Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing

Kiwan Maeng Brandon Lucia
Carnegie Mellon University
{*kmaeng,blucia*}@*andrew.cmu.edu*
http://intermittent.systems

Abstract

Energy-harvesting devices have the potential to be the foundation of emerging, sensor-rich application domains where the use of batteries is infeasible, such as in space and civil infrastructure. Programming on an energy-harvesting device is difficult because the device operates only intermittently, as energy is available. Intermittent operation requires the programmer to reason about energy to understand data consistency and forward progress of their program. Energy varies with input and environment, making intermittent programming difficult. Existing systems for intermittent execution provide an unfamiliar programming abstraction and fail to adapt to energy changes forcing a compromise of either performance or assurance of forward progress.

This paper presents Chinchilla, a compiler and runtime system that allows running unmodified C code efficiently on an energy-harvesting device with little additional programmer effort and no additional hardware support. Chinchilla overprovisions code with checkpoints to assure the system makes progress, even with scarce energy. Chinchilla disables checkpoints dynamically to efficiently adapt to energy conditions. Experiments show that Chinchilla improves programmability, is performant, and makes it simple to statically check the absence of non-termination. Comparing to two systems from prior work, Alpaca and Ratchet, Chinchilla makes progress when Alpaca cannot, and has 125% mean speedup against Ratchet.

1 Introduction

The maturation of energy-harvesting technology and low-power microcontrollers fostered batteryless devices that operate using energy from their environment. *Energy-harvesting devices* operate by collecting and buffering energy in a capacitor, and only *intermittently* executing the software when there is available energy in the capacitor. During execution, a device computes, uses volatile and non-volatile (e.g., FRAM [50]) memory, reads sensors and communicates. Recent work enabled intermittent software execution [30]. Some capture checkpoints [36, 44, 51] automatically at arbitrary points to make progress despite power failures. Other work asks the programmer to decompose code into idempotent, atomic tasks [12, 31, 35] that attempt to execute repeatedly until completing uninterrupted.

While successful enablers of intermittent computing, these prior systems compromise on one or more important system design aspects: performance, programmability, and avoidance of non-termination. Automatic checkpointing approaches [36,44,51] make programming simple, but often incur a high execution time overhead due to excessive checkpoints. Explicit task models [12, 31, 35] require the programmer to adhere to the task-based programming model, hampering programmability. Checkpointing and task-based systems also do not provide a simple way of checking whether the system can encounter non-termination. The code between two consecutive checkpoints or within a task (we refer to both as "task execution") may require more energy to complete than will ever be available in the device's fixed energy buffer. When a task's execution consumes more energy than hardware can buffer, the task will not execute to completion, and the system faces *non-termination* — repeated attempts to execute a task that will never complete. Even a hard reboot does not recover from non-termination, because intermittent operation spans power failures. The only fix is to change the code to use smaller tasks or add more frequent checkpoints and re-flashing the code onto the device.

Avoiding non-termination is an important correctness property in an intermittent system that no prior system satisfactorily provides. Task-based systems [12, 31, 35] complicate programming by asking the programmer to estimate the energy use of code regions and to divide code into arbitrary tasks that execute atomically. Checkpoint-based systems [36, 44, 51] place checkpoint at semantically meaningful points in a program, oblivious to energy consumption between checkpoints. Energy consumption may vary with input and the environment, and static energy modeling [4,7,13] is often imprecise and overly conservative. Without a way of reliably checking a program for non-termination, the burden of writing performant, correct applications lies entirely on the programmer in existing intermittent systems.

In this work, we observe that *adaptation* of checkpoint timing based on energy consumption is the key to achieve the three goals: high performance, accessible programmability, and a simple static check for the absence of non-termination. Based on this observation, we propose Chinchilla, [1] a checkpointing, task-based runtime system that dynamically adapts the interval between checkpoints based on direct observations of program progress. Chinchilla does not ask the programmer to specify task boundaries, making programming simple. Chinchilla statically overprovisions a program with potential checkpoints and makes it simple to check that the span between two potential checkpoints will not exceed the device's energy capacity. Chinchilla achieves high performance by dynamically adapting which potential checkpoints to collect, based on the program's rate of progress, which will vary across platforms and inputs.

We implemented a full prototype of Chinchilla including compiler support, a runtime system, and a non-termination checker. We evaluated Chinchilla running on several real RF-harvesting hardware setups, running a collection of programs from the literature [19, 35] and comparing to two representative systems from prior work, Alpaca and Ratchet [35, 51]. We show that Chinchilla improves programmability, supporting most of the C language (including libraries) and avoids re-engineering code for every new platform. Chinchilla achieves high performance, with a 2.25x average speedup compared to Ratchet, the previous state-of-the-art automatic checkpointing system. Chinchilla achieves performance parity with the task-based Alpaca that is faster than Ratchet, but requires substantial code re-engineering. Additionally, we show that Chinchilla makes it simple to check for the absence of non-termination, providing an assurance that code will run correctly once deployed.

In summary, Chinchilla's main features are:

- A substantial performance improvement compared to state-of-the-art intermittent checkpointing systems.
- Enabling a simple, static check that gives assurance that a program avoids non-termination.
- A simple programming model that supports most of the C language.
- A dynamic run-time checkpointing adaptation mechanism that accommodates varied inputs and environmental conditions.

2 Background

Energy-harvesting devices extract energy from their environment and execute software according to the intermittent execution model, which presents unique challenges that are not present under continuous execution.

[1]**C**orrect, **H**ardware-agnostic **IN**termittent **CH**eckpointing **I**nstrumentation **L**ayer with **L**ow-overhead **A**daptation

2.1 Energy-Harvesting Devices

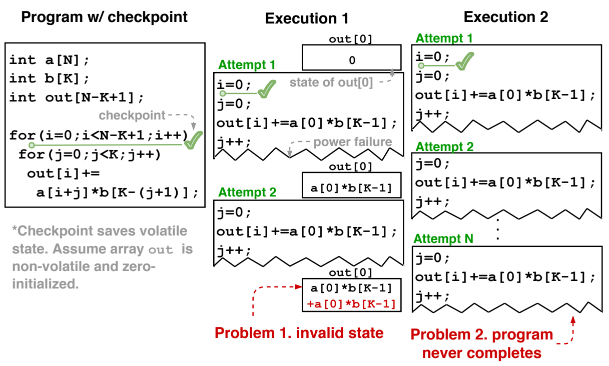
An energy-harvesting device includes a microcontroller (MCU), sensors, volatile and non-volatile memory, and radios. An energy-harvesting device extracts energy from its environment, e.g., radio, vibration, or light, and operates *intermittently*, only when energy is available. A device collects energy into a fixed size energy buffer, usually a capacitor. While the device is inactive, energy slowly accumulates in the buffer. When the energy level in the buffer reaches a defined threshold, the device operates, quickly consuming the buffered energy. The time to accumulate energy is usually greater by orders of magnitude than the time to consume the energy. For example, a WISP with a nearby RF power supply may charge for a second to support 10 ms of operation [14, 47]. At a failure, the device loses the contents of its registers, volatile memory, and peripheral configuration, while retaining the contents of its non-volatile memory (e.g., FRAM [50]).

2.2 Correctness in Intermittent Execution

Software on an energy-harvesting device executes according to the *intermittent execution model*. After a power failure, control resumes from some prior point and execution continues instead of terminating. Key challenges of intermittent execution are: ensuring (1) memory consistency, and (2) forward progress.

Figure 1 shows the challenges of intermittent execution. The figure shows a code for a 1-D convolution that preserves execution progress on each power failure by collecting a volatile execution context (registers, stack) on each outer loop iteration (a model similar to Mementos [44]). The `out` array is allocated in non-volatile memory, initialized to zero. The two executions in the figure show two problematic intermittent execution behaviors. Execution 1 shows that if power fails after updating `out[0]` but without reaching the checkpoint, control flow reverts to the top of the inner loop (`j = 0`) on reboot. However, the partially updated value of `out[0]` persists after the power failure. On reboot, the code updates `out[0]` again, leading to a memory state that is impossible in a continuously-powered execution. Execution 2 shows that if the inner loop's bound, `K`, is sufficiently large, the system will exhaust its energy before reaching the checkpoint, leading to a non-termination.

Several prior strategies successfully ensure memory consistency on intermittent execution, i.e. they solved the problem from Execution 1. However, they show limitations in avoiding non-termination (problem from Execution 2). We discuss how two popular previous approaches — explicit task-based models and automatic checkpointing systems — try to avoid non-termination on intermittent execution and what their limitations are.



