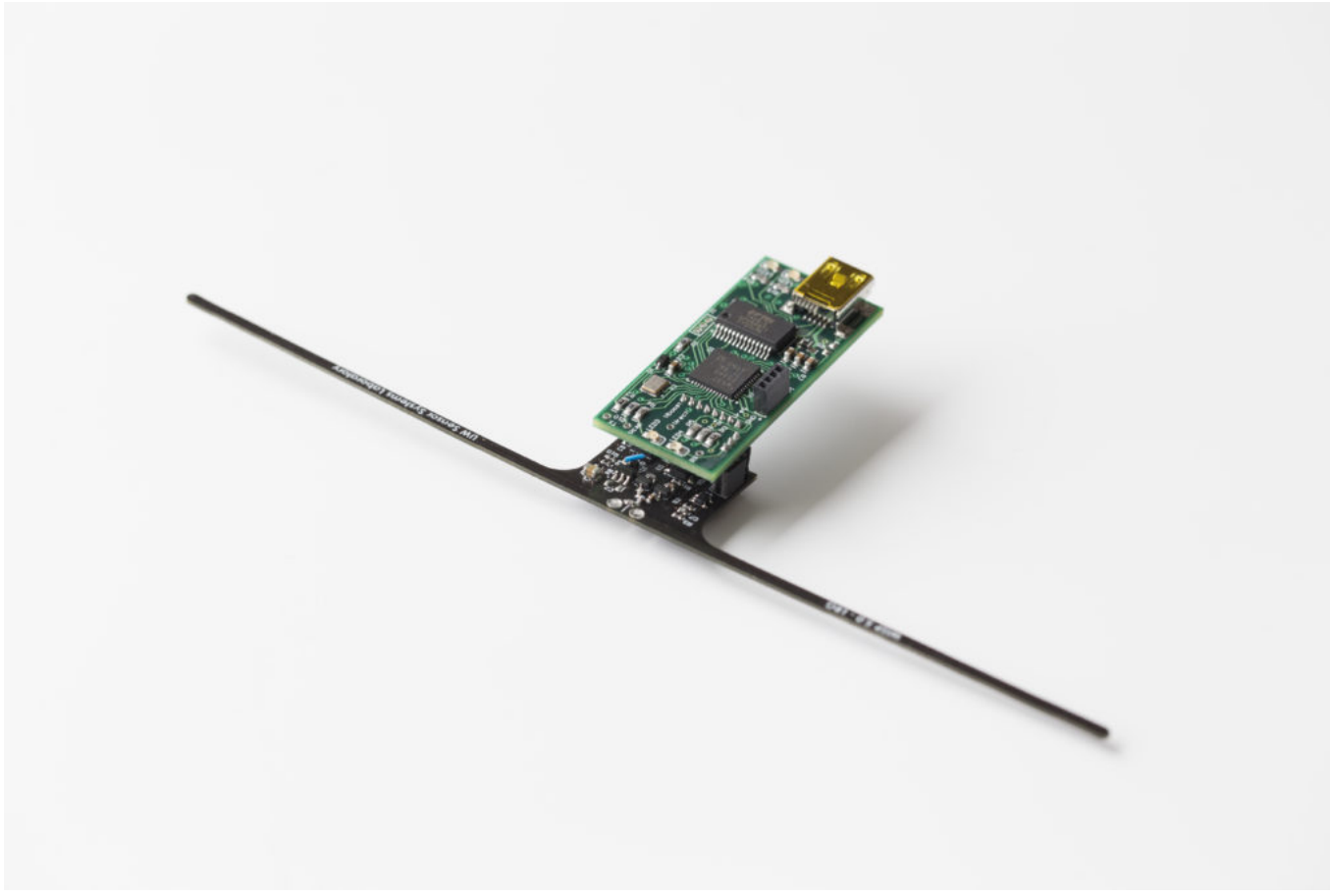
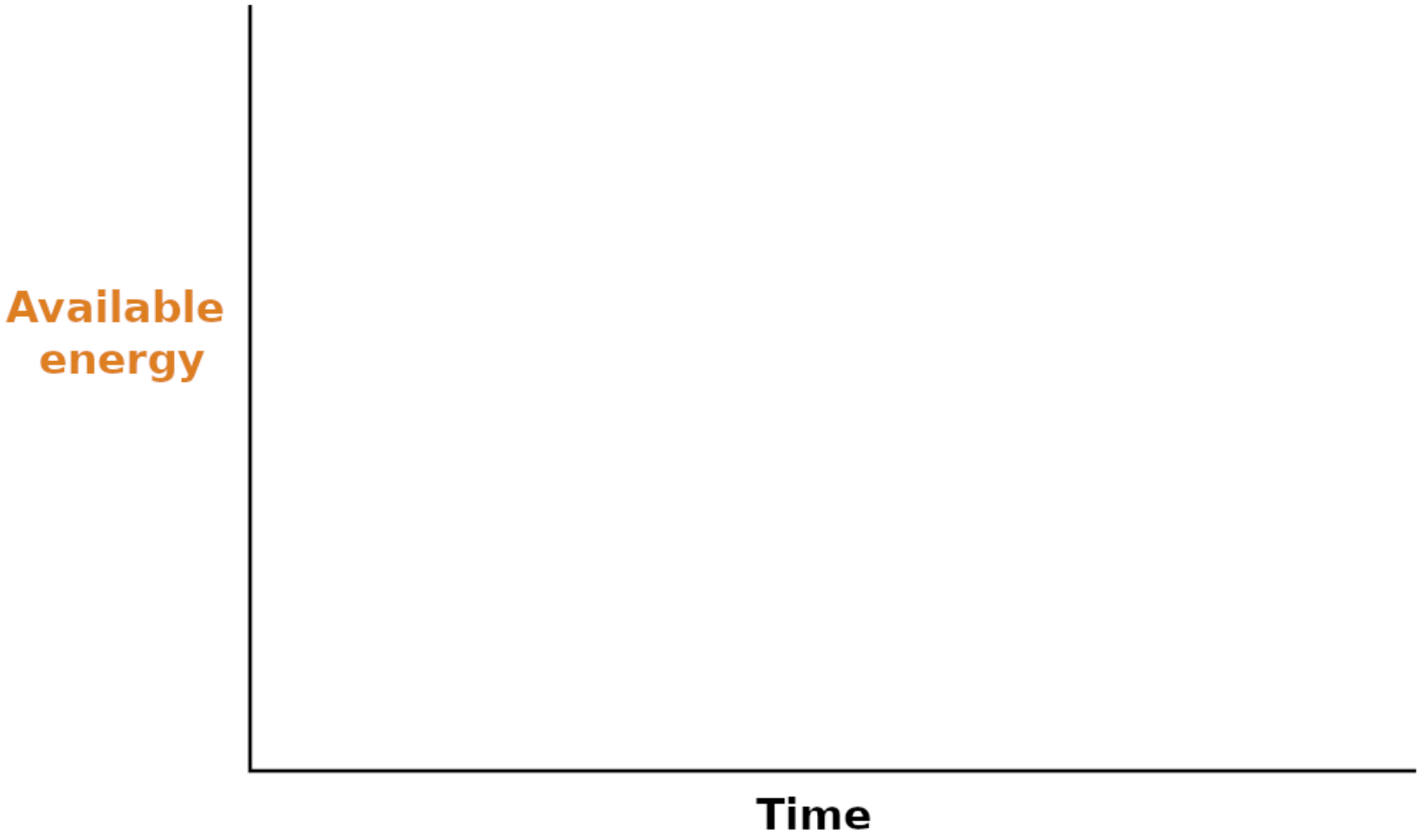
Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing

