Alpaca: Intermittent Execution without Checkpoints

KIWAN MAENG, Carnegie Mellon University, USA
ALEXEI COLIN, Carnegie Mellon University, USA
BRANDON LUCIA, Carnegie Mellon University, USA

The emergence of energy harvesting devices creates the potential for batteryless sensing and computing devices. Such devices operate only intermittently, as energy is available, presenting a number of challenges for software developers. Programmers face a complex design space requiring reasoning about energy, memory consistency, and forward progress. This paper introduces Alpaca, a low-overhead programming model for intermittent computing on energy-harvesting devices. Alpaca programs are composed of a sequence of user-defined tasks. The Alpaca runtime preserves execution progress at the granularity of a task. The key insight in Alpaca is the *privatization* of data shared between tasks. Shared values written in a task are detected using idempotence analysis and copied into a buffer private to the task. At the end of the task, modified values from the private buffer are atomically committed to main memory, ensuring that data remain consistent despite power failures. Alpaca provides a familiar programming interface, a highly efficient runtime model, and places few restrictions on a target device's hardware. We implemented a prototype of Alpaca as an extension to C with an LLVM compiler pass. We evaluated Alpaca, and directly compared to two systems from prior work. Alpaca eliminates checkpoints, which improves performance up to 15x, and Alpaca avoids static multi-versioning, which improves memory consumption by up to 5.5x.

CCS Concepts: • **Computer systems organization → Embedded software**; *Reliability*;

Additional Key Words and Phrases: energy-harvesting, intermittent computing

ACM Reference Format:
Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (October 2017), 30 pages. https://doi.org/10.1145/3133920

1 INTRODUCTION

The emergence of extremely energy-efficient processor architectures creates the potential for computing and sensing systems that operate entirely using energy extracted from their environment. Such *energy-harvesting* systems can use energy from radio waves [Sample et al. 2008; Zhang et al. 2011a], solar energy [Lee et al. 2012; Zac Manchester 2015], and other environmental sources. An energy-harvesting system operates only *intermittently* when energy is available in the environment and experiences a power failure otherwise. To operate, a device slowly buffers energy into a storage element (e.g., a capacitor). Once sufficient energy accumulates, the device begins operating

Authors' addresses: Kiwan Maeng, Electrical and Computer Engineering Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, US; Alexei Colin, Electrical and Computer Engineering Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, US; Brandon Lucia, Electrical and Computer Engineering Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, US.
Authors' addresses: Kiwan Maeng, Electrical and Computer Engineering Department, Carnegie Mellon University, USA, kmaeng@andrew.cmu.edu; Alexei Colin, Electrical and Computer Engineering Department, Carnegie Mellon University, USA, acolin@andrew.cmu.edu; Brandon Lucia, Electrical and Computer Engineering Department, Carnegie Mellon University, USA, blucia@andrew.cmu.edu.

2475-1421/2017/10-ART96
https://doi.org/10.1145/3133920

and quickly consumes the stored energy. Energy depletes more quickly during operation (e.g. milliseconds) than it accumulates during charging (e.g., seconds). When energy is depleted and the device powers off, volatile state, e.g. registers and stack memory, is lost, while non-volatile state, e.g., ferroelectric memory (FRAM), persists. The charge/discharge cycle of an energy-harvesting device forces software to execute according to the *intermittent execution model* [Colin and Lucia 2016; Lucia and Ransford 2015; Van Der Woude and Hicks 2016]. An intermittent execution includes periods of activity perforated by power failures. The key distinction between intermittent execution and continuously-powered execution is that in the intermittent model a computation may execute only partially before power fails and must be resumed after the power is restored. Correct and efficient intermittent execution requires a system to meet a set of correctness requirements (C1-3) and performance goals (G1-3).

C1: A program must preserve progress despite losing volatile state on power failures.
C2: A program must have a consistent view of its state across volatile and non-volatile memory.
C3: A program must respect high-level atomicity constraints (e.g., sampling related sensors together).

G1: Applications should place as few restrictions on the processor and memory hardware architecture as possible.
G2: Applications should be tunable at design time to use the energy storage capacity efficiently.
G3: Applications should minimize runtime overhead, memory footprint, restore overhead, and re-executed code.

Recent work made progress toward several of these goals. Volatile checkpointing approaches [Balsamo et al. 2016, 2015; Mirhoseini et al. 2013; Ransford et al. 2011a,b] ensure progress (C1). Static multi-versioning [Colin and Lucia 2016] and dynamic versioning [Lucia and Ransford 2015; Van Der Woude and Hicks 2016] systems selectively create copies of modified data to keep volatile and non-volatile state consistent (C2). Task-based systems [Colin and Lucia 2016; Lucia and Ransford 2015] allow programmers to express application-level atomicity constraints (C3) and match the energy cost of a task to the device energy capacity (G2).

However, prior approaches necessarily compromise on some of the goals. Volatile-only checkpointing does not ensure that volatile and non-volatile data remain consistent (C2). Static and dynamic versioning systems have high overheads in time [Lucia and Ransford 2015] or space [Colin and Lucia 2016] (G3). Idempotent compilation [Van Der Woude and Hicks 2016] reduces overhead (G3), but applies only to devices where all memory is non-volatile, not to off-the-shelf microcontrollers with hybrid volatile and non-volatile memory on the market today (G1). The fully-automatic nature of idempotent compilation deprives the programmer of control over the energy cost of the resulting tasks (G2) and complicates forcing tasks to respect high-level atomicity constraints (C3).

