Internet of Things

Lecture 16

Energy 2

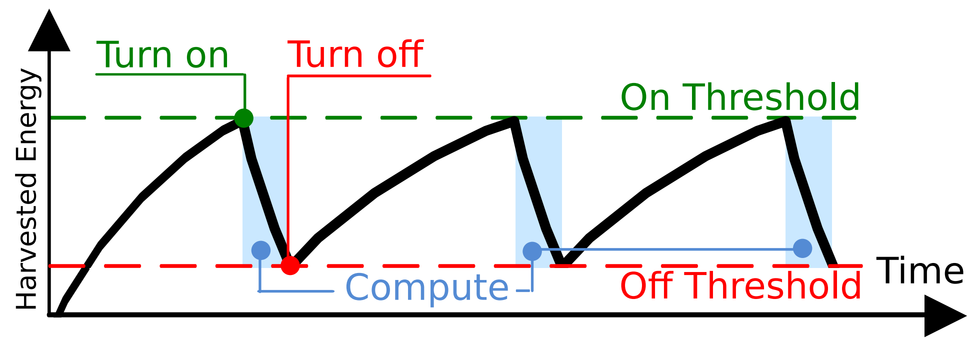
Intermittent computing

Michael Engel

Intermittent Computing

Question:
How can we handle the unreliable availability of energy when using energy harvesting?

- There are often only short amounts of runtime between long periods of recharging
- How can *application state* be preserved during phases without power?



Approaches:

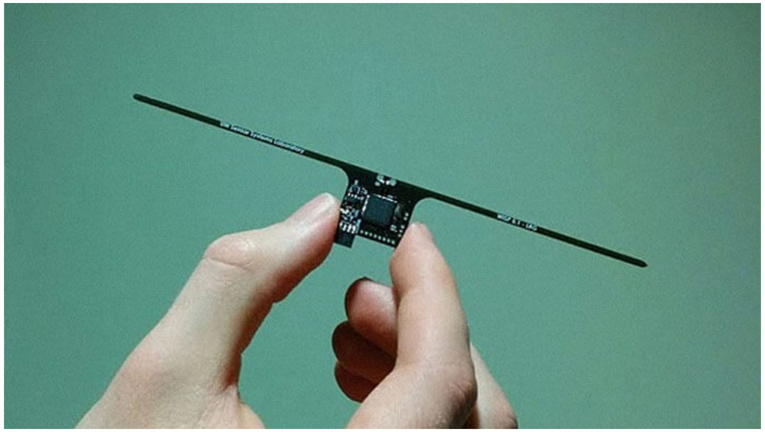
- Compute only when enough energy is available, turn off the device in phases without sufficient power
  - How can we guarantee ***progress in the computation?***

→ ***Intermittent computing approaches***

Intermittent operation in the IoT

- IoT devices harvest and buffer energy as it is available and operate when sufficient energy is stored [1]

- Operation in these devices is ***intermittent*** because energy is not always available to harvest
  - Even when energy is available, we need to buffer enough energy to do a useful amount of work, which takes time



WISP RF-powered energy-harvesting platform

- The key distinction between a conventional execution and intermittent execution is that a conventionally executing program is assumed to run to completion but an intermittent execution must span power failures

Intermittent operation in the IoT

- Intermittent execution…
  - makes control-flow unpredictable
  - compromises an application’s forward progress
  - leaves memory inconsistent
  - leaves a device inconsistent with its environment
  - and complicates device-to-device communication

- To tolerate power failures that occur hundreds of times per second, multiple layers of the system require an intermittence-aware design, including languages, runtimes, and application logic

Intermittent computing challenges

- **Control-flow:**
  - To an executing program, resuming after a power failure is a discontinuity in control-flow that is not explicitly expressed in source code
  - Programmers must deal with *implicit control flows* to potentially unpredictable points in an execution's history, such as a recent checkpoint or the beginning of a task
- **Data consistency:**
  - A naive combination of checkpointing and direct access to non-volatile memory in an intermittent device can lead to memory inconsistencies
- **Environmental consistency:**
  - Intermittently operating devices receive inputs via sensors
  - Sensed data become stale and unusable if they are buffered across a long time period without harvestable energy
- **Concurrency:**
  - Sensors, peripheral devices, and collections of MCUs may all operate concurrently as a single, intermittent device