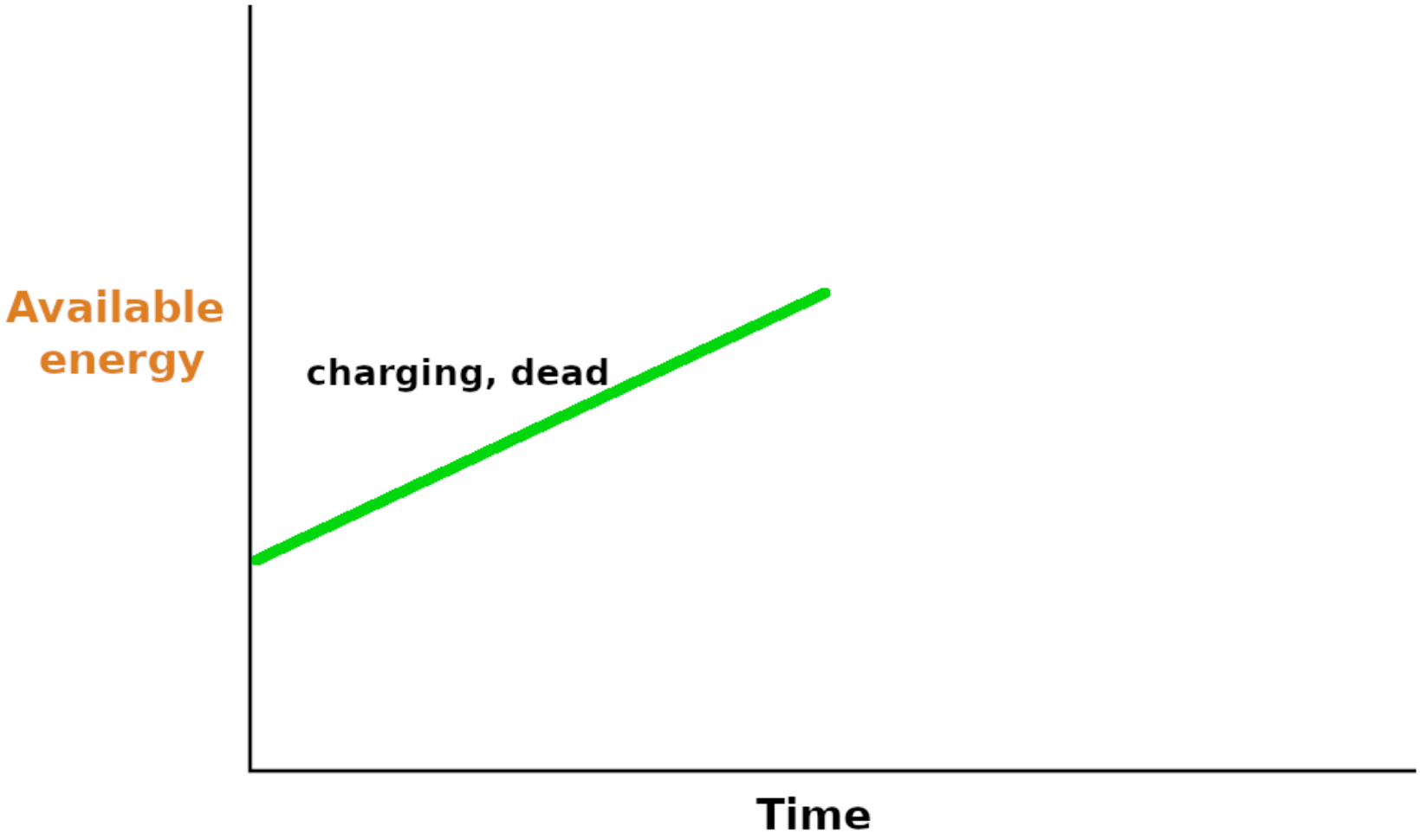
Kiwan Maeng Brandon Lucia

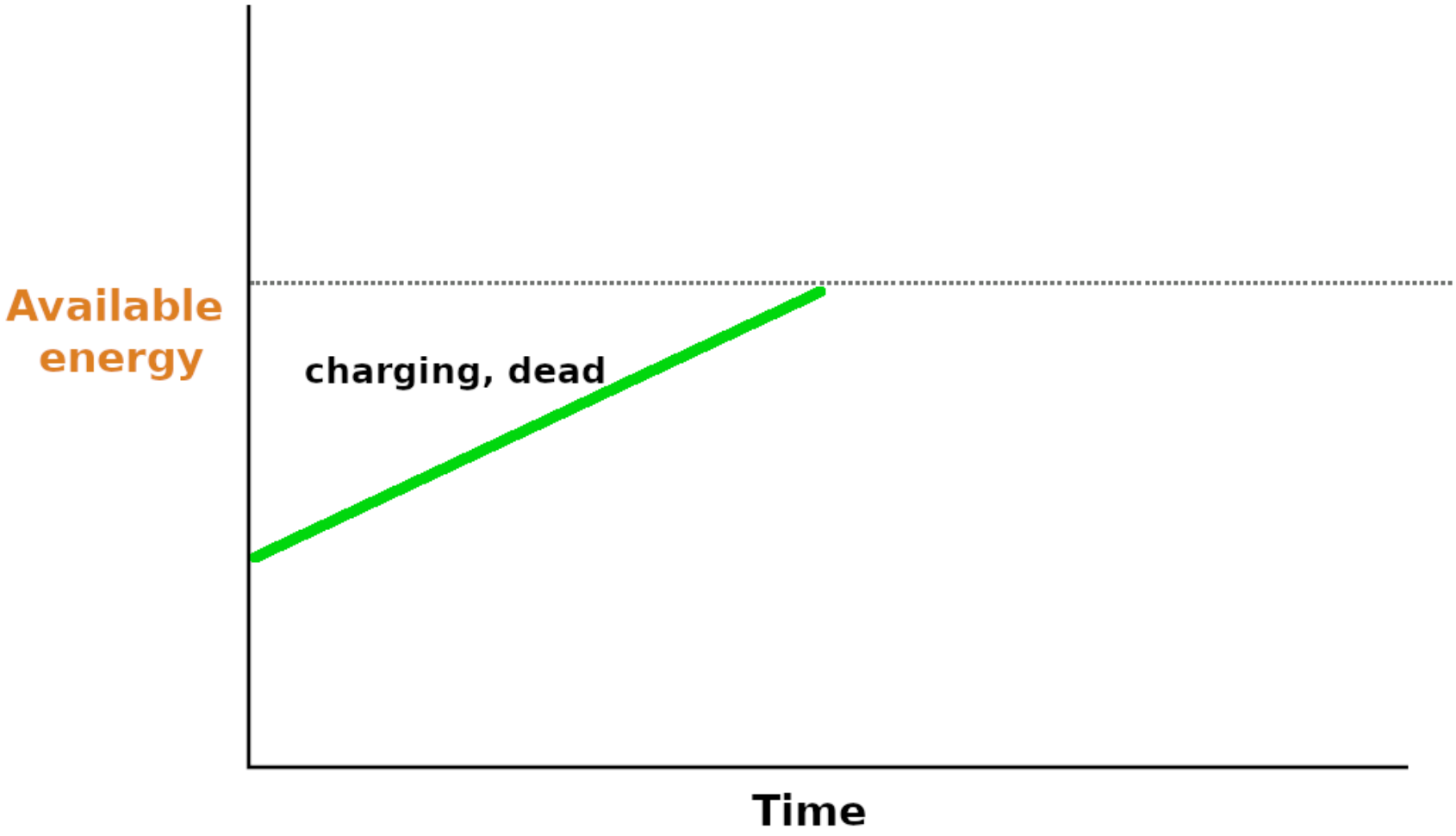
Energy-Harvesting Devices

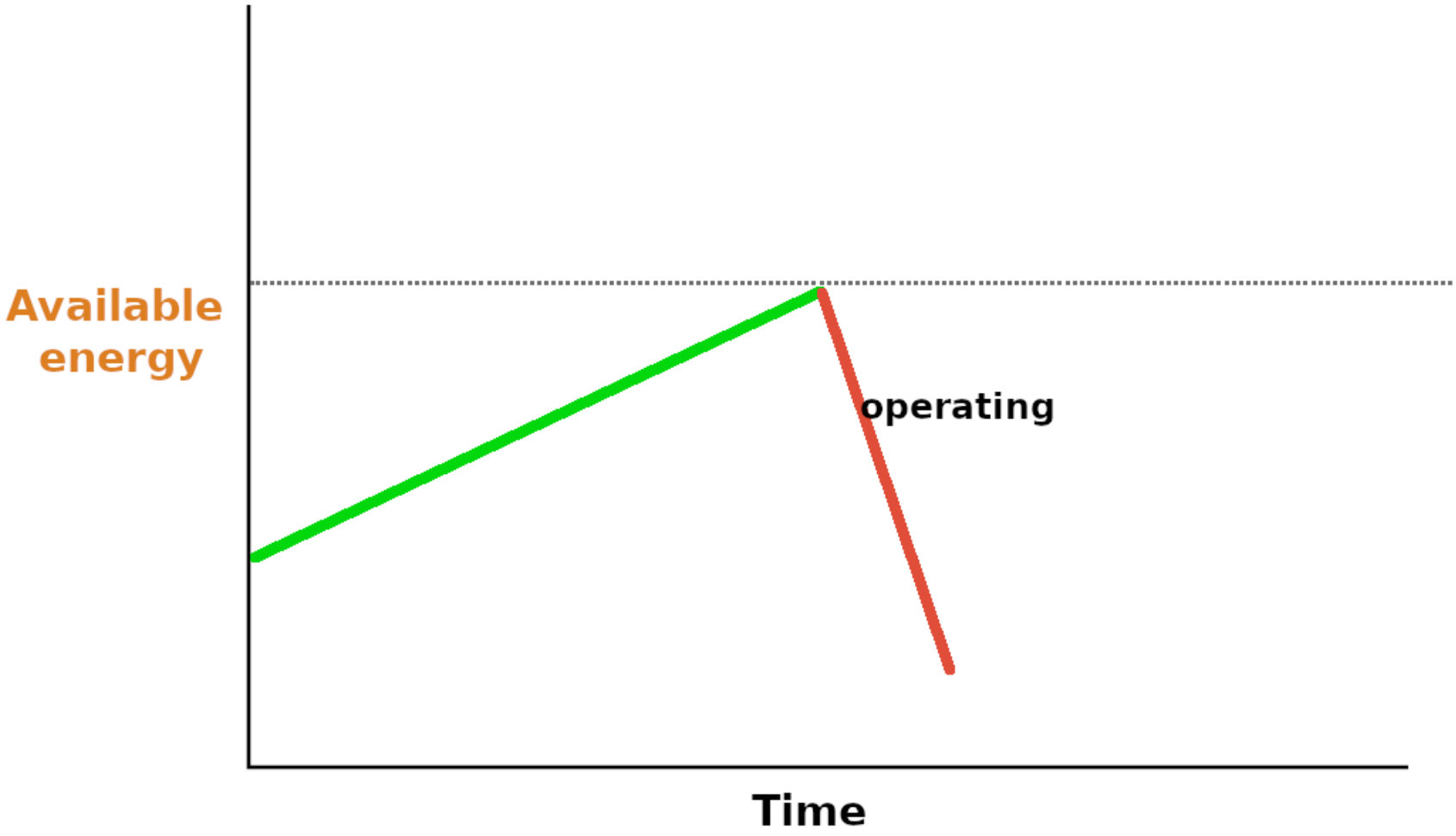