This paper develops Alpaca [1], a programming and execution model that allows software to execute intermittently. Like state-of-the-art systems, Alpaca preserves progress despite power failures (C1) and ensures memory consistency (C2). Alpaca uses a *static task model* without checkpoints and explicit restoration of volatile state. Alpaca's approach to maintaining memory consistency despite power failures is conceptually similar to transactional memory (TM) [Shavit and Touitou 1995] with redo-logging. An Alpaca task manipulates privatized copies of data and commits updates to those data atomically when the task completes. By discarding privatized copies, Alpaca can restart a task with negligible cost, similarly to Chain [Colin and Lucia 2016], but without the memory overhead of static multi-versioning (G3). Alpaca dynamically versions non-volatile data, like DINO [Lucia and Ransford 2015], but without checkpointing volatile state (G3). Alpaca selectively versions non-volatile memory locations identified by a compiler-based idempotence analysis, like

[1] Alpaca: Adaptive Lightweight Programming Abstraction for Consistency and Atomicity

Ratchet [Van Der Woude and Hicks 2016], but without volatile state checkpoints (G3) and without being limited to hardware with only non-volatile memory (G1). Alpaca's design differences, relative to state-of-the-art systems, translate into performance gains of 4.2x on average (up 15x in some cases) and a non-volatile memory footprint smaller by 1.3-5.5x. In contrast to a fully automated compiler-only approach [Van Der Woude and Hicks 2016] and just-in-time dynamic checkpointing approach [Balsamo et al. 2016, 2015; Mirhoseini et al. 2013; Ransford et al. 2011a,b], Alpaca's task model allows the programmer to control where tasks begin and end. With this control, Alpaca applications can satisfy application-level atomicity requirements (C3) and use tasks sized to device energy capacity (G2).

Section 2 provides background on intermittent computing. Sections 3 and 4 describe the Alpaca programming model and its implementation. Section 5 discusses key design decisions. Sections 6 and 7 describe our benchmarks and evaluation. We conclude with a discussion of related (Section 8) and future (Section 9) work.

2 BACKGROUND AND MOTIVATION

Energy-harvesting systems operate intermittently, losing power frequently and unexpectedly. Intermittent operation compromises forward progress and leads to inconsistent device and memory states, with unintuitive consequences that demand abstraction by new programming models.

2.1 Energy-Harvesting Devices and Intermittent Operation

Energy-harvesting devices operate using energy extracted from their environment, such as solar power [Lee et al. 2012; Zac Manchester 2015], radio waves (RF) [Sample et al. 2008], or mechanical interaction [Karagozler et al. 2013; Paradiso and Feldmeier 2001]. As the processor on such a device executes software to interact with sensors and actuators or communicate via radio, it manipulates both volatile and non-volatile memory. An energy-harvesting device can operate only intermittently, when energy is available. Common energy-harvesting platforms [Sample et al. 2008] use a power system that charges a capacitor slowly to a threshold voltage. At the threshold, the device begins operating, draining the capacitor's stored energy much more quickly than it can recharge. The system eventually depletes the capacitor, and the device turns off and waits to again recharge to its operating voltage. These power cycles can occur frequently: RF-powered devices may reboot hundreds of times per second [Sample et al. 2008].

2.2 Device Model and Hardware Assumptions

Our work makes few assumptions about device hardware. A device's memory system can include an arbitrary mixture of volatile and non-volatile memory, unlike prior work that requires all memory to be non-volatile [Ma et al. 2015b; Ransford et al. 2011a; Van Der Woude and Hicks 2016]. Alpaca works on devices with non-volatile memories that support atomic read and write operations, e.g. Ferroelectric RAM [TI Inc. 2017b] and Flash. In commercially available FRAM implementations that rely on destructive reads (i.e., rewrite-on-read), access atomicity is satisfied by means of an internal capacitor that buffers sufficient energy to complete an in-progress access. Our model allows arbitrary peripheral (I/O) devices as detailed in Section 5.

2.3 Intermittent Execution and Memory Consistency

Software on an energy-harvesting device operates *intermittently*: an intermittent execution does not *end* when power fails; instead the execution alternates between active periods and inactive periods. On each power failure, the register file and volatile memory (i.e., stack and globals) are erased. Variables in non-volatile memory persist. Prior work [Balsamo et al. 2016, 2015; Mirhoseini et al. 2013; Ransford et al. 2011a,b] checkpoints volatile state periodically and restores a checkpoint

after a power failure. Other prior work [Colin and Lucia 2016; Lucia and Ransford 2015; Van Der Woude and Hicks 2016] found that if an application directly manipulates non-volatile memory, checkpointing only the volatile state is not enough to guarantee consistency. The problem exists because some memory operations may repeat after restarting from a checkpoint. Non-volatile state written before a power failure persists after a restart, and if re-executing code reads the non-volatile state without first over-writing it, the code may operate using inconsistent values. The resulting program behavior is impossible if the device were powered continuously. Precisely, a non-volatile value that may be read and later written (i.e., a "write-after-read", or *W-A-R*) between two consecutive checkpoints can become inconsistent [De Kruijf and Sankaralingam 2013; Lucia and Ransford 2015; Van Der Woude and Hicks 2016].

Figure 1 illustrates how the combination of a *W-A-R* dependence and volatile-only checkpointing can leave data inconsistent. The code, excerpted from our implementation of RSA [Rivest et al. 1978], multiplies two numbers `in1` and `in2` digit by digit, accounting for carries. The `NV` prefix denotes non-volatile data. The code preserves per-digit progress using non-volatile variables `d`, `carry`, and `prod[]`, the output digit index, most recent carry value, and output product. In the execution, `carry` is updated, power fails, and after restarting, `mult()` uses the already-updated value of `carry`, producing the wrong result (Figure 1b). The code first reads, then writes `carry` (a *W-A-R*), putting it at risk of inconsistency. While the figure shows a problem with `carry` only, `d` is also read, then written, presenting another potential consistency problem.