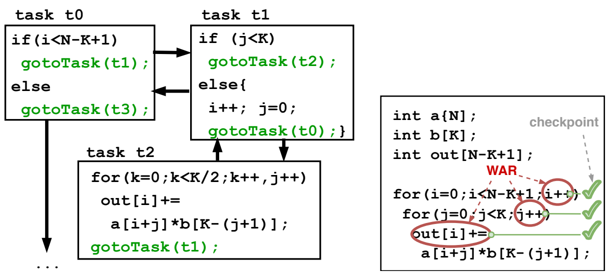
```
int a[N];
int b[K];
int out[N-K+1];
for(i=0;i<N-K+1;i++)
 for(j=0;j<K;j++)
  out[i]+=
   a[i+j]*b[K-(j+1)];
```

```
i=0;
j=0;
out[i]+=a[0]*b[K-1];
j++;
```

```
out[0]
0
a[0]*b[K-1]
a[0]*b[K-1]
+a[0]*b[K-1]
```

```
j=0;
out[i]+=a[0]*b[K-1];
j++;
```

Figure 1: **Challenges of intermittent execution.** Code with volatile state checkpoints may leave memory inconsistent (Execution 1) or never terminate (Execution 2).



```
task t0
if(i<N-K+1)
 gotoTask(t1);
else
 gotoTask(t3);

task t1
if (j<K)
 gotoTask(t2);
else{
 i++; j=0;
 gotoTask(t0);}

task t2
for(k=0;k<K/2;k++,j++)
 out[i]+=
  a[i+j]*b[K-(j+1)];
gotoTask(t1);
...
```

```
int a[N];
int b[K];
int out[N-K+1];
for(i=0;i<N-K+1;i++)
 for(j=0;j<K;j++)
  out[i]+=
   a[i+j]*b[K-(j+1)];
```

(a) Explicit task model

(b) Automatic checkpointing system

Figure 2: **Different systems for the convolution.** A convolution code written in (a) explicit task model (Alpaca [35]), and a code generated by (b) automatic checkpointing system (Ratchet [51]).

Explicit Task Models Explicit task-based intermittent programming and execution models *require* the programmer to explicitly specify task boundaries [12, 31, 35]. In these models, it is solely the programmer's responsibility to avoid non-termination. These models require careful programming, because if a task consumes more energy than the device can buffer, the task will enter non-termination. The programmability cost of specifying task boundaries is high, especially because estimating the energy use of a task for various inputs is difficult.

Automatic Checkpointing Systems Automatic checkpointing systems statically insert a checkpoint at arbitrary program points using compiler and runtime support [36, 44, 51]. Most of these systems insert a large number of checkpoints throughout the binary without considering whether there are sufficiently frequent checkpoints to avoid non-termination. Excessive frequent checkpoints can have high overhead, and there is no easy way in such a system to statically check the presence of non-termination. Additionally, automatic checkpointing systems do not allow the programmer to control over the duration and energy consumption of a task. If task energy demand exceeds the device energy supply, the programmer has no recourse to fix the issue, because checkpoint placement is not part of the programming model. Some propose *ad hoc*, dynamic fallbacks that can have high overhead, and are difficult to characterize [51].

Some recent systems tried to estimate task energy cost and place checkpoints accordingly instead of heuristically. However, prior work showed that precisely estimating the energy cost for an arbitrary code is a challenge even when restricting the model only to the MCU core instead of the full system [9,29]. Models that rely on the instruction counting as a proxy for energy [4] or that use statistical energy models [13] are useful, but limited in precision. To make non-termination checking simple, Chinchilla should have a static check that accounts for full-system power and does not rely on proxy measurements or statistical models.

2.3 Programmability and Performance in Existing Models

In addition to non-termination, prior systems may make programming complex or have poor performance. Figure 2 shows how prior task and checkpointing systems may have programmability and performance issues.

Explicit Task Models Programming with explicit tasks is difficult. Figure 2a shows how the programmer could write a 1-D convolution code in a task-based model [12, 35]. The syntax deviates from plain C and requires the programmer to decide how many loop iterations fit in a task without exceeding the device's energy budget (e.g., task t2 is chosen to hold K/2 loop iterations). A bad choice that puts too many iterations in a task leads to non-termination. A different bad choice that puts too few iterations in a task sacrifices performance. Crucially, if the energy buffer or input changes, the programmer has to *re-write the code* to make tasks differently. Recent platforms support a dynamically variable energy buffer size [14], making the problem more urgent.

Automatic Checkpointing Systems Although automatic checkpoint insertion to the binary incurs low to no additional programming effort, an excess of checkpoints may lead to high execution time overhead. As shown in Figure 2b, Ratchet [51], the state-of-the-art automatic checkpointing system, inserts checkpoint between every Write-After-Read (WAR) dependence, possibly inserting *two* checkpoints on each inner loop iteration and one checkpoint on each outer loop iteration in the example.

If the system's energy buffer can complete multiple iterations of the loop without a power failure, Ratchet suffers an unnecessarily high overhead. Ratchet cannot selectively skip a checkpoint because checkpointing is required by Ratchet's memory consistency model.

2.4 Task Atomicity

An automatic checkpointing system also fails to provide a mechanism for specifying and enforcing application-level atomicity constraints on checkpoint placement. For example, if a program should access two sensors atomically at the same time, they should not be interleaved by a checkpoint. If a checkpoint splits the atomic region, the value collected by the first sensor may be from before a power interruption, and the value collected from the second sensor access may be from much later, after the power interruption, resulting in stale data. Chinchilla allows the programmer to specify such atomicity constraints if an application needs them.

3 System Overview

Chinchilla is a software system that uses a novel, adaptive checkpointing scheme to make software on an intermittently-operating system execute correctly and efficiently. Chinchilla statically overprovisions code with potential checkpoints and dynamically deactivates unnecessary checkpoints at run time to minimize performance overhead. Chinchilla is designed to improve the *programmability* and *efficiency* of intermittent systems, while avoiding *non-termination*. Programming is easy, because Chinchilla inserts checkpoints automatically. Execution is efficient, because Chinchilla's dynamic adaptation mechanism minimizes its checkpointing and state management overhead. Chinchilla exposes a simple, statically-checkable property to determine whether a program will behave correctly on a given platform, allowing Chinchilla to avoid non-termination and effectively making Chinchilla portable to systems with a wide range of energy buffering capacities.

Figure 3 provides an overview of Chinchilla's main features, which are implemented in an instrumenting compiler analysis and software runtime system. Chinchilla compiler inserts checkpointing instrumentation that captures registers and part of the non-volatile data. The compiler also uses static analysis to detect which *protected* data must persist across checkpoints and power failures. Chinchilla's runtime system implements checkpoint and restart, a non-volatile stack to avoid full stack checkpointing, dynamic support for selectively activating checkpoints, and undo logging to ensure consistency.

First, we discuss *where* the compiler inserts checkpoints, second we describe *what* data are checkpointed and logged, and third, we describe *how* Chinchilla selectively activates checkpoints to mitigate overheads.

3.1 Placing Checkpoints to Enable Static Non-Termination Checks

Chinchilla inserts checkpoints into a program that preserve its progress by saving execution context that Chinchilla can restore after a power failure. Chinchilla's goals in placing checkpoints, are to preserve progress and avoid non-termination, and to minimize run time overhead. These goals are in tension. To avoid non-termination, Chinchilla must insert a checkpoint along any program path that consumes more energy than the system can buffer, conservatively checkpointing *as frequently as possible* to avoid non-termination with an arbitrarily small energy buffer. To minimize checkpoint overheads, Chinchilla should only checkpoint at the *boundaries* of a path that consumes more energy than the device can buffer, ideally checkpointing *as infrequently as possible*. Compounding the problem, measuring the full-system energy consumption of arbitrary code is challenging and imprecise [13, 29] because path energy depends on inputs and peripheral state.

Chinchilla escapes the checkpoint placement dilemma by inserting checkpoints conservatively into the program so that the resulting program can be simply assured to avoid non-termination, and selectively disabling checkpoints that are unnecessary to minimize overhead. Chinchilla inserts a checkpoint at each boundary of arbitrarily-defined spans of the program, which we refer to as *checkpoint blocks*. A checkpoint block defines the minimum span of code after which a checkpoint might occur — Chinchilla inserts checkpoints at the boundaries of checkpoint blocks, but not inside a block. If no checkpoint block consumes more energy than the device can buffer, then the program will not suffer non-termination. Given this *block energy sufficiency* premise, eventually every checkpoint block will complete, reaching the next checkpoint, and preserving its progress.