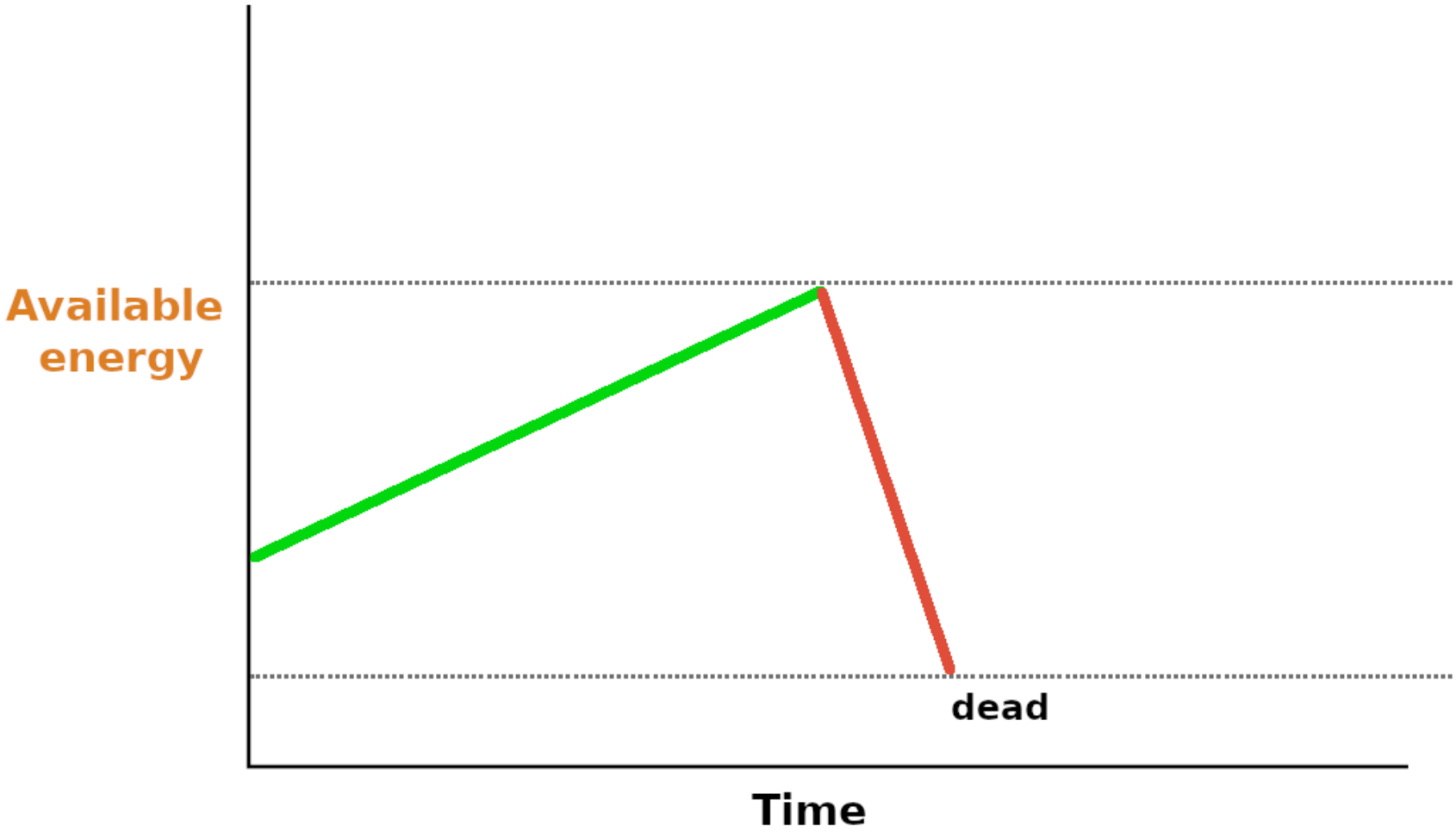
Intermittent Computing

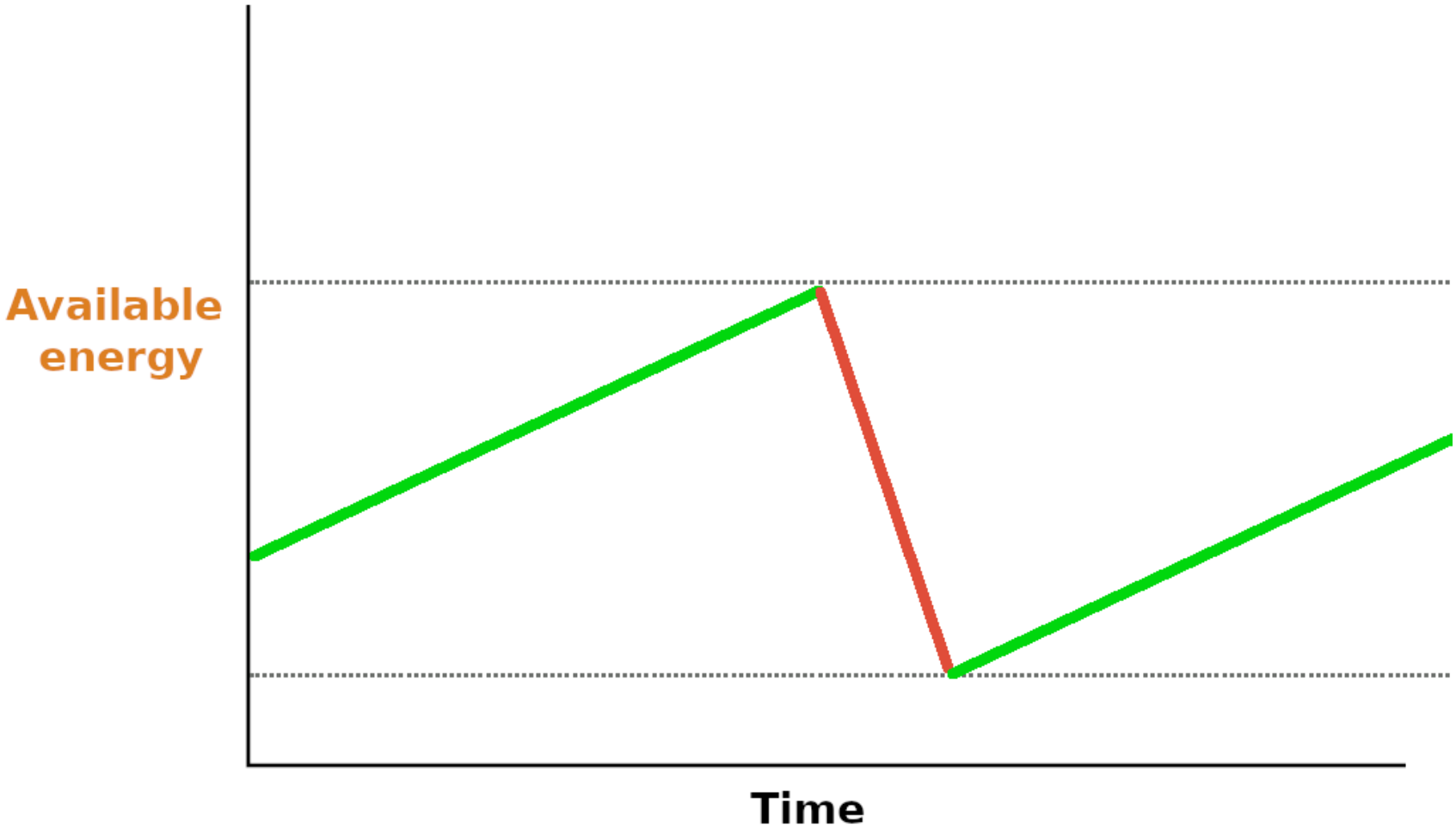

Available energy
Time


Available energy
charging, dead
Time


Available energy
charging, dead
Time


Available energy
operating
Time


Available energy
dead
Time


Available energy
Time

Prior Work

- General idea: periodically save state to non-volatile memory

- General idea: periodically save state to non-volatile memory
- Two main approaches:

- General idea: periodically save state to non-volatile memory
- Two main approaches:
  - Explicit tasks

- General idea: periodically save state to non-volatile memory
- Two main approaches:
  - Explicit tasks
  - Automatic checkpointing

Alpaca

Alpaca - explicit tasks

Problems:

- Programmability overhead

Alpaca - explicit tasks

Problems:

- Programmability overhead
- Not obvious how to estimate energy use of code

Alpaca - explicit tasks

Problems:

- Programmability overhead
- Not obvious how to estimate energy use of code
- Not portable

Ratchet

Ratchet - automatic checkpointing

Problems:

- High execution overhead

Ratchet - automatic checkpointing

Problems:

- High execution overhead
  - Ratchet inserts checkpoints between every Write-After-Read dependency

Ratchet - automatic checkpointing

Problems:

- High execution overhead
  - Ratchet inserts checkpoints between every Write-After-Read dependency
- No task atomicity

Ratchet - automatic checkpointing

Problems:

- High execution overhead
  - Ratchet inserts checkpoints between every Write-After-Read dependency
- No task atomicity
- Might be hard to solve/pin-point non-termination issues

Let's solve all of these!

Chinchilla

Correct, **H**ardware-agnostic **IN**termittent **CH**eckpointing **I**nstrumentation **L**ayer with **L**ow-overhead **A**dapation

Strategy:

- overprovision checkpoints

Strategy:

- overprovision checkpoints
- dynamically deactivate unnecessary ones at run-time

This should give us:

- No programmability overhead

This should give us:

- No programmability overhead
  - Just write C!

This should give us:

- No programmability overhead
  - Just write C!
- Tends towards minimal runtime overhead

This should give us:

- No programmability overhead
  - Just write C!
- Tends towards minimal runtime overhead
- Portable to systems with different energy buffers

Approach

- Instrument code with checkpoints

Approach

- Instrument code with checkpoints
- Start a timer

Approach

- Instrument code with checkpoints
- Start a timer
- Execute checkpointing code only if timer passed a threshold

Approach

- Instrument code with checkpoints
- Start a timer
- Execute checkpointing code only if timer passed a threshold
  - We start with a high threshold

Approach

- Instrument code with checkpoints
- Start a timer
- Execute checkpointing code only if timer passed a threshold
  - We start with a high threshold
- At next reboot, if we hadn't hit a checkpoint, halve the timer threshold

- Once we find an interval that works, increase threshold to median of new and old threholds

- Once we find an interval that works, increase threshold to median of new and old threholds
- Binary search for an optimal interval!

Further timer adaptation

When checkpoints are not frequent enough

- Occasional power failures without a checkpoint are not a bad sign

When checkpoints are not frequent enough

- Occasional power failures without a checkpoint are not a bad sign
  - Means we're close to the actual operating period

When checkpoints are not frequent enough

- Occasional power failures without a checkpoint are not a bad sign
  - Means we're close to the actual operating period
- Failure followed by a successful checkpoint - make a small decrement to timer interval

When checkpoints are not frequent enough

- Occasional power failures without a checkpoint are not a bad sign
  - Means we're close to the actual operating period
- Failure followed by a successful checkpoint - make a small decrement to timer interval
- Many consecutive failures - halve the timer

When checkpoints occur too frequently

- Hit two checkpoints during an interval - double the timer

When checkpoints occur too frequently

- Hit two checkpoints during an interval - double the timer
- Optimization timer:

When checkpoints occur too frequently

- Hit two checkpoints during an interval - double the timer
- Optimization timer:
  - second timer with slightly higher threshold than checkpoint timer

When checkpoints occur too frequently

- Hit two checkpoints during an interval - double the timer
- Optimization timer:
  - second timer with slightly higher threshold than checkpoint timer
  - if the optimization timer expires before a power failure, make small increment to checkpoint timer

Instrumentation Implementation

Checkpoints

- Checkpoints added between each basic block

Checkpoints

- Checkpoints added between each basic block
  - easily identifiable

Checkpoints

- Checkpoints added between each basic block
  - easily identifiable
  - occurs frequently

Checkpoints

- Checkpoints added between each basic block
  - easily identifiable
  - occurs frequently
  - easy to measure block energy cost

Checkpoints

- Checkpoints added between each basic block
  - easily identifiable
  - occurs frequently
  - easy to measure block energy cost
  - easy to subdivide if necessary

Protected variables

- Identify protected variables via liveness analysis

Protected variables

- Identify protected variables via liveness analysis
- If `x`'s live-range crosses a checkpoint, it needs to be protected