2.4 Quantifying the Overhead of Existing Approaches

Intermittent programming systems that handle volatile and non-volatile memory consistency preserve progress across power failure by either taking checkpoints [Van Der Woude and Hicks 2016] or bounding tasks [Colin and Lucia 2016; Lucia and Ransford 2015]. We use an example to analyze the overhead incurred by systems in each of these two categories. Figure 2 shows a program that manipulates two arrays, `A` and `B`, of size `N`. A task boundary or a checkpoint is denoted uniformly by `TaskBoundaryOrCheckpoint()`. Existing systems can execute this program correctly but not without considerable overhead, which we quantify in Table 1, and restrictions on the memory layout.

Coarse, volatile-only checkpointing, e.g. Mementos [Ransford et al. 2011b], Idetic [Mirhoseini et al. 2013], copies registers and the stack to non-volatile memory. The arrays must reside in volatile memory to ensure they are part of the checkpoint . Each checkpoint will copy both `A` and `B` to non-volatile memory, and each restore operation will copy both arrays from non-volatile back to volatile memory. The copy takes place even if only part of the array is being manipulated. Most copies of array `B` are especially wasteful because `B` is only written after the third checkpoint – until then the system wastefully copies uninitialized values.

```
NV int in1[], in2[], prod[], carry, d;
void bigint_mult(){
  d = 0; carry = 0;
  while (d < NUM_DIGITS){
    TaskBoundaryOrCheckpoint();
    (p,c) = mult(in1[d], in2[d], carry);
    carry = c;
    prod[d++] = p;
  }
}
```

(a) Sample code from RSA.

```
...
(p,c) = mult(in1[0], in2[0], carry);
carry = c;
prod[0] = p;
TaskBoundaryOrCheckpoint();
(p,c) = mult(in1[1], in2[1], carry);
carry = c;

TaskBoundaryOrCheckpoint();
(p,c) = mult(in1[1], in2[1], carry);
carry = c;
prod[1] = p;
...
```

(b) Intermittent execution.

Fig. 1. RSA code with intermittent execution.

Volatile checkpointing with non-volatile versioning, e.g. DINO [Lucia and Ransford 2015], statically inserts code to selectively make a copy of, or *version* non-volatile data that may become inconsistent before it is overwritten. Guaranteeing consistency for non-volatile data allows storing A and B in non-volatile memory. Allocating the arrays in non-volatile memory reduces the amount of copied data to the subset of values that are at risk of inconsistency (e.g., B never needs copying because it is never read). However, a copy must still be made of *all* elements of A, because the updated part of the array is dependent on the inputs to `update()`, which are not known at compile time. Versioning all of A is wasteful when only a small array slice is updated, i.e. when `(end - begin)` is small.

```
void update(int begin, int end){
  int A[N], B[N], i;
  for (i=begin; i<end; i++){A[i]=0;}
  TaskBoundaryOrCheckpoint();
  for (i=begin; i<end; i++){A[i]=A[i]+1;}
  TaskBoundaryOrCheckpoint();
  for (i=begin; i<end; i++){A[i]=A[i]*2;}
  TaskBoundaryOrCheckpoint();
  for (i=begin; i<end; i++){B[i]=A[i]-2;}
  TaskBoundaryOrCheckpoint();
  ...
}
```

Fig. 2. An application example with arrays.

Static multi-versioning, e.g. Chain [Colin and Lucia 2016], has negligible checkpointing or restoring overhead, at the cost of allocating multiple non-volatile copies of the data. Chain allocates memory for a copy of a data structure (i.e., a "channel") per each pair of tasks that share that data structure. In our example, Chain allocates three channels each with a copy of array A – one copy for each task boundary across which A is shared. Since B is not shared, no channel is allocated to it. In addition to the memory overhead, on some channel accesses, Chain incurs runtime overhead to select the channel that contains the most recently modified value. Furthermore, Chain application code burdens the programmer with marshaling data through channels explicitly.

Compiler-automated checkpointing approaches, e.g., Ratchet [Van Der Woude and Hicks 2016], Mementos [Ransford et al. 2011b], are limited by their static analyses. For example, Ratchet inserts a checkpoint at every potential idempotence violation, i.e. every *W-A-R* dependence, which may be many more checkpoints than required to preserve progress, given the device's energy buffer size. In our example, Ratchet may execute as many as 6N checkpoints (2N for `A[i]` and 4N for `i`). Similarly, Mementos inserts a checkpoint on every loop in "loop-latch mode", potentially executing up to 4N checkpoints, which may be an unnecessarily large number.

Ratchet absorbs the high cost of frequent checkpoints by restricting the program to allocate *all its state in non-volatile memory*. Under this strict assumption, the only volatile state that needs to be saved in each checkpoint is the register file. However, this assumption is limiting in several ways. An access to non-volatile memory uses more energy and more time compared to access to volatile memory. The difference in energy ranges from 2-2.5x, in our experiments on TI MSP430FR5969 microcontroller, to 5x implied by worst-case specifications in the device datasheet[TI Inc. 2017a]. Latency differs by 5-8x [Takashima et al. 1999] , which increases the number of cycles for memory access at frequencies above 8MHz on MSP430FR5969. Low-power microcontrollers suitable for energy-harvesting applications available on the market today, e.g.