The effectiveness of Chinchilla relies on a well-chosen definition of a checkpoint block. A well-chosen block definition is easily identifiable statically, permits frequent block boundaries, allows easily measuring block energy cost, and yields blocks with low energy variance. Statically identifying block extents is important for statically enumerating all possible program control-flow behavior, especially in the presence of complex I/O. Block boundaries must naturally occur frequently enough in a program, or must be easy to insert arbitrarily frequently to ensure block energy sufficiency, even with a small energy buffer. A block's energy should be easy to measure and have low variance, which precludes any block definition that has unbounded loops or input-dependent control-flow paths with wildly different energy costs.

While many block definitions may fit these requirements, Chinchilla uses the basic block as its checkpoint block definition because it fits the criteria well. Basic blocks are statically defined, frequently occurring, and can be arbitrarily subdivided by a compiler as needed to suit small energy buffers. Basic blocks do not contain branches precluding loops and input-dependent paths, which may vary substantially in energy consumption.

Some multi-basic-block regions of code must be atomic and cannot be spliced by a checkpoint, such as code that reads, processes, and records values from related sensors (cf. Section 2.4). The programmer can annotate such code as an `atomic` block, and Chinchilla will treat it as a single checkpoint block. The programmer must manually ensure that such an `atomic` block meets the criteria of a well-chosen checkpoint block. Annotation of the `atomic` blocks is also a feature of Chinchilla that previous compiler-based systems neglected [44, 51].

Checking for Non-Termination. While Chinchilla compiler itself does not provide a static termination guarantee, it makes *checking* for non-termination simple: if no block's energy consumption exceeds the device's energy buffer, the program avoids non-termination. A programmer can check for non-termination by measuring basic block energy consumption under exhaustive, randomized, or representative inputs. In this work, Chinchilla adapts the CleanCut energy-measuring compiler's block measurement tool [13] to check block energy, exposing the checker directly to the programmer. After the compiler instruments each checkpoint block and the programmer checked that no block's energy consumption leads to non-termination using the checker, the program is *safe*, but *over-provisioned* with checkpoints. Section 3.3 describes how Chinchilla selectively disables checkpoints to avoid excessive overheads.

Limitations of Chinchilla's Assurance of Non-Termination. Even with Chinchilla's compiler and checker, Chinchilla cannot always guarantee the absence of non-termination due to possible variation in energy consumption with variation in input and environment. Despite this limitation, Chinchilla provides two major advantages. First, Chinchilla shifts the scope of reasoning about non-termination from arbitrary inter-checkpoint code regions to a single basic block. Single basic blocks have a lower variance in their energy consumption, simplifying energy measurement and non-termination reasoning [13, 29]. Second, Chinchilla selectively disables unnecessary checkpoints allowing for conservative, static over-provisioning with checkpoints (i.e., on every block). Leveraging these properties Chinchilla provides improved assurance of non-termination (although not a guarantee of its absence in all conditions). Practically, Chinchilla eliminates non-termination. Our evaluation shows that Chinchilla's conservative over-provisioning with checkpoints leaves a 2,100% margin between the device's energy capacity and the highest energy cost of any block; even extreme variation in block energy cost due to inputs or environment is unlikely to exceed such a large margin and cause non-termination (Figure 8).

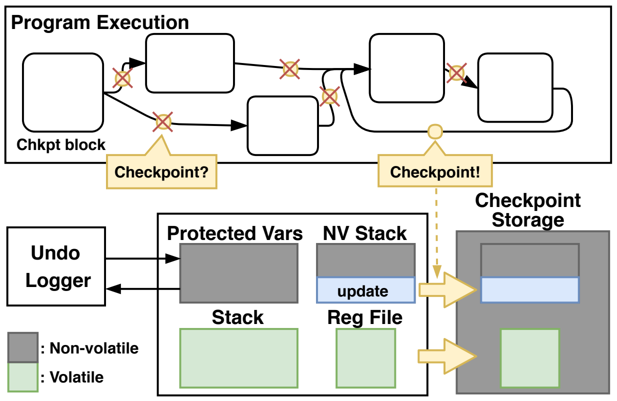


Figure 3: **Overview of Chinchilla.**

3.2 Checkpointing and Undo Logging

Chinchilla checkpoints execution context to preserve and uses undo logging to keep selected, protected non-volatile data consistent across failures.

Checkpointing. Chinchilla checkpoints the execution context, consisting of just the register file and part of the non-volatile data, but not the stack or global data, making the time and energy cost of checkpointing small and predictable. Chinchilla is unlike prior work that uses a fully non-volatile stack (e.g., [25, 45, 51]) to afford register-only checkpointing. Instead, Chinchilla uses an efficient volatile stack and *promotes* a subset of variables to reside in non-volatile memory. Chinchilla only promotes data that may not be re-initialized after a checkpoint to non-volatile memory, leaving all other data on the volatile stack. Chinchilla compiler uses a live-range analysis [3] to identify stack data to promote. If a variable's live range begins after a checkpoint, the variable will be assigned before it is read after a power interruption. Such a variable is safe to leave on the volatile stack without additional protection. The data which need promotion but are not visible to the compiler pass (e.g, data generated by the latter stage of the compiler) is handled by our non-volatile stack discussed below.

Undo Logging. Chinchilla keeps compiler-selected protected, non-volatile variables consistent using undo logging. The key problem, as prior work [12, 25, 31, 35, 43, 51] observed, is that if a non-volatile memory access is involved in a write-after-read (WAR) dependence, then an update to the variable during an execution attempt before a power interruption may incorrectly be visible to a re-executed read after the power interruption.

To prevent code from reading incorrect values, Chinchilla instruments each *write* to a protected variable with undo logging code. At run time, the undo logging code saves to a log the value of the protected variable before the variable's first write after a checkpoint. Chinchilla rolls back updates to protected variables before restarting execution after a power interruption using the log. Section 4.3 describes our undo logging implementation.

Non-Volatile Stack Data. Chinchilla uses a small non-volatile stack to persist stack data that are not visible to Chinchilla compiler pass. These data include return addresses and spilled registers. Compared to the stack, which may be kilobytes, the non-volatile stack is typically small ($\sim$10 bytes) and its elements short-lived. Section 4.2 describes the compiler back-end and runtime system for the non-volatile stack.

3.3 Selective Checkpointing

Checkpointing on every basic block would have a high run time cost that is usually unnecessary because a system is unlikely to fail on every basic block. Chinchilla mitigates the cost of its non-termination-avoiding, conservative, provisioning of checkpoints by skipping some checkpoints at execution time.

Chinchilla sets a timer at startup that, upon its expiration, indicates that Chinchilla should collect the next dynamically executed checkpoint. The runtime skips any checkpoint it encounters while the timer is running, i.e., before it elapses. The key challenge for Chinchilla is identifying a timer duration that expires before the device exhausts its buffered energy (ideally checkpointing before failing), but does not expire too frequently (ideally checkpointing only just before failing).

Chinchilla binary searches for an ideal timer interval at runtime. Chinchilla's search starts by running with a long timer interval. If power fails before the timer expires and Chinchilla collects no checkpoint, the interval is too long; Chinchilla halves the interval and tries again. Assuming that no block consumes more energy than the device can buffer, the timer duration eventually decreases sufficiently to reach a checkpoint.

After finding a sufficiently short interval Chinchilla tries to avoid excessively frequent checkpointing by opportunistically increasing the interval again. Given a *new* shorter interval and the *old* longer interval, Chinchilla increases the interval to a new *median* interval halfway between the new and old intervals. Chinchilla increases its interval to the median interval *only if* execution continues past the new median interval and successfully captures a checkpoint. While non-termination requires immediate interval adjustment, increasing the interval is less urgent. Chinchilla allows the user to decide when in the code to update intervals (e.g., every 100 reboots, each outer loop) by manually annotating a *tuning point*. We put a tuning point on the outermost loop in our benchmarks.

4 Chinchilla Implementation

We implemented a prototype of Chinchilla with four parts: an instrumenting compiler pass and back-end, a runtime library, and a block non-termination checker.

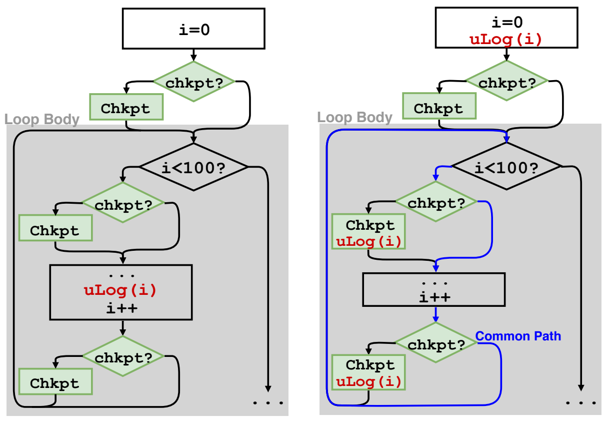
4.1 The Chinchilla Compiler

Chinchilla's compiler transforms C code to use the Chinchilla runtime for safe intermittent execution. The compiler performs five transformations on the code. First, the compiler adds checkpoints at the entry of each basic block. Second, the compiler uses live variable analysis to identify variables that need protection. Third, Chinchilla adds undo logging instrumentation to writes to protected variables. Fourth, the compiler lays out protected variables in memory to efficiently support metadata. Fifth, Chinchilla re-writes `main()` to re-initialize peripherals and roll back the undo logs on reboot.