Solution approaches

- Fail-restart is the common case. We have to:
  - Ensure Progress
  - Ensure Correctness

- Common techniques:
  - **Checkpointing**
    - General checkpoints interleaved in code
    - State is saved before power off
    - State is restored at power on
  - **Task-based**
    - Computation divided into separate tasks
    - Task internal state is thrown away at power off
    - Task restarted from scratch at power on

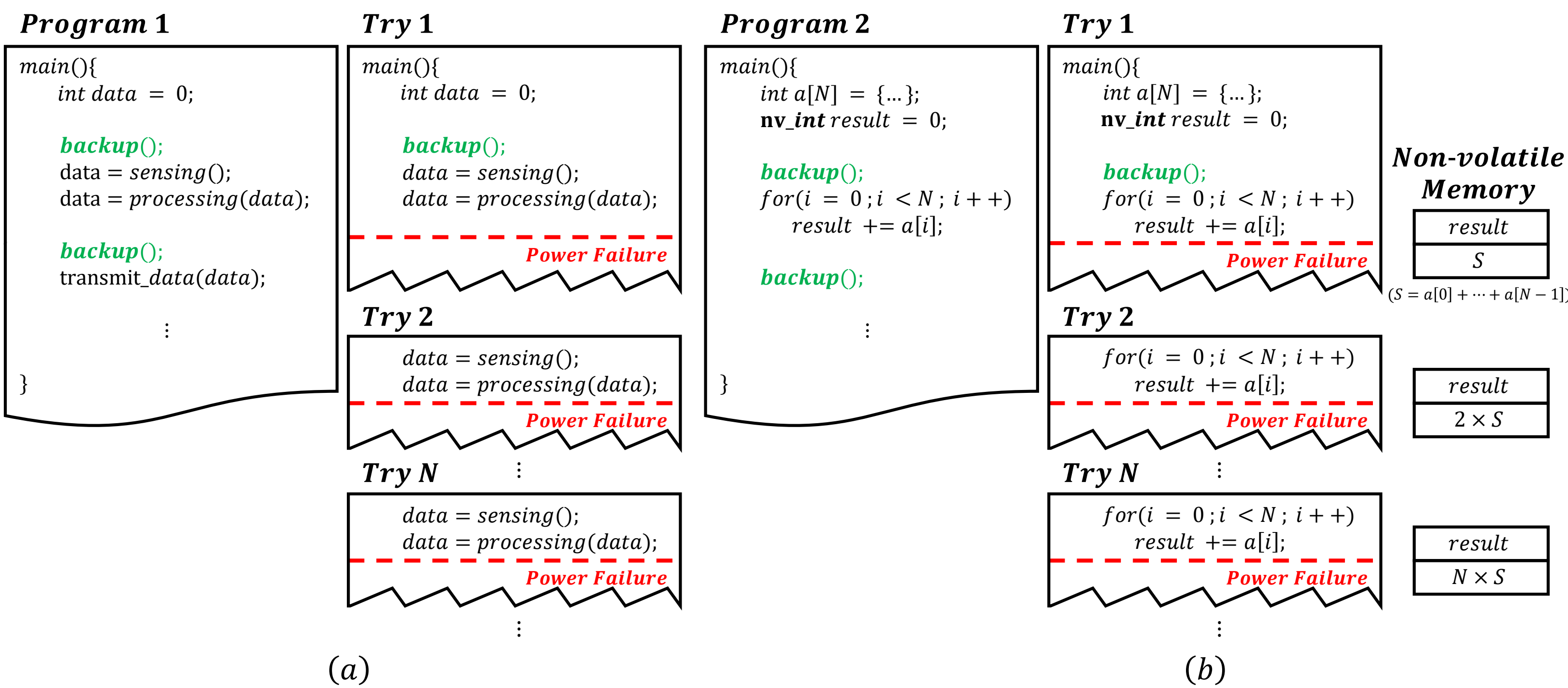
Intermittent computing and memory

- Software running on an intermittently operating device executes until energy is depleted and the device "browns out"
- When energy is again available, software resumes execution from some point in the history of its execution
  - the beginning of `main()` or a checkpoint