Protected variables

- Identify protected variables via liveness analysis
- If `x`'s live-range crosses a checkpoint, it needs to be protected
  - allocated in non-volatile memory

Protected variables

- Identify protected variables via liveness analysis
- If `x`'s live-range crosses a checkpoint, it needs to be protected
  - allocated in non-volatile memory
  - consistency between reboots thanks to undo logging

Protected variables

- Identify protected variables via liveness analysis
- If `x`'s live-range crosses a checkpoint, it needs to be protected
  - allocated in non-volatile memory
  - consistency between reboots thanks to undo logging
    - log before every potential write to a protected variable that may be the variable's first write since a checkpoint

Reinitilization

- On startup, reinitialization is first ran in `main`

Reinitilization

- On startup, reinitialization is first ran in `main`
- Calls programmer-provided `init` to setup peripherals

Reinitilization

- On startup, reinitialization is first ran in `main`
- Calls programmer-provided `init` to setup peripherals
- Restores protected variables from log

Non-volatile stack

- After compilation, some parts of the stack also need to be protected

Non-volatile stack

- After compilation, some parts of the stack also need to be protected
- Store them on a non-volatile stack

Non-volatile stack

- After compilation, some parts of the stack also need to be protected
- Store them on a non-volatile stack
- For consistency, keep track of top and depth pointer

Non-volatile stack

- After compilation, some parts of the stack also need to be protected
- Store them on a non-volatile stack
- For consistency, keep track of top and depth pointer
  - depth is the deepest the stack has popped to since last checkpoint

Non-volatile stack

- After compilation, some parts of the stack also need to be protected
- Store them on a non-volatile stack
- For consistency, keep track of top and depth pointer
  - depth is the deepest the stack has popped to since last checkpoint
  - at next checkpoint, need to back up what's between the two pointers