Checkpoint Instrumentation. The Chinchilla compiler inserts checkpoint code between every pair of basic blocks, except for blocks in explicitly annotated atomic regions. The checkpoint code checks a flag maintained by the Chinchilla runtime that indicates whether the checkpoint interval has elapsed since the last power failure. When the flag is set, the interval has elapsed and the checkpoint code captures a checkpoint.

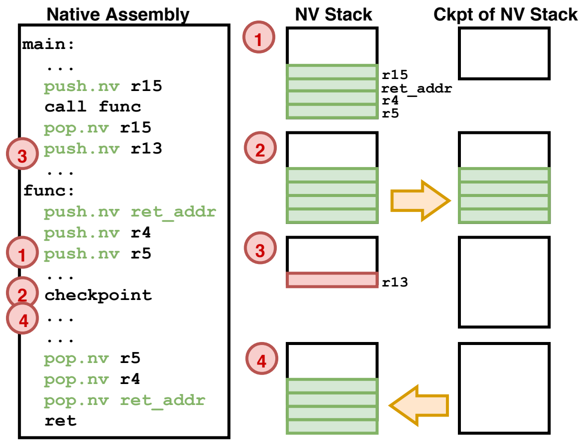
Liveness Analysis and Non-Volatile Promotion. Chinchilla's compiler performs liveness analysis for every variable used in the program to identify protected variables. Variables that are not protected do not need undo logging instrumentation. Chinchilla's liveness analysis calculates the span of code over which a variable may be used without being re-written [3] using a local, context-insensitive, backward CFG traversal from the variable's first use to any definition. Chinchilla leverages LLVM's (conservative) alias analysis: an operation that *may* use a variable starts a live range; only an operation that *must* write to a variable ends its live range. If a variable's live range crosses a checkpoint, the variable is protected and Chinchilla allocates it in non-volatile memory. Chinchilla keeps protected data consistent using undo log.

Undo Logging Instrumentation. Chinchilla's compiler adds undo logging code at accesses to protected variables. Chinchilla inserts a call to `uLog`, which implements undo logging, before every *potential write* to a protected variable that may be the variable's first write since a checkpoint. `uLog` takes the address of the variable as an argument and logs the variable's value before the write executes. Chinchilla uses the log to restore values after a power interruption. The compiler does not instrument accesses to data that do not change throughout the program, such as constant pointers to global arrays. The Chinchilla compiler pre-allocates non-volatile log storage equal in size to the sum of the protected variables' sizes and separated from the protected data store by a



(a) Original tight loop (b) Optimized tight loop

Figure 4: **Tight loop optimization.** (a) Original code and (b) optimized code. In the optimized code, if the system follows the common pass (blue), no undo log (`uLog`) function is called.



```
main:
  ...
  push.nv r15
  call func
  pop.nv r15
  push.nv r13
  ...
func:
  push.nv ret_addr
  push.nv r4
  push.nv r5
  ...
  checkpoint
  ...
  ...
  pop.nv r5
  pop.nv r4
  pop.nv ret_addr
  ret
```

Figure 5: **Non-volatile stack.** The Chinchilla back-end redirects certain `push` and `pop` operation to the non-volatile stack (`push.nv`, `pop.nv`). (1) Some data are pushed to the non-volatile stack. (2) On a checkpoint, only the updated part gets checkpointed. (3) If power fails after non-volatile stack is updated, (4) only the updated part is rolled back.

fixed offset for fast lookups. Chinchilla uses undo logging rather than redo logging (e.g., [35]) because undo logging requires no frequent commit and no instrumentation on read operations. Section 4.3 explains our undo logging implementation.

Memory Layout. Chinchilla organizes protected variables into aligned, fixed-size blocks placed in non-volatile memory with which it associates undo logging metadata; Section 4.3 explains the metadata. Chinchilla uses block metadata, rather than variable- or byte-metadata to amortize its storage overhead. The compiler puts variables smaller than a block in the same block, but disallows a variable to span two blocks. The compiler aligns a variable larger than a block to a block boundary. Chinchilla uses 8 byte blocks, which Section 5 empirically justifies.

Reinitialization. The Chinchilla compiler rewrites the main function to include peripheral reinitialization and log restoration code. The compiler inserts code to restore protected variables from the undo log on reboot. The compiler also inserts a call to a programmer-provided `init` function at the beginning of the main function that reinitializes peripherals on each reboot. Programmers can also perform task-specific re-initialization of the peripherals by setting a non-volatile flag in the task that can be referred in `init`.

Optimized Undo Logging in Tight Loops. Chinchilla's compiler optionally optimizes *tight loops*, which are loops with short bodies that can execute many iterations without exceeding the device's energy buffer. The optimization eliminates per-write undo logging on some variables, instead safely performing undo logging when collecting a checkpoint. Figure 4 shows the optimization applied to the undo logging of `i`. The optimization deletes the undo-logging code in the loop, instead logging (1) in the loop pre-header, and (2) after every checkpoint within the loop (Figure 4b). If the loop checkpoints less than once per iteration (i.e., following Figure 4b's blue path), Chinchilla runs the undo logging function once per checkpoint, rather than once per iteration. If the loop never checkpoints, the undo logging in the pre-header ensures correctness and the optimization never changes program behavior. To avoid optimizing long loops that may checkpoint many times per iteration and lose performance, Chinchilla heuristically selects loops that (1) are inner loops, (2) do not call a function, and (3) have only few (<6) basic blocks in their body.

The optimization idea is different from the static log coalescing from the previous work [4], since it is effectively coalescing the logging function *only when* there is enough energy to run multiple iterations.

4.2 Lightweight Non-Volatile Stack

After compilation, Chinchilla uses a compiler back-end transformation to modify compiled code, to redirect stack accesses that need to be protected but were not visible to Chinchilla's compiler pass, making them refer to Chinchilla's non-volatile stack. These accesses are inserted by the compiler back-end and include saving return addresses, saving and loading caller context, and spilling and reloading registers. Chinchilla must preserve in non-volatile memory the data involved in these accesses when there is a checkpoint between a write and a read of such data.

Chinchilla's back-end replaces these accesses with inlined runtime calls that maintain the non-volatile stack. Chinchilla identifies return address pushes and caller context saves and loads based on the calling convention. Chinchilla identifies register spills and reloads using LLVM IR metadata. We implemented the non-volatile stack transformation in a script that directly modifies assembly; an alternative implementation might modify the LLVM back-end (like Ratchet [51]) at additional implementation effort.

Chinchilla's non-volatile stack has an explicit *top pointer* and a *depth pointer* that tracks the deepest depth to which the top was popped since the most recent checkpoint. Chinchilla uses these pointers to efficiently keep the non-volatile stack consistent across failures. Chinchilla saves with each checkpoint the part of the stack between the top and the depth: the small fraction of the non-volatile stack changed since the last reboot. Clank [25] used a similar differential stack scheme, albeit with architecture support. Figure 5 illustrates the operation of the non-volatile stack management.

4.3 Chinchilla Runtime Library

The Chinchilla runtime library implements adaptive checkpoint collection and restore, undo logging, and non-volatile stack management.

Implementation of `uLog`. Chinchilla's `uLog` function implements undo logging for protected variables. `uLog` takes the address of the variable being accessed as its argument. `uLog` first defensively checks to ensure that it only does undo logging for memory addresses in the range of protected variables, simply returning otherwise. This is necessary because the compiler conservatively inserts undo logging before writes that *may* write to protected variables. Chinchilla compiler omits inserting such defensive check if the accessed data is statically known to be protected.

Chinchilla explicitly tracks whether an access to a variable is its first write since the last checkpoint using an efficient, block-based versioning scheme. Recall that Chinchilla divides memory into blocks of fixed size (Section 4.1). Each block has a one-byte version counter associated with it to track the first write to the block. Chinchilla maintains a global version counter that increments at each collected checkpoint, and at each power interruption. Chinchilla writes the value of the global version counter into a block's version counter each time `uLog` backs up the block (i.e., when a variable contained by the block is written for the first time since a checkpoint.) Chinchilla checks whether a block is in the undo log since the most recent checkpoint by comparing the global version counter to the block's version counter. If a block's version counter is less than the global version counter, the block must be copied to the undo log. Chinchilla clears all blocks' versions when the global version overflows. Figure 6 illustrates how Chinchilla uses versions for undo logging.