- Ideally, state of the computation has to be stored in non-volatile memory before power loss
  - enables software to continue at the point it was stopped due to power loss
  - ***checkpointing*** approaches used to store runtime data
    - processor registers, stack, heap
- **Challenge:** checkpointing uses time and energy
  can cause its own set of problems

Examples for Intermittent computing problems

- Problems that can occur in intermittent computing using checkpointing (from [2]):
  - (a) A problem that occurs when forward progress is not ensured.
  - (b) A problem that occurs when data consistency is not ensured.



Memory in IoT nodes

- **Volatile** memory – **loses** its state on a power failure
  - SRAM – caches, SPM, μC RAM: fast, energy efficient
  - DRAM – larger systems: main memory
  - SRAM is also used for CPU registers and caches

- **Non-volatile** memory – **retains** its state on a power failure
  - NAND Flash – SD cards, SSDs: write/erase in blocks only
  - NOR Flash – small capacity: byte erasable/writable
  - EEPROM – erasable ROM: byte erasable/writable
  - **FRAM** – ferroelectric: byte erasable/writable
  - Battery backed up SRAM – SRAM with battery, requires energy to retain contents when rest of system powered off

FRAM applications

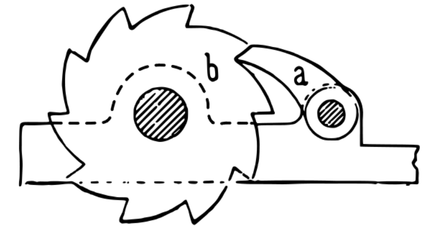
- Data logger in portable/implantable medical devices, as FRAM consumes less energy compared to other non-volatile memories such as EEPROM
- Event-data-recorder in automotive systems to capture the critical system data even in case of Crash or failure
- FRAM is used smart meters for its fast write and High endurance
- In Industrial PLC’s, FRAM is an ideal replacement for battery-backed SRAM (BBSRAM) and EEPROM to log machine data such as CNC tool machine position etc.

FRAM in TI MSP430 microcontrollers

- Texas Instruments microcontroller MSP430FR5994
  - 16 bit, 16 MHz microcontroller, 256KB ***FRAM***, 8KB SRAM
- FRAM [3]: ferro-electric non-volatile memory
  - long write endurance without degradation
    - 10^{10}–10^{15} read/write cycles
  - > 10 years data retention
  - does not require pre-erase, every FRAM write is nonvolatile

- Trade-offs using FRAM instead of SRAM
  - FRAM access speed is limited to ~8 MHz (ca. 100 ns)
    - SRAM can be accessed at the maximum device operating frequency
  - FRAM access results in a higher power consumption compared with SRAM