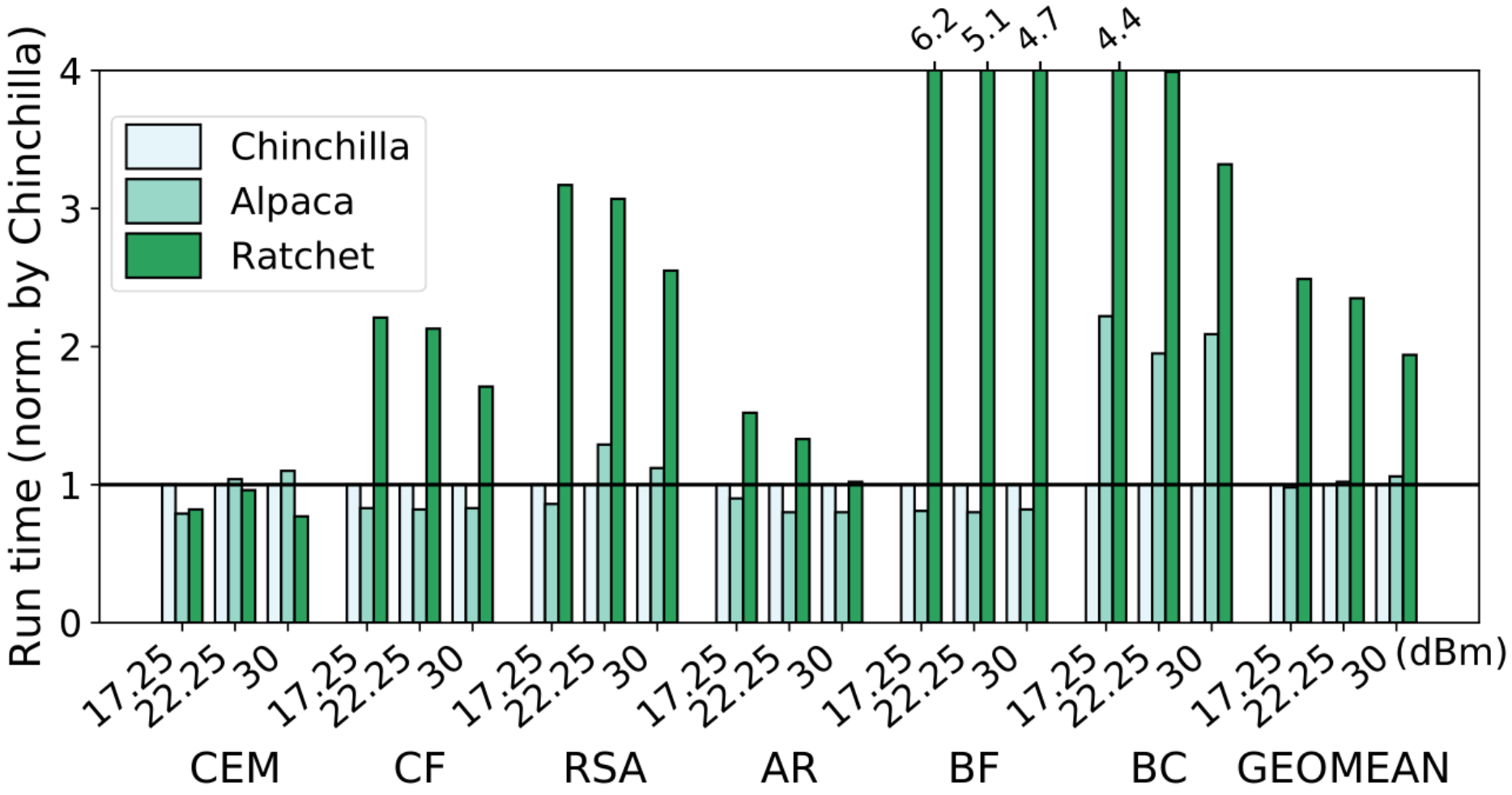
Benchmarks

- Compared to Alpaca and Ratchet

- Compared to Alpaca and Ratchet
- Average speed-up of 2.25x over Ratchet

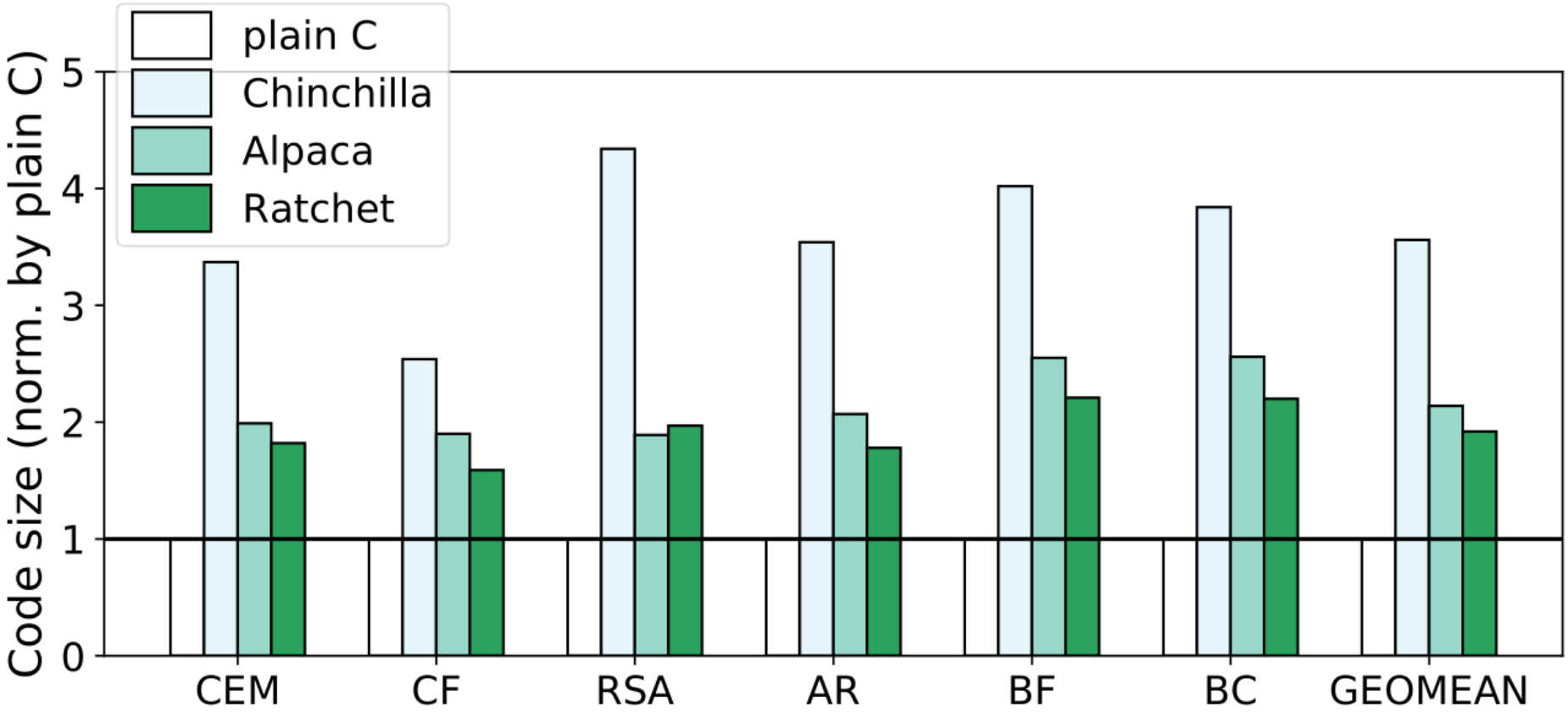
- Compared to Alpaca and Ratchet
- Average speed-up of 2.25x over Ratchet
- Similar perfomance to Alpaca, 2% speedup on average

- Compared to Alpaca and Ratchet
- Average speed-up of 2.25x over Ratchet
- Similar perfomance to Alpaca, 2% speedup on average
- Orders of magnitude less checkpoints than either solution


Run time (norm. by Chinchilla)
0
1
2
3
4
6.2
5.1
4.7
4.4
Chinchilla
Alpaca
Ratchet
17.25
22.25
30
(dBm)
CEM
CF
RSA
AR
BF
BC
GEOMEAN

# Checkpoints	RSA	CEM	BC
Chinchilla	16	30	15
Alpaca	315	1611	710
Ratchet	7643	2319	8907

Chinchilla does end up with larger binary size - cost we pay for performance and reliability.


plain C
Chinchilla
Alpaca
Ratchet
Code size (norm. by plain C)
0
1
2
3
4
5
CEM
CF
RSA
AR
BF
BC
GEOMEAN

Authors

- Kiwan Maeng - PhD student

Authors

- Kiwan Maeng - PhD student
- Brandon Lucia

Authors

- Kiwan Maeng - PhD student
- Brandon Lucia
  - Recommended watching: Reliable Intermittent Computing