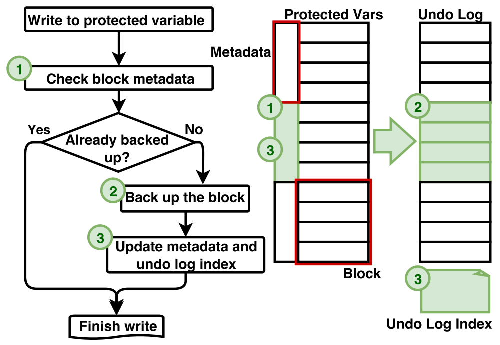


Figure 6: **Undo logging.** Chinchilla (1) checks the metadata and (2) backs up the data before overwriting, (3) updates metadata and the undo log index.

When `uLog` determines that an access is the first write to a variable in a block since the most recent checkpoint, Chinchilla must copy the variable's block to the undo log, which allows restoring the variable's value after a power interruption. A block's undo log location is a fixed offset away from the block; finding the undo log block requires adding the address and offset.

Chinchilla uses an *undo log index* to record backed-up blocks. The index makes it efficient to restore values from the undo log after a power failure, by iterating over the list of index only, not all log storage. Our implementation uses a space-inefficient fixed size index of 1000 entries, although a perfect index with one entry per block of protected variables is also possible. If the fixed-size index were to overflow, it should have the same effect as power failure has; Chinchilla will restart from a checkpoint and subsequently checkpoint more frequently to avoid a second overflow. Chinchilla ensures frequent checkpoints by decreasing its checkpoint timer, which we describe next.

Implementation of Checkpointing. Chinchilla's runtime system implements checkpoint collection and restoration. At a checkpoint, the system backs up the register file, saves the updated part of the non-volatile stack, and clears the undo log index. Chinchilla backs up the register file by saving the contents of registers to a fixed memory region. Chinchilla does not checkpoint the entire non-volatile stack, instead saving the *update* since the last checkpoint, which is contained between the top pointer and depth pointer as discussed in Section 4.2. Chinchilla's checkpoints are double buffered, and if power fails while capturing a checkpoint, Chinchilla reverts to the last successful checkpoint. After a checkpoint, Chinchilla clears its undo log index by resetting the iterator of the index.

Implementation of Restoring. After a power interruption, Chinchilla executes a restore procedure to revert the execution context to the most recent checkpoint before continuing execution. The procedure calls the `init` function, (discussed in Section 4.1) to reconfigure the system's peripherals. It then uses its undo log to revert modified protected variables to their value at the previous checkpoint and restores the changed portion of the non-volatile stack since the previous checkpoint. Finally, the restore routine restores the contents of the register file, restoring the program counter and continuing execution.

If power fails during restoration, Chinchilla continues to try to restore to the same checkpoint after rebooting. Chinchilla keeps its iterator of the undo log in non-volatile memory and during the continuation of the restore procedure Chinchilla can start restoring protected variables from where it left off in the undo log. Assuming the restore procedure successfully reverts at least one protected variable from the undo log, the amount of work in the restore procedure decreases with each attempt. After eventually reverting all entries in the undo log, the only restoration work remaining is to repopulate the register file and continue execution.

Timer Adaptation. Chinchilla uses a *checkpoint timer* to determine when to checkpoint. Chinchilla maintains the timer's interval and configures the timer to count that interval after a reboot. After the timer expires, the next checkpoint call executed collects a checkpoint; before the timer expires, checkpoints do nothing. Chinchilla adjusts the timer's interval during execution to avoid checkpointing too frequently or infrequently. If the interval is too long, power repeatedly fails before the timer expires each time Chinchilla reboots from the last checkpoint. On observing many consecutive failures with no progress, Chinchilla makes a *large decrement* to its checkpoint timer interval by halving its interval. If the checkpoint timer interval is very close to the device's operating period, power may occasionally fail before collecting a checkpoint. On observing a failure before reaching a checkpoint followed by a successfully collected checkpoint (i.e., non-repeated failures), the device makes a *small decrement* to its interval of half the amount of its last change (i.e., small or large).

If the timer interval is very short, Chinchilla may collect checkpoints too frequently. Chinchilla avoids this overhead in two ways. If the checkpoint timer expires twice without a power failure, Chinchilla makes a *large increment* by doubling the timer's interval. Chinchilla increases its checkpoint timer interval using a second *optimization timer* with an interval longer than the checkpoint timer by half the last change in the checkpoint timer's interval. If the optimization timer expires without a power failure and takes a checkpoint, Chinchilla makes a *small increment* to the checkpoint timer interval by assigning the value of the optimization timer interval, and again set the optimization timer interval longer by half the last change in the checkpoint timer's interval.

We implemented both the *checkpoint timer* and the *optimization timer* using a single hardware timer that counts up with two separate handlers. Chinchilla only keeps one *context-insensitive* checkpoint timer interval, assuming the time from boot to power failure is roughly constant regardless of which code is executing. Maintaining different timer intervals may provide a benefit for a system that varies significantly in its operating power (e.g., due to peripheral activity).

4.4 Non-Termination Checker

Our Chinchilla implementation places checkpoints on every basic block, making it simple to statically measure block energy consumption with high precision and ensure the absence of non-termination for a given device's energy buffer. We implemented an energy checker based on CleanCut [13], that measured block energy and compares to device energy automatically.

The checker extracts code between checkpoints from the program's assembly code and generates a measurement binary containing initialization code and the extracted code only. Memory accesses using an unknown reference are redirected to a known location, avoiding referencing invalid memory space as in CleanCut [13]. The checker inserts code that measures energy (i.e., capacitor voltage) at the start and end of the extracted code using EDB [11]. The checker applies this measurement procedure to every basic block in a program and repeatedly executes it multiple times to compensate for measurement noise. If none uses energy that exceeds the device's capacitor energy, the program is unlikely to experience non-termination.

If the checker reveals that a basic block uses more than the capacitor's energy, the block should be subdivided into multiple blocks by the compiler or the programmer. A subdivision is not often necessary: a typical device [47] can run thousands or tens of thousands of instructions before exhausting buffered energy, avoiding non-termination — since our checkpoint blocks are usually small, none of our experiments required subdivision (Section 5.4).

5 Evaluation

We evaluated Chinchilla on *real hardware*, the WISP5 [47] energy harvesting platform equipped with a TI MSP430FR5969 processor. We wirelessly powered the platform using RF energy from the ThingMagic Astra-EX RFID reader at various power levels, placed 20cm apart. We experimented with two WISP hardware configurations: a stock WISP5 (WISP), with the standard 47μF capacitor, and a physically modified WISP5

(WISP-tiny) on which we replaced the standard capacitor with a much smaller 10μF capacitor.

We compared to three previous systems, Alpaca [35], which is the most recent task-based intermittent programming model, Chain [12], another task-based programming model, and Ratchet [51], a compiler-automated approach. Alpaca is a task-based programming and execution model that asks the programmer to write a program as a collection of tasks, complicating programming but leading to high performance by eliminating some inefficiencies of automatic checkpointing systems. Alpaca's tasks have a fixed size and may exceed the device's energy buffer, leading to non-termination and limiting portability. Chain is similar to Alpaca, although with a more complex, channel-based memory model. Ratchet inserts checkpoints automatically while asking nothing of the programmer, but sacrificing performance for this programming simplicity. Ratchet also provides no easy way to check that an inter-checkpoint region will not exceed the device's energy buffer, risking non-termination. Our comparative evaluation shows that Chinchilla's adaptive checkpointing approach is "the best of both worlds," with programmability similar to Ratchet and performance comparable to Alpaca. Moreover, Chinchilla's simple block energy checking procedure allows deploying code with confidence that no block exceeds the device energy buffer, a unique feature that neither Alpaca nor Ratchet provides.

We used the released versions of Alpaca and Chain, directly from the authors. In correspondence with its authors, we ported Ratchet to MSP430 because Ratchet originally targeted ARM only [51]. Our port omits some ARM-specific back-end optimizations from Ratchet, resulting in possibly around 1.6x slowdown on average for our port according to the original work [51].