Example solution: Ratchet



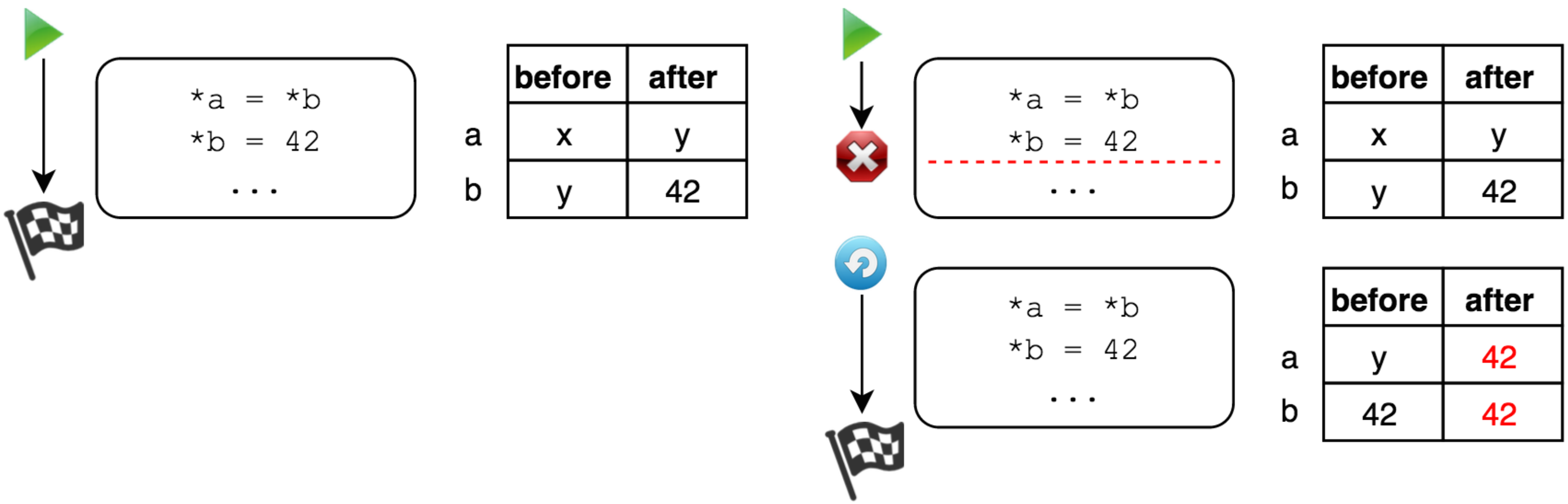
- Intermittent computation without hardware support or programmer intervention [4]
- Prior work: need specialized hardware / programmers
  - *Ratchet eliminates this need*
- Compiler automatically adds checkpoints to code between idempotent sections
  - Identifies *idempotent sections*

- Uses non-volatile memory exclusively
  - No volatile memory used
  - Does this solve the problem immediately?

Ratchet: Checkpointing problems

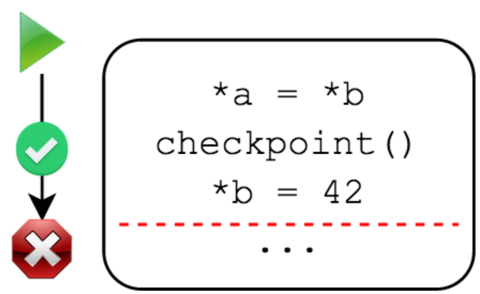
- Checkpoints cannot be added at arbitrary locations in a program's execution

- The Write-after-Read (WAR) problem:

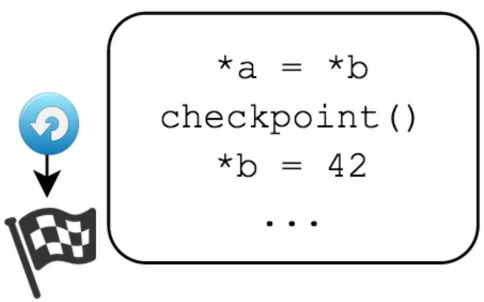


Ratchet: Idempotent sections

- Identify sections in the code which can be re-executed to get the same result
- Write-After-Read (WAR) problem:
  - Ratchet inserts a checkpoint between the write and the read



