must checkpoint. Sums the size of LLVM IR stack allocations
- second heuristic estimates the likelihood that a task will experience a reboot because it uses a power-hungry peripheral.

Runtime System

- When the program begins in `main()` , it first executes a call to `restore_state()` inserted by the DINO compiler. program counter, stack pointer, and frame pointer, NV and V data are restored.
- Also stores data to be versioned and volatile data into flash at each task boundary.

Why Not rely on hardware support?

- Systems that suffer consistency problems are already widely available without special hardware requirement
- new hardware features can increase energy requirements when underutilized increase design complexity, and increase device cost.
- Specialized hardware support requiring ISA changes raises new programmability barriers and complicates compiler design

Applications

- Three hardware/software embedded systems to evaluate DINO
- Each runs on a different energy-harvesting frontend

Activity Recognition

- Adapted a machine-learning–based activity-recognition system to run on the intermittently powered WISP hardware platform
- WISP harvests radio-frequency
- has an Analog Devices ADXL326z low-power accelerometer connected to an MSP430FR5969 MCU via 4-wire SPI
- AR maintains a time series of three-dimensional accelerometer values to distinguish shaking from resting
- AR counts total and per-class classifica-tions in nonvolatile memory.
- After classifying, AR's code updates the total and per-class counts in NV. And this must be done atomically.
- The per-class counts should sum to the total count and any discrepancy represents error.

Data Summarizer

- Implemented a data-summarization (DS) application on TI's TS430RGZ-48C project board with an MSP430FR5969 microcontroller
- Connected the board to a Powercast Powerharvester P2110 energy harvester and a half-wavelength wire antenna
- summarizes data as a key–value histogram in which each key maps to a frequency value
- One function adds a new data sample to the histogram and another one sorts the histogram by value using insertion sort
- Test harness for DS inserts random values, counting 2000 insertions with a nonvolatile counter and sorting after every 20 insertions
- sorting routine swaps histogram keys using a volatile temporary variable. If interrupted then two bins will have same key.
- structural invariant checks after each sorting step, is that each key appears exactly once in the histogram

MIDI Interface

- Implemented a radial MIDI (Musical Instrument Digital Interface) interface (MI) on TI's TS430RGZ-48C project board with an MSP430FR5969 microcontroller
- We connected the project board to a Peppermill power front end [36] that harvests the mechanical power of a manually rotated DC motor for use by the project board
- MI generates batches of MIDI Note On messages with a fixed pitch and a velocity proportional to the motor's speed. It stores messages in a circular-list data structure and tracks the index of the message being assembled, incrementing the index to store each new message.
- When all entries are populated, MI copies the messages to a separate memory region, clears the old messages, and resets the index
- A power failure can trigger an idempotence violation that increments the index multiple times

Evaluation

Config.	AR Err.	AR Rbts.	AR Ovhd.	DS ✗	DS ✓	DS Rbts.	DS Ovhd.	MI ✗	MI ✓
Baseline	6.8%	1,970	1.0x	3	7	13.4	1.0x	10	0
DINO	0.0%	2,619	1.8x	0	11	72.5	2.7x	0	10

- DINO keeps data consistent despite intermittence
- DINO's overheads are reasonable, especially given its correctness gains
- DINO reduces the complexity of reasoning about control flow
- We show that the our task-cost analysis correctly reports costly tasks.

DINO Enforces Consistency

- Columns 2–3 how the error AR experiences with and without DINO at 40cm from RF power [30dBm (1W)]
- AR sees no error with DINO but suffers nearly 7% Error Without DINO
- Columns 4–5 how failure data for DS running with and without DINO at 60cm from RF power.
- DS does not fail with DINO, but fails in three out of ten trials without DINO. DINO causes about 5.5× as many reboots
- Columns 9–10 how failure data for MI with and without DINO
- MI does not fail with DINO, but fails 100% of the Time without DINO, processing just 975 messages on average before failing.