5.1 Application Benchmarks

We evaluated Chinchilla using all six benchmarks from the Alpaca paper, ported to run on all systems in our setup [35]. The benchmarks are Cold-chain Equipment Monitoring (CEM), Cuckoo Filter (CF), RSA encryption (RSA), Activity Recognition (AR), Bitcount (BC), and Blowfish encryption (BF). For Alpaca, we directly used the benchmarks written by the authors.

CEM reads temperature sensor values and LZW-compresses them. For repeatability, we emulated the sensor with pseudo-random numbers. We used a 512-entry dictionary and a 64-byte compressed block size. CF stores and reads an input data stream using a cuckoo filter with 128 entries. RSA encrypts an eleven character string using RSA with a 64-bit key. AR computes the mean and standard deviation of a window of accelerometer readings to train a nearest neighbor model to detect a shaking movement. We used a window size of three and read 128 samples from each class (shaking or stationary) in the training phase. BF encrypts a given string of length 13 using blowfish encryption. BC counts the number of one bits in a bitstream. For every result presented, we executed the experiment repeatedly, at most more than 200 times if necessary, until the confidence interval converged into less than 10% of the result.

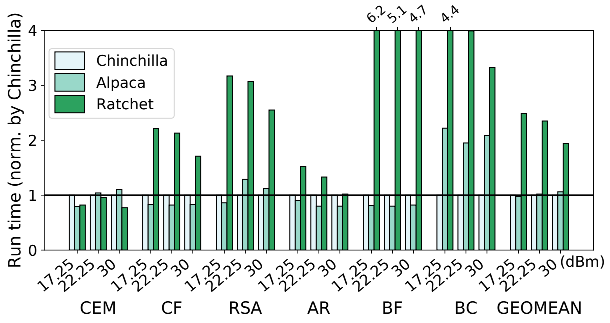


Figure 7: **Run time in different power conditions.**

5.2 Chinchilla is Efficient

Chinchilla ensures a program runs with reasonable overheads in a variety of different wireless power conditions, outperforming state-of-the-art task-based and automatic checkpointing systems in many cases. We measured run time with RF power at 17.25dBm, 22.25dBm, and 30dBm. Figure 7 shows the results.

Chinchilla's run time is faster than the previous programmability-oriented system, Ratchet, in all benchmarks except for CEM, showing an *average speedup around 2.25x*. Even when compared to the performance-oriented Alpaca, Chinchilla shows near-parity performance, with 2% speedup on average. The plot omits data comparing to Chain for brevity; Chinchilla consistently out-performed Chain, with 2.98x average speedup. The main performance benefit of Chinchilla comes from its ability to disable checkpoints, which will be further discussed in Section 5.4.

5.3 Chinchilla is Effectively Portable

Figure 8 shows the energy use of each basic block of Chinchilla in different benchmarks with standard deviation, measured with our checker. We compare block energy to both WISP and the WISP-tiny, shown in the plot as *WISP* and *tiny*. The result from the checker shows that all the benchmarks can run reliably on both platforms, with ample headroom of 2100% (WISP) and 375% (tiny). Thanks to Chinchilla's adaptive checkpointing scheme, this apparent overprovisioning does not impede high performance.

We measured Chinchilla's performance and ability to make progress with different energy buffer sizes (WISP,

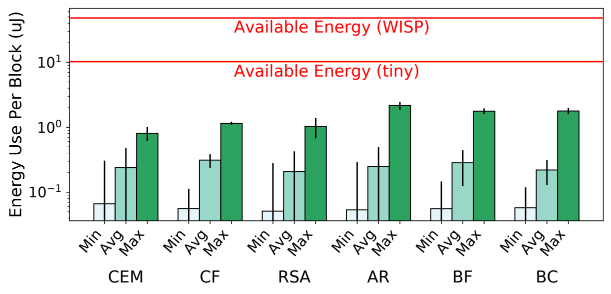


Figure 8: **Energy use of Chinchilla's basic block.** Energy held in WISP (47μF) and tiny (10μF) is also shown.

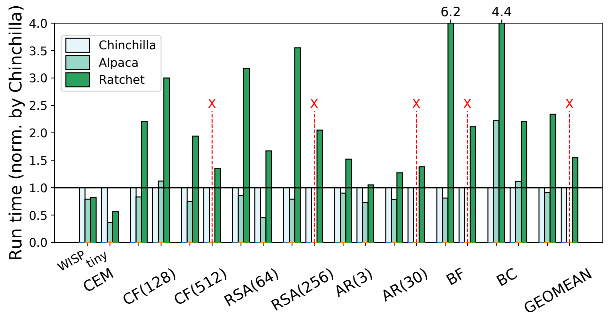


Figure 9: **Execution time in different capacitor size.** Ran on WISP (47μF) and tiny (10μF). A red X indicates the system failed to complete.

tiny), and with varied input sizes. We built variants of CF, RSA, and AR with larger inputs that scale execution time due to an input dependent loop. We increased CF's filter size to 512. We increased RSA's key size to 256. We increased AR's input window size to 30. We performed all tests at 17.25dBm.

The data in Figure 9 show that Chinchilla efficiently operates across a wide range of energy configurations, while Alpaca *fails to complete* in 4 out of 9 cases (CF(512), RSA(256), AR(30), BF) using WISP-tiny.

Ratchet's lack of adaptability makes it slower than Chinchilla across many inputs and energy buffers. Additionally, Ratchet inserts many checkpoints in these applications and never faces non-termination in these data. However, there is no simple way to check that for a different input or hardware configuration Ratchet will avoid non-termination.

5.4 Chinchilla Selectively Checkpoints

We evaluated Chinchilla's ability to adaptively checkpoint only when necessary. We ran complete trials of each application repeatedly and dynamically counted the number of collected checkpoints (Chinchilla, Ratchet) or task transition (Alpaca) and we refer to both as "checkpoint" for brevity.

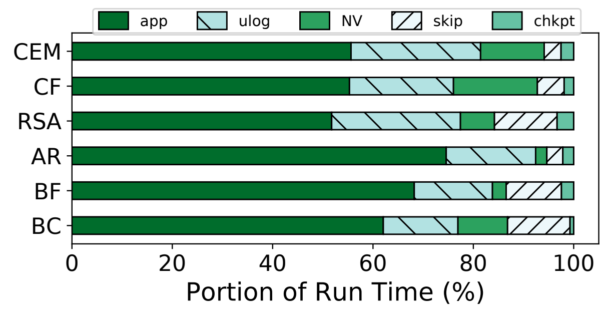


Figure 10: **Overhead breakdown.** Time spent for application code (app), undo log (ulog), non-volatile stack management (NV), skipping disabled checkpoint (skip), and checkpoint and restore (chkpt) is shown.

Table 1 shows the result of the experiment. On average, Alpaca collected 2,185% more checkpoints than Chinchilla and Ratchet collected 21,817% more checkpoints than Chinchilla. The result implies that neither the programmer (Alpaca) nor the compiler (Ratchet) places fixed checkpoints efficiently.

Table 1: **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

We characterize the major overheads of Chinchilla in each app to explain its performance. We measured using a Saleae Digital Logic Analyzer timing GPIO pulses instrumented into code to indicate when different operation types occur. To allow timing instrumentation, we measured overhead on continuous power, emulating power failures using a timer. The major overheads are undo logging (ulog), managing the non-volatile stack (NV), skipping disabled checkpoints (skip), and checkpointing and restoring the checkpoint (chkpt).

Figure 10 shows that undo logging is Chinchilla's major overhead. In contrast, checkpointing and restoring is less than 3.5% of run time across benchmarks, illustrating that Chinchilla avoids unnecessary checkpoints. The result shows that Chinchilla can effectively eliminate checkpointing overhead, which is the major source of performance improvement against Ratchet. However, the additional cost of undo-logging and non-volatile stack management for enabling dynamic checkpointing became the new bottleneck of the system.

5.5 Chinchilla Programming is Simple

Chinchilla makes programming simple by allowing the programmer to use all of C, except for dynamic memory allocation, which is uncommon in embedded code with strict resource constraints. Programming with Chinchilla is simpler than programming with a task-based system. We compare the programmability of Chinchilla against three task-based systems, Alpaca [35], Chain [12], and DINO [31].

There are three aspects of Chinchilla that make it easier to program than task-based models. First, Chinchilla allows using plain C with no special keywords, and Chinchilla's compiler automatically makes code intermittence-safe. Second, Chinchilla allows complex use of pointers, by disambiguating memory references dynamically during undo logging. In comparison, Chain and DINO prohibit pointers to non-volatile memory [12, 31] and Alpaca prohibits some uses of pointers [35]. Third, Chinchilla's block energy checker frees the programmer from reasoning directly about energy consumption while coding. Chinchilla also eliminates the need to rewrite code when hardware or input changes.