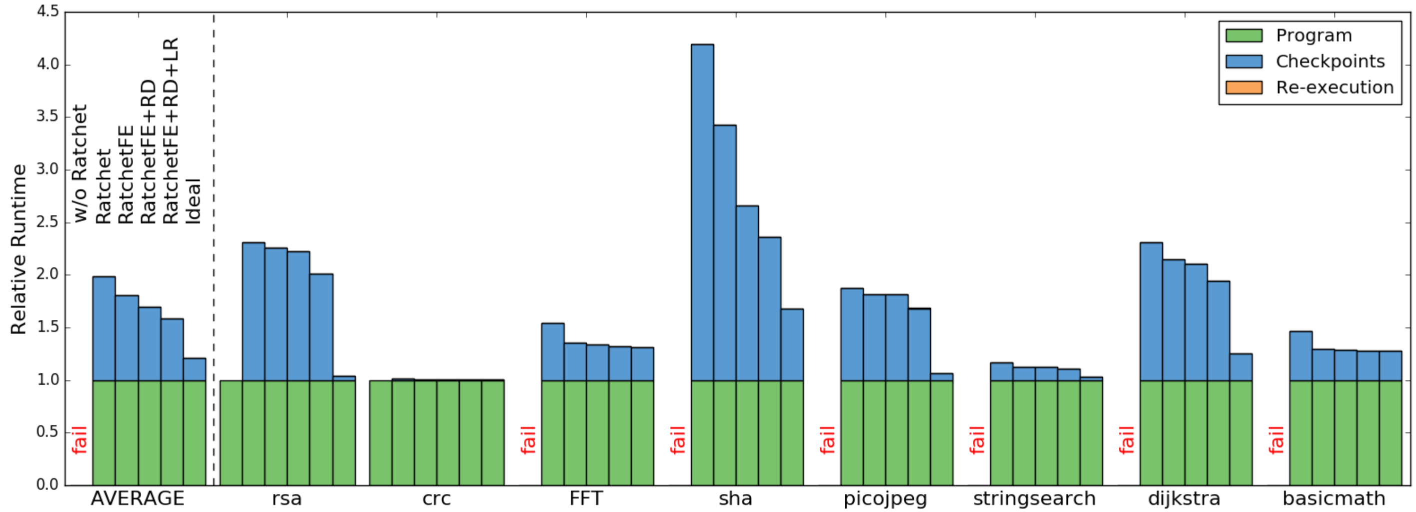
	before	after
a	x	y
b	y	42



	before	after
a	y	y
b	42	42

Ratchet overhead

Execution time:



FE: Function entry optimization, RD: Remove redundancy, LR: Live registers only

Code size:

(added recovery code, checkpoint calls)

Program	Ratchet	Uninstrumented	Change
AVERAGE	563720	560824	1.79%
rsa	41326	40694	1.55%
crc	36037	34677	3.92%
FFT	182362	183612	-0.68%
sha	3286631	3284544	0.06%
picojpeg	379134	373051	1.63%
stringsearch	184656	177567	3.99%
dijkstra	183554	178465	2.85%
basicmath	216053	213978	0.96%

Table 2: Code size increase due to Ratchet (sizes are in bytes).

Example solution: Alpaca

- Ratchet only works on devices with only non-volatile memory
  - Many off-the-shelf microcontrollers have *hybrid memory*
  - Costs more energy and time than volatile memory

- Ratchet approach limited by static analysis

- Alpaca [5] uses a **static task model**
  - Does not use checkpoints
  - Tasks manipulate privatized copies of data
  - Commits modifications atomically upon completion
  - Power failure – private data discarded with no cost
    - Similar to transactional memory with redo-logging

Task-based programming in Alpaca

- Programmer decomposes program into tasks
    - Explicitly transfers control between tasks

- Task-**shared** Variables: global scope, non-volatile
- Task-**local** Variables: private scope, initialized by task, volatile