Table 1. Overhead of prior approaches.

System	# Ckpts.	Ckpt. Size	NV Mem.
Mementos	~4N*	A + B + RF	A + B + RF
DINO	4	A + RF	A + RF
Chain	4	PC	2A + PC
Ratchet†	~6N	RF	RF

RF = all registers; PC = program counter register
* Conditional checkpoints. † Assumes all memory non-volatile.

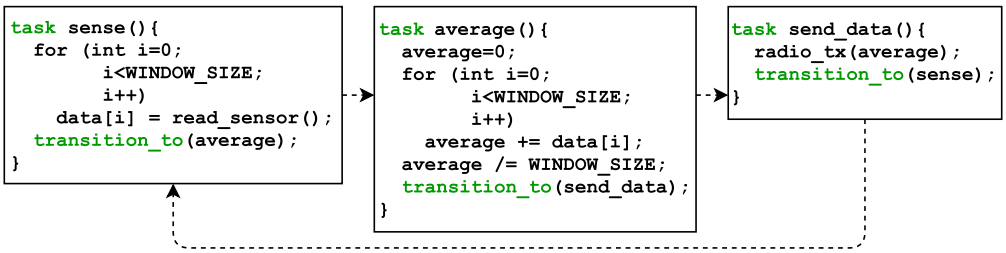


Fig. 3. An application written in Alpaca. The program samples a sensor, calculates an average, and transmits via radio.

[Sample et al. 2008; TI Inc. 2017b], offer a hybrid memory with volatile and non-volatile regions, which would be underutilized due to this restriction imposed by the software system.

3 ALPACA PROGRAMMING MODEL

Alpaca is a programming interface that allows programmers to write software that behaves correctly under an intermittent execution model. Alpaca aims to overcome the limitations of prior work described in Section 2 and to meet design requirements C1–C3 and design optimization goals G1–G3 from Section 1. The Alpaca programming model consists of two core concepts, *tasks* and *privatization*. A task is a programming abstraction that is useful for preserving progress, implementing atomicity constraints, and controlling an application's energy requirements. Privatization is a language feature that guarantees to the programmer that any volatile or non-volatile memory accessed by a task remains consistent, regardless of power conditions.

3.1 Task-Based Programming

A task in Alpaca is a user-defined region of code that executes on a consistent snapshot of memory and produces a consistent set of outputs. An Alpaca task that eventually has sufficient energy to execute to completion is guaranteed to have behavior (i.e., control-flow and memory reads and writes) that is equivalent to some continuously-powered execution regardless of arbitrarily-timed power failures. As Section 4 describes, if power fails during a task's execution, Alpaca effectively discards intermediate results and execution starts again from the beginning of the task. Consequently, a programmer can reason as though tasks are atomic, like transactions in a TM system. Computations that consume more energy than the hardware can provide between two consecutive power failures must be split into multiple tasks.

To program in Alpaca, the programmer decomposes application code into tasks, each marked with the `task` keyword. Each task explicitly transfers control to another task (or to itself). A program's control flow is defined by the execution of tasks in the sequence specified by the transfer statements. To transfer control from a task to one of its successors, the programmer uses the `transition_to` keyword, which takes the name of a task as its argument and immediately jumps to the beginning of that task. `transition_to` statements are valid along any control-flow path within a task, and all paths through a task must end in a `transition_to` statement or program termination. The programmer specifies which task should run on when the system powers on for the first time using `entry` keyword. Entire list of keywords Alpaca introduces is listed in Section 3.3. Figure 3 shows a sensing application written using Alpaca. Alpaca tasks are syntactically similar to Chain tasks [Colin and Lucia 2016], but the memory model for task interactions differs completely.

Alpaca guarantees to the programmer that a task executes *atomically* even if power fails during its execution. When the task completes and the next task begins, changes to memory made by the completed task are guaranteed to be visible and control never flows backward to the completed task again, unless an explicit `transition_to` statement executes. Conversely, if a task does not complete due to a power failure, control does not advance to any other task, which prevents the partially updated state from becoming visible. Alpaca allows only a single task sequence and does not support parallel task execution. This design choice is reasonable because parallel hardware is extremely rare in intermittent devices due to its relatively high power consumption. Alpaca does not support concurrent (i.e., interleaved as threads) task sequencing. Concurrency is limited to I/O routines only, which are addressed in Section 5.3.

Task atomicity guarantees correctness by ensuring that if any of a task execution's effects become visible, then all of them are visible, and by ensuring that a completed task's execution takes effect only once. Moreover, task-based execution *preserves progress*, assuming that eventually the system buffers sufficient energy to complete any task. Alpaca's atomicity property derives from its memory model and data privatization mechanism.

3.2 Alpaca Memory Model and Data Privatization

Alpaca's memory model provides a familiar programming interface allowing tasks to share data via global variables. At the same time, the memory model design allows an efficient implementation of the task-atomicity guarantee. The Alpaca memory access model divides data into *task-shared* and *task-local* data. Multiple tasks or multiple different executions of the same task may share data using task-shared variables. Task-shared variables are named in the global scope and are allocated in non-volatile memory. Task-shared variables have a typical load/store interface: once a task wrote a value to a task-shared variable, that same task or another task may later read the value by referencing the variable name. Task-local variables are scoped only to a single task, must be initialized by that task, and are allocated in the efficient volatile memory.