Table 2 quantifies programming complexity counting system-specific keywords in our test programs. Chinchilla only requires the programmer to place a checkpoint timer interval tuning function, which our benchmarks do at each outer loop iteration. Chinchilla also allows, but does not require, the programmer to mark atomic regions and none of our applications called for any atomic regions. Compared to other systems, Chinchilla asks very little of the programmer: Alpaca, Chain, and DINO require the programmer to declare system-specific data structures, define tasks, and manually place boundaries and checkpoints.

Table 2: **Summary of programming complexity.**

	App	Chinchilla	Alpaca	Chain	DINO
# Keywords	CEM	1	47	122	13
	CF	1	48	132	11
	RSA	1	67	203	35
	AR	1	45	110	8
	BC	1	49	106	10
	BF	1	42	122	9
Prog. Complexity		Similar to C	High	High	Med
Portability		No Extra Cost	Low	Low	Med
Pointer Support		Always	Med	Low	Low

5.6 Metadata Block Size

As Section 4.1 describes, Chinchilla associates undo logging metadata with a block of data. A larger block size has a lower metadata storage overhead, but incurs a higher run time undo logging cost because the undo log moves data at block granularity. We measured the storage and run time overhead for different block sizes. We omit full data due to space constraints, but we experimentally determined that when the metadata overhead is 12.5% or more (i.e., eight-byte blocks), time overhead is low. Larger blocks had higher run time overhead and we use eight-byte blocks. We also found that few variables (1%–4%) could be kept in SRAM and most were promoted to non-volatile memory because most lifetimes spanned a checkpoint.

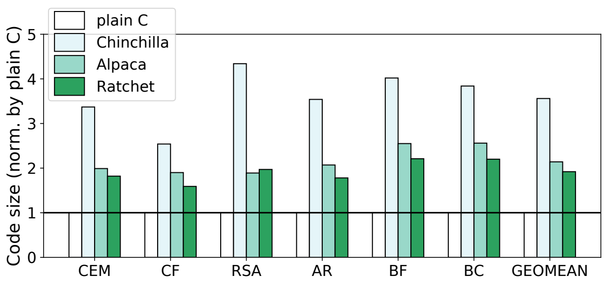


Figure 11: **Compiled code size of different systems.**

5.7 Code Size Increase

Chinchilla inserts checkpointing code between *every* basic block, increasing code size. Figure 11 shows the normalized code size, measured by directly inspecting compiled binaries. Note that while the plain C code is smallest, it does not run correctly on intermittent energy.

All intermittent computing systems see a code size increase due to instrumentation and libraries. Chinchilla has a 3.56x code size increase compared to plain C, 1.66x compared to Alpaca, and 1.86x compared to Ratchet. The code size increase is the cost Chinchilla pays for its performance and reliability benefits. While increased code size may increase instruction cache miss rate, Section 5.2 shows that Chinchilla has higher performance than prior systems regardless of any potential increase.

5.8 Alternate Checkpointing Heuristic

We studied an alternative to Chinchilla's timer-based checkpoint disabling heuristic that decides whether to collect a checkpoint based on whether the checkpoint was used to restore in the recent past. If execution never resumes from a checkpoint, the checkpoint is unlikely to be useful and should be disabled.

We implemented this alternative *history-based* checkpoint disabling heuristic that disables checkpoints that were collected but not used for a fixed period of the execution. The system stores a *score* for each checkpoint that indicates its likely usefulness. On power failure, the system updates a checkpoints' scores, incrementing the score of the checkpoint used for restoration, and decrementing the scores of checkpoints collected and not used. Periodically, the system disables checkpoints with a score below a threshold.

Figure 12 compares the performance of Chinchilla and Chinchilla reimplemented to use this alternative heuristic. We observed that the history-based heuristic was sometimes comparable to Chinchilla's approach, but suffered performance degradation for some bench-

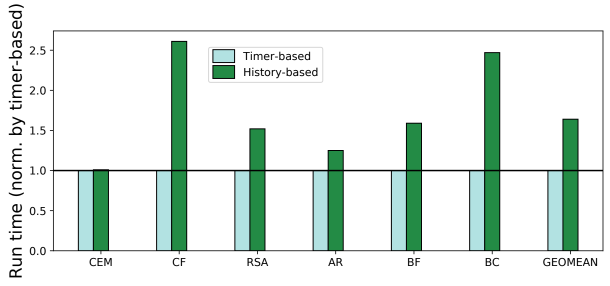


Figure 12: **Run time of two different heuristics.** Run time of timer-based checkpoint disabling versus history-based checkpoint disabling.

marks. The heuristic does especially poorly with functions called from multiple different calling contexts because the score associated with a checkpoint is *context-insensitive*. The timer-based heuristic was more consistent and simpler to implement. We ultimately exclude the history-based heuristic from Chinchilla's design.

6 Related Work

Various prior work influenced the design of Chinchilla. The most related work is on intermittent computation on energy-harvesting devices. Work on maintaining non-volatile memory consistency and approaches that leverage undo logging to maintain consistent execution of the program, such as transactional memory, is related as well. Our work is also related to the prior work that tried to estimate energy use of an arbitrary code.

Intermittent Execution Prior work [36, 44, 51] preserves progress with automatically inserted checkpoints of the execution context. Automatic checkpointing often insert redundant checkpoints, impeding performance. Ensuring progress or atomicity with these techniques is complex because they insert checkpoints arbitrarily. Some systems estimate code energy cost to place checkpoints [4,7,13], but estimating energy in arbitrary code is difficult and error-prone [13]. Task systems ask the programmer to place task boundaries [12,31,35], requiring the programmer to form tasks that do not consume too much energy. Mayfly [23] adds real-time constraints on I/O processing to a task system. Forcing the programmer to define tasks complicates programming and offers no simple way to ensure non-termination. Moreover, these models preclude some C features. Chinchilla eliminates programming complexity and allows most of C. Some systems checkpoint "on demand" [5, 6, 26, 45] by monitoring supply voltage. These avoid unnecessary checkpoints, but require extra hardware, and require complex tuning of a checkpoint trigger threshold; a bad threshold risks failing to checkpoint. Non-volatile processors (NVP) [34] change the architecture to save state. Incidental computing [33] and NEOFog [32] optimize the NVP for latency insensitive code and fog computing. Clank [25] implements undo-logging in microarchitecture. Capybara [14] adds a flexible energy storage capacitor, meeting varied energy demand. UFoP [21] assigns a capacitor for each peripheral, and Flicker [22] assists rapid prototyping of an energy-harvesting device. TARDIS and CusTARD [24] keeps time with low power on an energy-harvesting device. Chinchilla requires no architecture or hardware support.

Abundant prior work addressed low-energy embedded systems, but not explicitly intermittent execution. Tock [28] is an OS with multi-tenancy for low-power systems [2]. Dewdrop [8] supports energy-harvesting, but not intermittent execution. Eon [48] allows specifying how tasks of different cost should be scheduled as energy conditions change. ZebraNet [27] used energy-harvesting devices to track wildlife, but with large batteries and solar panels, not intermittent operation. Other work addresses deep neural networks on an energy-harvesting devices [18]. Some work helps develop intermittent code. Wisent [49] and Stork [1] update software on intermittent hardware. Ekho [54] and EDB [11] helps with testing on energy-harvesting devices.

Non-Volatile Memory Consistency Prior work on memory persistency in powered systems support consistency in mixed-volatility memory with access reordering [41, 42, 55]. Others support consistent, non-volatile data structure and file systems [10, 15, 16, 17, 39, 40, 52, 53]. Transactions and transactional memory systems [20, 37, 38, 46] also support consistency and persistence. Chinchilla also supports non-volatile memory consistency (i.e., persistency), but unlike prior work, does so for intermittently powered devices. The rate of failures and constraints on energy and resources faced by Chinchilla makes adopting these solutions difficult.

Energy Measurement CleanCut [13] estimates energy cost of arbitrary code, to aid in checkpoint placement. Other work [4, 7] estimates energy use of code by looking at instruction or cycle count. Both have limitations in precisely estimating the energy use correctly. Chinchilla avoids the problem by confining energy measurement to a basic block. Other works outside the domain of energy-harvesting also tried estimating energy use of code by using evolutionary modeling [29] or by the number of active gates [9]. However, these approaches only estimate the energy use of a processor core, while Chinchilla checker checks the energy of the entire platform.

7 Conclusion

Chinchilla is a fully-automatic, adaptive system that enables correct intermittent execution without additional programming complexity. Automatic compilation and undo logging enables writing unmodified C code. dynamic checkpoint adaptation offers portability across platforms, inputs, and environments without recompilation. Chinchilla brings its benefits with low run time cost compared to the state of the art, with an average 2% speedup compared to Alpaca, and a 125% speedup over Ratchet. Chinchilla is the first system to simplify programmability using adaptive checkpoints, and provide strong static assurance of progress without the aid of specialized hardware.