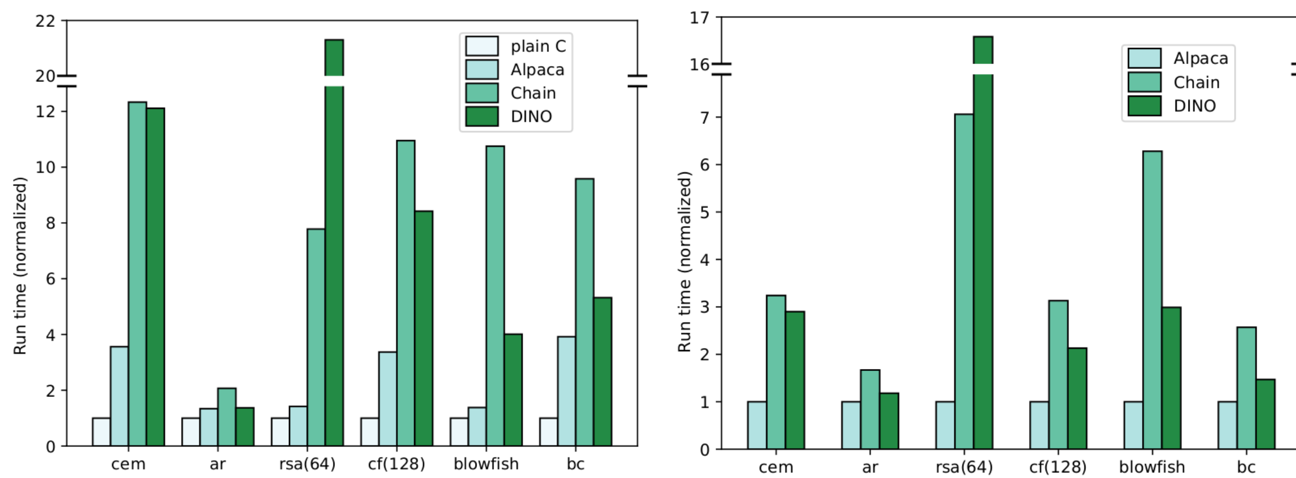
- Guarantee **task atomicity**

- **Values are privatized** (example: scalars)
    - Variable is copied to private buffer (in non-volatile memory)
    - Subsequent accesses redirected to private buffer
    - Optimization:
      only privatize vars involved in WAR dependencies

Alpaca overhead

Performance:

Normalized to
(a) Plain C
(b) Alpaca

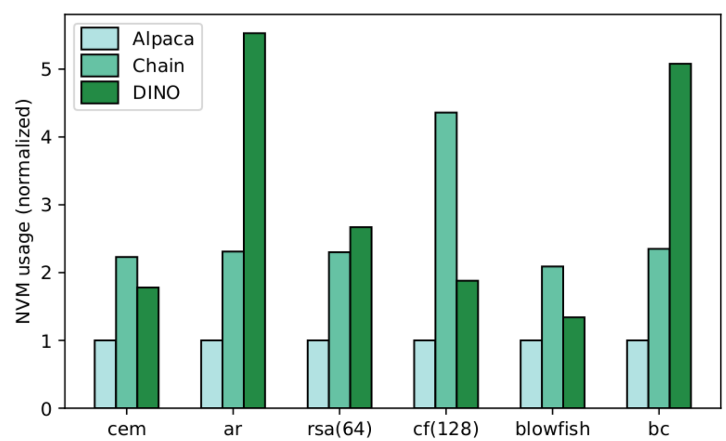


(a) On continuous power

(b) On harvested energy

Memory:

(Low non-volatile
memory usage
important, compared
to alternative solutions)



Example solution: Chinchilla

- Hard to guarantee forward-progress / termination with previous approaches:

  - Explicit Task Model (Alpaca)
    - Requires careful programming for non-termination
    - Hard to estimate energy use for various task input

  - Automatic Checkpointing (Ratchet)
    - Blindly insert checkpoints without considering non-termination
    - No programmer control over duration / energy-consumption of a task/section (cannot be fixed)

Example solution: Chinchilla

- Chinchilla idea: use **adaptive dynamic checkpointing**
  - Conservatively insert checkpoints to avoid non-termination
  - Dynamically disable checkpoints to minimize overhead

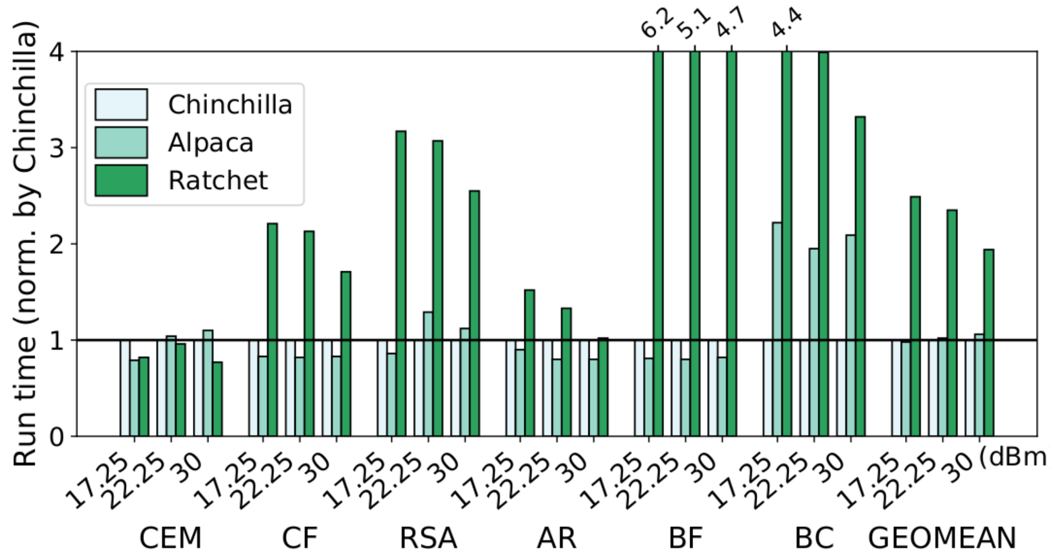
- Decompose code into short, predictable blocks
  - Criteria
    - Statically defined
    - Frequent enough
    - Easy to measure energy cost
    - Low energy variance
  - **Basic blocks** or user-defined **atomic blocks** (using the added "atomic" keyword)