As discussed in Section 2.3, directly manipulating non-volatile memory in an intermittent execution can leave data inconsistent due to *W-A-R* dependencies. To prevent these inconsistencies, Alpaca *privatizes* task-shared variables to a task during compilation. Privatization creates a task-local copy of a task-shared variable in a *privatization buffer*. As the task executes, it manipulates the copies in the privatization buffer. When the task completes it copies data to a *commit list* that the task uses to atomically commit all updates buffered in the privatization buffer. Section 4.2 describes how privatization works and why it is sufficient to keep data consistent. We emphasize, however, that from the programmer's perspective, privatization is invisible. To support our privatization analysis, the programmer need only specify (1) tasks and (2) task-shared variables. With this information alone, Alpaca provides its consistency guarantee automatically and efficiently.

3.3 Summary of Alpaca Syntax and Semantics

To summarize, Alpaca introduces five new syntactic elements to a C-like base language: `task`, `transition_to`, `TS`, `entry`, and `init`.

- `task` identifies a function as an Alpaca task.
- `transition_to` ends one task and redirects control-flow to another task, specified by name or reference as an argument to `transition_to`. Recall that tasks cannot be called directly.
- `TS` identifies a variable as task-shared, which locates the variable in non-volatile memory and exposes it to Alpaca's *W-A-R* dependence and privatization analysis.
- `entry` is a qualifier on a single task declaration that identifies the *entry task*, which executes when the device boots for the first time.

Table 2. Summary of privatization semantics of Alpaca.

Premise	Statement	Semantics
$x \in V, x \notin \{TS \cap WAR[t]\}, r \in R$	$r := x$	$r \leftarrow M[x]$
$x \in V, x \notin \{TS \cap WAR[t]\}, r \in R$	$x := r$	$M[x] \leftarrow r$
$x \in V, x \in \{TS \cap WAR[t]\}, B[x] = \texttt{undef}, r \in R$	$r := x$	$B[x] \leftarrow M[x]$; $r \leftarrow B[x]$
$x \in V, x \in \{TS \cap WAR[t]\}, B[x] \neq \texttt{undef}, r \in R$	$r := x$	$r \leftarrow B[x]$
$x \in V, x \in \{TS \cap WAR[t]\}, r \in R$	$x := r$	$B[x] \leftarrow r$
$t, t2 \in T$	`transition_to` $t2$ from t	$\forall x \in WAR[t] : M[x] \leftarrow B[x]$; $pc \leftarrow t2$

- `init` is a function qualifier that identifies an *init function*, which executes first on every reboot, to reinitialize hardware (e.g., sensors, radios) and interrupt handlers.

Table 2 shows an informal overview of the memory and task transition semantics of the Alpaca language. Section 4 discusses Alpaca's implementation in detail. In Table 2, T is a set of tasks, V is the set of all variables, TS is the set of task-shared variables, R is the set of registers, $WAR[t]$ is the set of variables involved in *W-A-R* dependences within a task t, B is a privatization buffer, $B[x]$ is x's entry in the privatization buffer, M is memory, $M[x]$ is x's location in memory, pc is the program counter, and `undef` is the value of an uninitialized variable. In task $t \in T$, a read or write to x, a task-shared variable ($x \in TS$) involved in a *W-A-R* dependence ($x \in WAR[t]$), is redirected to the privatization buffer (row 3, 4, 5); otherwise the read or write has typical load/store semantics (row 1, 2). Task $t \in T$ privatizes a task-shared, *W-A-R* variable $x \in (TS \cap WAR[t])$ to $B[x]$ before or at the first read or write to x in t (row 3, 5) and all accesses to x in t are redirected to $B[x]$ (row 3, 4, 5). On completing, t atomically commits each variable x in the privatization buffer $B[x]$ to its original location $M[x]$ and jump to $t2$(row 6).

These privatization semantics reflect our Alpaca prototype's *redo-logging* approach, in which tasks manipulate privatized copies and commit them at the end of a task. We observe, however, that Alpaca is not fundamentally tied to redo-logging, and could instead use an *undo-logging* approach that buffers variables' original values, directly manipulates memory, and restores variables' values after a failure. We leave an undo-logging formulation of Alpaca as future work.

4 ALPACA IMPLEMENTATION

Our prototype implements the programming model defined in Section 3 using a compiler analysis and a runtime library. The key requirements for an Alpaca implementation are (1) preserving progress at the granularity of tasks, (2) ensuring that task-shared and task-local data are consistent, and (3) doing so efficiently.

To meet these requirements, our Alpaca implementation uses two techniques. The first technique is *data privatization*, which ensures that data remain consistent by transparently copying selected values into temporary buffers and redirecting the task's accesses to the buffer. The second technique is *two-phase commit*, which both preserves progress and guarantees that a completed task's updates to its privatized values are all rendered consistently in memory. Alpaca's use of *task-based execution* is the foundation of its efficient support for privatization and two-phase commit.

4.1 Task-Based Execution

Alpaca tasks are void functions with arbitrary code identified with the `task` keyword. Alpaca maintains a global `cur_task` pointer in non-volatile memory that records the address of the task

that began executing at the last successful task transition. Alpaca also maintains a global non-volatile 16-bit counter, `cur_version`, which is initially 1, is incremented on each reboot or task transition, and is reset to 1 when it reaches its maximum value. The counter is used to privatize arrays efficiently (Section 4.4). To transition from one task to the next at a `transition_to` statement, Alpaca assigns `cur_task` to the address of the next task and jumps to the start of that task. When task execution resumes after a power failure, control transfers to the start of `cur_task`.