Acknowledgements

Thanks to the anonymous reviewers for the insightful feedback and to Alexei Colin, Emily Ruppel, and Adwait Dongare for the valuable discussion and feedback about the work. We are also grateful to Cristiano Giuffrida for shepherding our final draft. This work is based on work supported by National Science Foundation grant CSR-1526342 and CAREER Award CCF-1751029. Kiwan Maeng was supported by a scholarship from the Korea Foundation for Advanced Studies.

References

[1] AANTJES, H., MAJID, A. Y., PAWEŁCZAK, P., TAN, J., PARKS, A., AND SMITH, J. R. Fast downstream to many (computational) rfids. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE* (2017), IEEE, pp. 1–9.

[2] ADKINS, J., CAMPBELL, B., GHENA, B., JACKSON, N., PANNUTO, P., AND DUTTA, P. The signpost network: Demo abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (New York, NY, USA, 2016), SenSys '16, ACM, pp. 320–321.

[3] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: principles, techniques, and tools*, vol. 2. Addison-wesley Reading, 2007.

[4] BAGHSORKHI, S. S., AND MARGIOLAS, C. Automating efficient variable-grained resiliency for low-power iot systems. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (2018), ACM, pp. 38–49.

[5] BALSAMO, D., WEDDELL, A. S., DAS, A., ARREOLA, A. R., BRUNELLI, D., AL-HASHIMI, B. M., MERRETT, G. V., AND BENINI, L. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35*, 12 (2016), 1968–1980.

[6] BALSAMO, D., WEDDELL, A. S., MERRETT, G. V., AL-HASHIMI, B. M., BRUNELLI, D., AND BENINI, L. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters 7*, 1 (2015), 15–18.

[7] BHATTI, N. A., AND MOTTOLA, L. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks* (2017), ACM, pp. 209–219.

[8] BUETTNER, M., GREENSTEIN, B., AND WETHERALL, D. Dewdrop: An energy-aware runtime for computational rfid. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 197–210.

[9] CHERUPALLI, H., DUWE, H., YE, W., KUMAR, R., AND SARTORI, J. Determining application-specific peak power and energy requirements for ultra-low power processors. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ACM, pp. 3–16.

[10] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 105–118.

[11] COLIN, A., HARVEY, G., LUCIA, B., AND SAMPLE, A. P. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 577–589.

[12] COLIN, A., AND LUCIA, B. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2016), OOPSLA 2016, ACM, pp. 514–530.

[13] COLIN, A., AND LUCIA, B. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction* (2018), ACM, pp. 116–127.

[14] COLIN, A., RUPPEL, E., AND LUCIA, B. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM.

[15] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[16] DOSHI, K., AND VARMAN, P. Wrap: Managing byte-addressable persistent memory. In *Memory Archiecture and Organization Workshop.(MeAOW)* (2012).

[17] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 15.

[18] GOBIESKI, G., BECKMANN, N., AND LUCIA, B. Intermittent deep neural network inference.

[19] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on* (2001), IEEE, pp. 3–14.

[20] HERLIHY, M., AND MOSS, J. E. B. *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[21] HESTER, J., SITANAYAH, L., AND SORBER, J. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2015), SenSys '15, ACM, pp. 5–16.

[22] HESTER, J., AND SORBER, J. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems* (2017), ACM, p. 19.

[23] Hester, J., Storer, K., and Sorber, J. Timely execution on intermittently powered batteryless sensors. In *Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2017), SenSys 2017, ACM.

[24] Hester, J., Tobias, N., Rahmati, A., Sitanayah, L., Holcomb, D., Fu, K., Burleson, W. P., and Sorber, J. Persistent clocks for batteryless sensing devices. *ACM Trans. Embed. Comput. Syst. 15*, 4 (Aug. 2016), 77:1–77:28.

[25] Hicks, M. Clank: Architectural support for intermittent computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ACM, pp. 228–240.

[26] Jayakumar, H., Raha, A., Stevens, J. R., and Raghunathan, V. Energy-aware memory mapping for hybrid fram-sram mcus in intermittently-powered iot devices. *ACM Trans. Embed. Comput. Syst. 16*, 3 (Apr. 2017), 65:1–65:23.

[27] Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L. S., and Rubenstein, D. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ASPLOS X, ACM, pp. 96–107.

[28] Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Pannuto, P., Dutta, P., and Levis, P. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 234–251.

[29] Liqat, U., Bankovic, Z., Lopez-Garcia, P., and Hermenegildo, M. V. Inferring energy bounds statically by evolutionary analysis of basic blocks. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2016)* (2016).

[30] Lucia, B., Balaji, V., Colin, A., Maeng, K., and Ruppel, E. Intermittent computing: Challenges and opportunities. In *LIPIcs-Leibniz International Proceedings in Informatics* (2017), vol. 71, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[31] Lucia, B., and Ransford, B. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI 2015, ACM, pp. 575–585.

[32] Ma, K., Li, X., Kandemir, M. T., Sampson, J., Narayanan, V., Li, J., Wu, T., Wang, Z., Liu, Y., and Xie, Y. Neofog: Nonvolatility-exploiting optimizations for fog computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), ACM, pp. 782–796.

[33] Ma, K., Li, X., Li, J., Liu, Y., Xie, Y., Sampson, J., Kandemir, M. T., and Narayanan, V. Incidental computing on iot nonvolatile processors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, ACM, pp. 204–218.

[34] Ma, K., Zheng, Y., Li, S., Swaminathan, K., Li, X., Liu, Y., Sampson, J., Xie, Y., and Narayanan, V. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (2015), IEEE, pp. 526–537.

[35] Maeng, K., Colin, A., and Lucia, B. Alpaca: Intermittent execution without checkpoints. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2017), OOPSLA 2017, ACM.

[36] Mirhoseini, A., Songhori, E. M., and Koushanfar, F. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered asics. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on* (2013), IEEE, pp. 216–224.

[37] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS) 17*, 1 (1992), 94–162.

[38] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., Wood, D. A., et al. Logtm: log-based transactional memory. In *HPCA* (2006), vol. 6, pp. 254–265.

[39] Moraru, I., Andersen, D. G., Kaminsky, M., Tolia, N., Ranganathan, P., and Binkert, N. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (2013), ACM, p. 1.

[40] Narayanan, D., and Hodson, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 401–410.

[41] Pelley, S., Chen, P. M., and Wenisch, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 265–276.

[42] Pelley, S., Chen, P. M., and Wenisch, T. F. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro 35*, 3 (2015), 125–131.

[43] Ransford, B., and Lucia, B. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness* (New York, NY, USA, 2014), MSPC '14, ACM, pp. 5:1–5:3.

[44] Ransford, B., Sorber, J., and Fu, K. Mementos: System support for long-running computation on rfid-scale devices. 159–170.

[45] Ransford, B., Sorber, J., and Fu, K. Mementos: System support for long-running computation on rfid-scale devices. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 159–170.

[46] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2006), PPoPP '06, ACM, pp. 187–197.

[47] Sample, A. P., Yeager, D. J., Powledge, P. S., Mamishev, A. V., and Smith, J. R. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement 57*, 11 (2008), 2608–2615.

[48] Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2007), SenSys '07, ACM, pp. 161–174.

[49] Tan, J., Pawełczak, P., Parks, A., and Smith, J. R. Wisent: Robust downstream communication and storage for computational rfids. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE* (2016), IEEE, pp. 1–9.

[50] TI Inc. Products for msp430frxx fram. `http://www.ti.com/lsds/ti/microcontrollers-16-bit-32-bit/msp/ultra-low-power/msp430frxx-fram/products.page`, 2017. Accessed: 2017-04-08.

[51] Van Der Woude, J., and Hicks, M. Intermittent computation without hardware support or programmer intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 17–32.

[52] Venkataraman, S., Tolia, N., Ranganathan, P., and Campbell, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (Berkeley, CA, USA, 2011), FAST'11, USENIX Association, pp. 5–5.

[53] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 91–104.

[54] Zhang, H., Salajegheh, M., Fu, K., and Sorber, J. Ekho: Bridging the gap between simulation and reality in tiny energy-harvesting sensors. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems* (New York, NY, USA, 2011), HotPower '11, ACM, pp. 9:1–9:5.

[55] Zhao, J., Li, S., Yoon, D. H., Xie, Y., and Jouppi, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2013), MICRO-46, ACM, pp. 421–432.