Chinchilla overhead vs. Ratchet and Alpaca

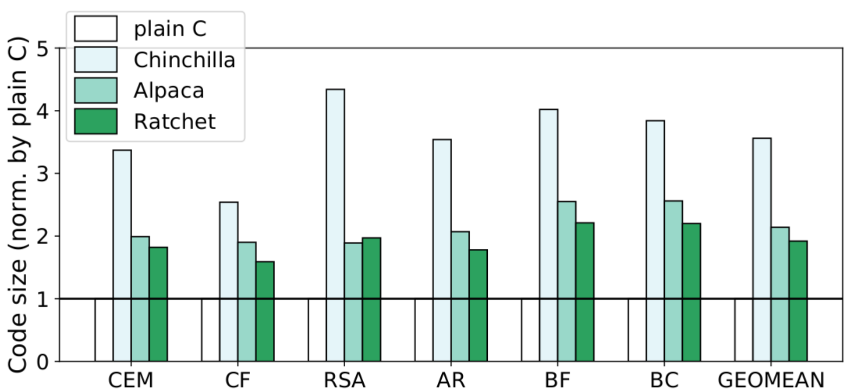
Performance:

Average **2.25x** speedup over **Ratchet**

Only an average **2%** speedup over **Alpaca** (hand optimized)



Code size:



Number of checkpoints taken:

# Chkpt.	CEM	CF	RSA	AR	BF	BC
Chinchilla	30	10	16	26	175	15
Alpaca	1611	452	315	265	1081	710
Ratchet	2319	2478	7643	2911	31881	8907

Clear advantage for selective checkpointing in Chinchilla!

Conclusion

- Intermittent computing challenges
  - progress of computation
  - correctness of computation

  …in the presence of frequent power failures

- Non-volatile memory used to store important state
  - e.g. FRAM

- Two major approaches:
  - checkpointing
  - task-based decomposition of code

- Advantages and drawbacks of different approaches discussed and compared

References

[1] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng4, and Emily Ruppel, *Intermittent Computing: Challenges and Opportunities*, Leibniz International Proceedings in Informatics, Article No. 8; pp. 8:1–8:14, 2017

[2] Kwak, Junho, Hyeongrae Kim, and Jeonghun Cho. 2021, *ICEr: An Intermittent Computing Environment Based on a Run-Time Module for Energy-Harvesting IoT Devices with NVRAM*, Electronics 10, no. 8: 879

[3] Texas Instruments, *MSP430TM FRAM Technology – How To and Best Practices*, Document SLAA628B, 2021

[4] Joel Van Der Woude and Mathew Hicks, *Intermittent computation without hardware support or programmer intervention,* In USENIX Conference on Operating Systems Design and Implementation (OSDI), pages 17–32, November 2016

[5] Kiwan Maeng, Alexei Colin, and Brandon Lucia, *Alpaca: intermittent execution without checkpoints,* Proc. ACM Program. Lang., OOPSLA, 2017

[6] Kiwan Maeng and Brandon Lucia, *Adaptive dynamic checkpointing for safe efficient intermittent computing,* In Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI'18), USENIX Association, 2018

[7] Domenico Balsama, Alex Weddell, Geoff Merrettt, Bashir Al-Hashimi, Davide Brunelli, and Luca Benini, *Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems,* IEEE Embedded System Letters, 7(1):15–18, March 2015]