4.2 Privatization

Alpaca privatizes a subset of task-shared variables in a task to keep them consistent in case power fails as the task executes. We describe privatization of scalar (i.e., non-array) data first. Privatization of arrays is described later in Section 4.4. To privatize a variable, Alpaca statically allocates a *privatization buffer* and copies the variable that may become inconsistent to its local privatization buffer. Alpaca re-writes subsequent memory access instructions to refer to the privatization buffer instead of the original memory location of the variable. At the end of the task, right before the transition to the following task, Alpaca commits any changes made to the privatized copy to its original location, using the *two-phase commit* procedure (Section 4.3). Privatization ensures that tasks execute idempotently because updates to memory are committed only after a task has completed. Idempotent execution ensures that a task's effects are atomic, which is one of Alpaca's main language-level guarantees.

The correctness and efficiency of Alpaca's privatization analysis rely on several key properties of Alpaca's design. For efficiency, Alpaca does not privatize all task-shared variables. Instead, Alpaca detects *W-A-R* dependencies during compilation and privatizes only the variables involved in the dependencies (Section 2). To identify affected variables, Alpaca performs an inter-procedural, backward traversal of each task's control-flow graph, tracking accesses to each task-shared variable along each path. If a write and then a read to the same task-shared variable are encountered along *any path* in the backward traversal, Alpaca privatizes that task-shared variable.

Alpaca's compiler generates the instructions for privatizing a variable. The compiler first allocates a privatization buffer in non-volatile memory for each variable that needs to be privatized. At the beginning of the task, the compiler inserts code that copies the variable value from its original location to its privatization buffer. Then, the compiler replaces each reference to the original value inside the task with a reference to the privatization buffer. Before each `transition_to` statement, the compiler invokes the first phase of the two-phase commit operation, `pre_commit` (Section 4.3), passing as arguments the addresses of the original variable and its privatization buffer along with its size.

Figure 4 shows a sketch of Alpaca's instrumentation for an example task code. Compiler-inserted privatization code is in green and code deleted by the compiler is struck-through. As in Line 1, the user defines task-shared variable by annotating it as `TS`. `TS` variables are saved in non-volatile memory. The code in this example requires only `c` to be privatized because it is the only *W-A-R* variable; code accessing all other data requires no instrumentation. Variable `c` is privatized on Line 3, and the access to it on Line 6 is re-written to refer to the private copy `c_priv` (Line 7). After privatization, only the commit operation can modify the location `c`. *Selective* instrumentation avoids runtime overhead and is the key to Alpaca's high performance.

```
1   TS int a, b, c; NV int c_priv;
2   task example_1() {
3     c_priv = c;
4     a = 3;
5     int d = b;
6     ~~c++;~~
7     c_priv++;
8     pre_commit(&c_priv, &c, sizeof(c));
9     transition_to(example_2)
10  }
```

Fig. 4. Privatization and commit. `transition_to` calls `commit`.

Our implementation of the compiler analysis privatizes variables in functions called from multiple tasks, assuming the variable requires privatization in some of its callers. During analysis of a task that calls a function that accesses such a variable, the compiler rewrites the function's body to refer to the variable's privatized copy. Consequently, the variable will remain privatized for any other caller of the same function, even if that caller does not involve the variable in a *W-A-R* dependency. This "contagious" privatization is safe, conservative, and could be eliminated by replicating the function body, creating a version for each combination of privatized and non-privatized variables that the function refers to. We allow contagious privatization in favor of the code bloat from replication. In practice, redundant privatization is rare in the benchmarks that we studied.

Algorithms 1–3 depict Alpaca's privatization analysis. The analysis identifies variables potentially involved in *W-A-R* dependences, adds code to privatize those variables, and adds code to atomically commit privatized copies when a task completes. The code at the end of Algorithm 1 identifies the largest possible number of variables that may need to be committed by a single task and statically allocates a commit list that accommodates them all. Section 4.3 explains in detail how Alpaca uses its `commit_list` to commit privatized data.

Algorithm 1 Pseudo-code for Alpaca Compiler.

```
1: function AlpacaCompiler(Module M)
2:     for t ∈ M.tasks do
3:         warSet ← AlpacaFindWAR(t)                    ▷ Find W-A-R variables
4:         AlpacaTransform(t, warSet)                   ▷ Modify code for W-A-R variables
5:         maxCommitListSize ← Max(maxCommitListSize, warSet.size)
6:     SetCommitListSize(maxCommitListSize)             ▷ Determine commit_list size
```

Algorithm 2 Function Finding *W-A-R* Variables for Each Tasks.

```
1: function AlpacaFindWAR(Task t)
2:     warSet ← ∅
3:     for i ∈ t.instructions do
4:         for v ∈ i.possibleWriteAddress do            ▷ Find writes
5:             if v ∈ taskSharedVariables then
6:                 i.writeSet ← i.writeSet ∪ v
7:         for v ∈ i.possibleReadAddress do             ▷ Find reads
8:             if v ∈ taskSharedVariables then
9:                 i.readSet ← i.readSet ∪ v
10:    for i ∈ t.instructions do                        ▷ Detect W-A-R
11:        for j ∈ i.possiblePreviousInst do
12:            for v ∈ i.writeSet ∩ j.readSet do
13:                warSet ← warSet ∪ v
14:        if i.isFunctionCall then                     ▷ For function call (See Section 4.2)
15:            f ← i.getCalledFunction
16:            for v ∈ f.usedTaskSharedVariables do
17:                warSet ← warSet ∪ v
18:    return warSet
```

Algorithm 3 Function Inserting Privatization and Pre-commit Code When Needed.