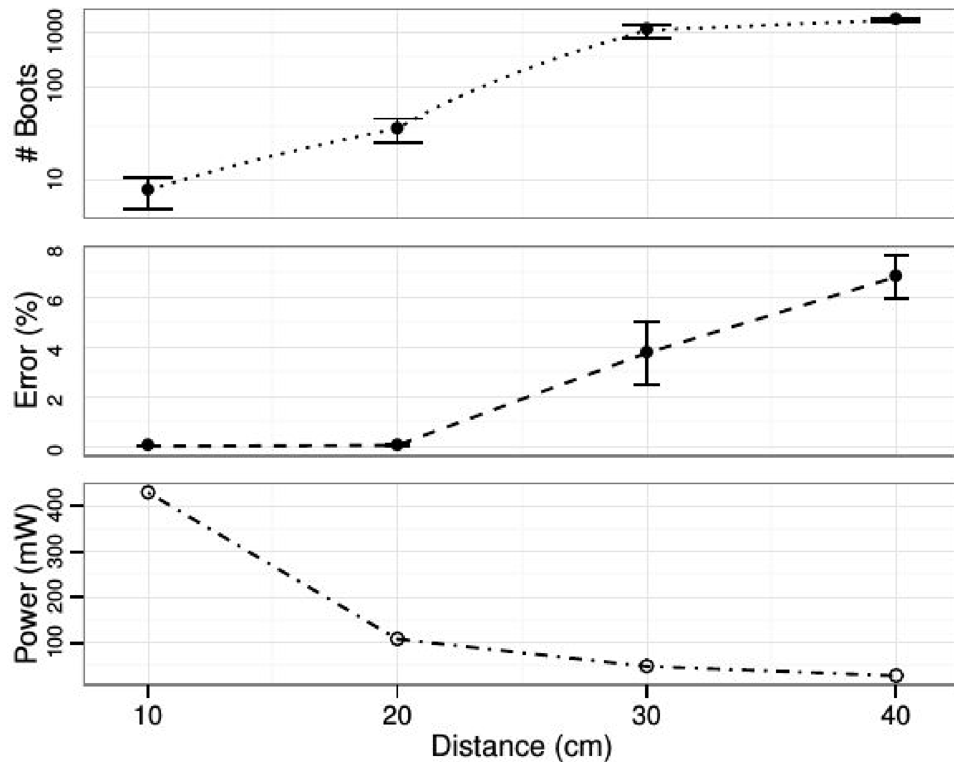


Figure 10: Error vs. Distance for AR without DINO. Under RF harvesting, AR without DINO experiences reboots (top, log scale) and error rates (middle) that scale with distance. Error bars represent standard error. The bottom plot shows analytical (computed) available power.

DINO Imposes Reasonable Overheads

Two main sources of run-time overhead

- Time spent checkpointing and versioning data
- second is the increase in reboots when running on intermittent power. Reboots cost cycles and require restoring data

➔ externally timed each of our experiments' executions withand without DINO
➔ The run time of DS with DINO is 2.7× higher than without DINO. The run time of AR with DINO is 1.8× higher.
➔ The number of reboots with DINO is higher by 30% for AR and about 550% higher for DS.
➔ DINO uses a checkpointing scheme that incurs a large, fixed 4KB storage cost to store checkpoints and versions

DINO Reduces Control Complexity

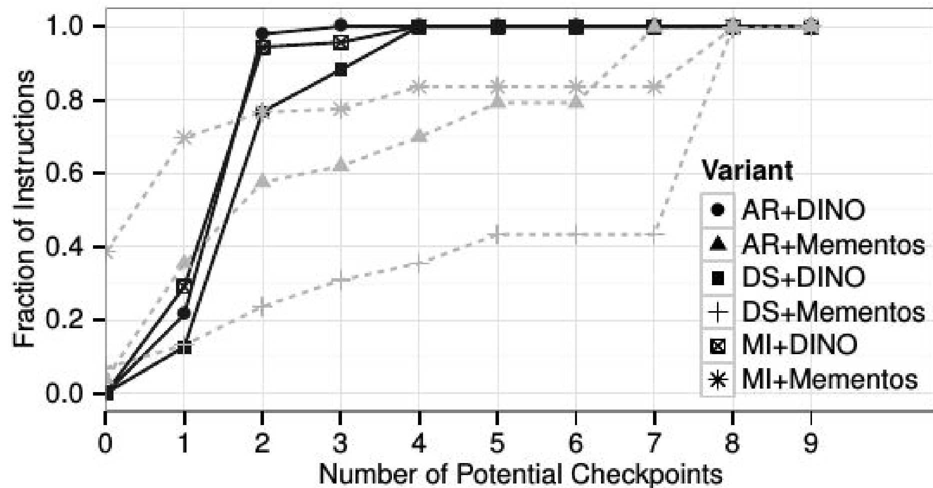


Figure 11: CDF of checkpoints reachable by each instruction. The x-axis shows at how many points execution may resume after a reboot for what fraction of instructions on the y-axis.

DINO Helps Place Task Boundaries

- started with unmodified (non-DINO) application code and scripted the addition of a single task boundary at every possible program location that did not cause a compilation error
- manually verified the correct presence of warnings for tasks with peripherals in AR, DS, and MI
- For DS, 32% of variants generated a suggestion
- Suggestions
- garnered a maximum reduction of checkpointed data of 50 byte the equivalent of 1.7× the minimum checkpoint size. minimum savings per task boundary was 2 bytesand the average was 8 byte
- For AR, following the compiler’s suggestions yielded average savings of 6 bytes per task boundary, or 20% of the minimum checkpoint size
- For MI, there were only two suggestions, with a maximum potential savings of 2 bytes to checkpoint, or 7% the minimum checkpoint size

Thank You!!!