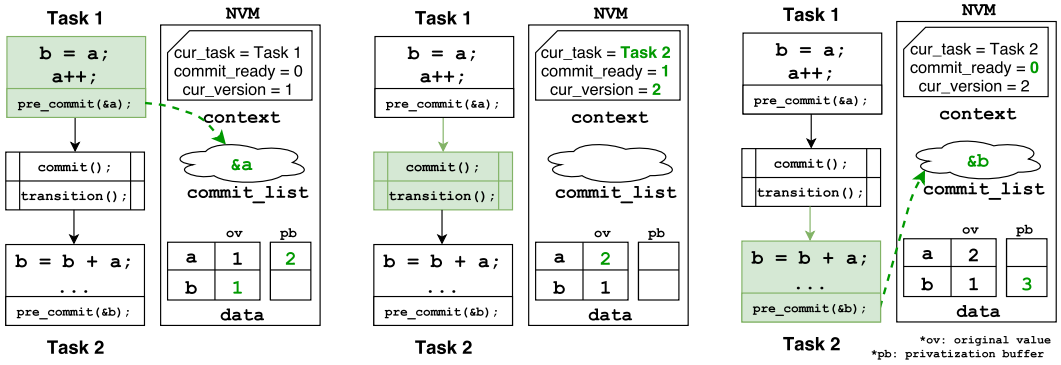
```
1:  function AlpacaTransform(Task t, Set warSet)
2:      for v ∈ warSet do
3:          if v.isPrivatizationBufferAbsent then          ▷ Create privatization buffer
4:              CreateBuffer(v)
5:          InsertPrivatizationCode(t, v)                  ▷ Insert privatization code
6:          for i ∈ t.instructions do
7:              if v ∈ i.usedOperands then                 ▷ Redirect accesses
8:                  RedirectUsageToBuffer(i, v)
9:              if i.isTransitionTo then                   ▷ Insert pre-commit code
10:                 InsertPrecommitBefore(i, v)
```

4.3 Committing Privatized Data

At the end of a task's execution (i.e., upon reaching a `transition_to` statement) Alpaca performs a two-phase commit of updates made to privatized data by that task. The commit operation atomically applies all updates to variables' original locations. The operation is divided into two phases: *pre-commit* and *commit*. The *pre-commit* operation is implemented by the `pre_commit` function in Alpaca runtime library. This function takes the variable information as an argument and records it in an entry in the `commit_list` table, depicted in Figure 5a. The `commit_list` is a table with exactly one entry for each privatized variable. A variable's `commit_list` entry contains the variable's original address, privatization buffer's address, and size. Calls to `pre_commit` are inserted by the compiler at `transition_to` statements, as was described in Section 4.2.

The `commit_list` generated in the first phase records updates to privatized data that must be committed in the second phase. Alpaca stores an *end-index* that always points to the entry after the last valid entry in the `commit_list`. The `commit_list` must be stored in non-volatile memory since its contents must persist if a failure happens during the second phase. As seen in Algorithm 1, our implementation statically allocates a region of memory large enough to fit the maximum number of entries that may be required by any task in the program (i.e., the maximum number of calls to `pre_commit` at any `transition_to` statement in any task). After the last `pre_commit` call before each `transition_to`, the compiler inserts an instruction to set a non-volatile `commit_ready` bit that marks the task ready for the second phase, as shown in Figure 5b. Alpaca runtime checks `commit_ready` on boot. If `commit_ready` is unset, the previously executing task was either in progress or had completed only a partial *pre-commit*, so that task is re-executed from its start, discarding the partial execution or the partial *pre-commit*. Otherwise, the second phase is invoked.

The second phase, *commit*, is implemented in the Alpaca runtime library by a void function, `commit`. The function iterates over entries in the `commit_list` from the first up to *end-index*. For each entry, the variable value is copied from its privatization buffer to its original memory location. The *commit* operation succeeds when it copies all entries in the `commit_list` and sets *end-index* to zero. After a successful commit, the runtime clears the `commit_ready` bit and proceeds to the following task (Figure 5c). If power fails during *commit*, `commit_ready` remains set. Since the runtime checks the bit on boot, it will retry the *commit* operation until it completes successfully. If power fails after *commit* but before `transition_to` completes the transition to the next task, then *commit* will re-execute on next boot and will trivially complete since *end-index* is zero. The `transition_to` that failed to complete will then run again.



(a) Executing task 1 (b) Committing task 1 (c) Executing task 2

Fig. 5. Making progress in Alpaca. Each panel shows the execution at left and the system state at right. The current phase is shaded. We omit privatization instructions for clarity. The system state shows that a in Task 1 and b in Task 2 are privatized into privatization buffers marked pb. Initially, a=1 and b=0. (a) Task 1 writes to b directly and writes to a's privatization buffer because a is involved in a *W-A-R* dependence. Updates to privatized variables are written to the commit_list during the pre-commit phase of the task. A power failure during execution or pre_commit restarts at the beginning of the task. (b) Task 1 proceeds to the commit phases where Task 1 applies its update to a. A power failure during commit restarts in commit. (c) The transition_to operation atomically begins Task 2, which privatizes b because Task 2 reads then writes it.

4.4 Privatizing and Committing Arrays

Alpaca privatizes and commits array variables differently from scalar variables because naively privatizing an entire array (i.e., copying the entire array to a privatization buffer as a task starts) is unnecessary if the task accesses only part of the array. Alpaca statically pre-allocates a privatization buffer for each array that may be read then written (i.e., may be involved in a *W-A-R* dependence). The array's privatization buffer contains the same number of entries as the original array. Privatization takes place at the granularity of an array element. In the example in Figure 6, to privatize array C, the compiler allocates C_priv buffer (Line 2) and inserts the instrumentation code that is highlighted in green (and explained below).

Like a scalar variable, privatizing an array element involves initializing a copy in the privatization buffer (Line 11), redirecting accesses to the buffer (Lines 12-13), and adding the variable to the commit_list via a call to pre_commit (Line 16). Alpaca uses the compiler to redirect array element accesses to their privatization buffers the same as for scalars, but initializing privatization buffers and pre-commit for arrays are different. Alpaca initializes an array element's privatization buffer the first time an execution accesses the element: either explicit instrumentation inserted by Alpaca initializes the buffer before the element's first read *or* the element's first write directly writes to the privatization buffer. Alpaca does pre-commit for an array element only once after the first write to the element.

One key design choice in Alpaca was to decide when instrumentation on a read operation should initialize an array element's privatization buffer. Read instrumentation should not initialize the privatization buffer after a previous write in the task because the initialization would overwrite the written value. Instead, the read instrumentation can initialize the privatization buffer either once before the first read that happens before the first write or (possibly redundantly) at every read

before the first write. We chose the latter option to avoid the overhead of dynamically tracking the first read, which incurs a high runtime overhead.

We avoid invoking pre-commit unconditionally after every write because multiple writes to the same element would append duplicate entries to the `commit_list`, which is inefficient and precludes a statically sized `commit_list`. Furthermore, pre-commits cannot be batched and executed before a task transition (like for scalar variables), because the set of elements dynamically accessed is unknown statically. Batching would require dynamically tracking the set of modified elements in a data structure that supports efficient insertion and traversal which is complex. Executing pre-commit after the first write obviates the complexity of batching and only requires Alpaca to identify the first write to an array element.

```
1   TS int A[30], B[30], C[30];
2   NV int C_priv[30];
3   NV int C_vbm[30];
4   task example_1() {
5     int r = 0;
6     for (int k=0; k<15; k++) {
7       r = rand()%30;
8       A[r] = 3;
9       int d = B[r];
10      if (!vbm_test(C_vbm[r]))
11        C_priv[r] = C[r];
12      C[r]++;
13      C_priv[r]++;
14      if (!vbm_test(C_vbm[r])) {
15        vbm_set(C_vbm[r]);
16        pre_commit(&C_priv[r], &C[r],
            sizeof(C[r]));
17      }
18    }
19    transition_to(example_2)
20  }
```

```
1   #def vbm_test(v) v == cur_version
2   #def vbm_set(v)  v = cur_version
```

Fig. 6. Privatization and commit for arrays.

Correctly handling array privatization and pre-commit requires some instrumentation to execute conditionally, only on an element's first write. To identify an element's first write, Alpaca must track the set U of array elements that have been written since the beginning of the task in the current execution attempt. A write of an element is first if and only if the element is not in this set U at the time of the access. The data structure that represents U needs only to provide efficient insertion and lookup, which our *version-backed bitmask* data structure does. A version-backed bitmask is a bitmask that supports a constant-time clear operation using a versioning mechanism described later in this section. We represent U by setting logical bits (i.e., "entries") in a version-backed bitmask that is statically allocated for each array being privatized. In Figure 6, the version-backed bitmask for `C` is `C_vbm` allocated on Line 3.

Each version-backed bitmask entry is a 16-bit integer *version*. To set an entry (`vbm_set`), Alpaca copies the global `cur_version` counter into the entry. To test an entry (`vbm_test`), Alpaca compares the version stored in that entry to the global `cur_version` counter; equality indicates the entry is set, inequality indicates unset. Consequently, when the global `cur_version` counter changes, all version-backed bitmasks are implicitly cleared. When the `cur_version` counter overflows and rolls over, the runtime explicitly resets all entries in all version-backed bitmasks to zero.

To track the set U of array elements updated in the current task execution attempt, the Alpaca compiler instruments reads and writes to array elements with code to set and test entries in the array's version-backed bitmask. When reading from an array element that has not been modified yet, i.e. its entry in U is unset (Line 10), then the runtime initializes the element copy in the privatization array (Line 11). When writing to an array element for the first time, after checking that its entry in U is not set (Line 14), it inserts the element into U by setting its entry (Line 15), and appends the written array element to the `commit_list` by calling `pre_commit` (Line 16). The set U is cleared at the next task transition or reboot, since the `cur_version` counter increments on each task transition and reboot (Section 4.1), which implicitly clears the version-backed bitmask.

5 ALPACA DISCUSSION

Alpaca's programming model guarantees that tasks will execute atomically. Our Alpaca implementation efficiently provides this atomicity guarantee by selectively privatizing data. Besides programmability, efficiency, and consistency, Alpaca supports I/O operations and allows modular re-use of code. This section discusses these aspects of Alpaca and characterizes its main limitations.

5.1 Low Overhead

A key contribution of this work is that our Alpaca implementation has low overhead compared to existing systems to which we can directly compare (we quantify the difference in Section 7). Alpaca's overhead is low, because privatization is simple and because Alpaca privatizes variables selectively. Privatization has a low cost, primarily because it rarely occurs: most variables are not privatized because they are either local to a task or shared but not involved in *W-A-R* dependences. Furthermore, Alpaca's task-based execution avoids all checkpointing cost. Alpaca needs to retain only the information about which task was last executing. Alpaca does not incur the cost of tens of writes to non-volatile memory to save registers, like Ratchet [Van Der Woude and Hicks 2016], nor the even higher additional cost to save the stack, like DINO [Lucia and Ransford 2015]. By reducing copying and privatizing only when necessary, Alpaca saves time and energy.

5.2 Memory Consistency

Alpaca preserves memory consistency despite arbitrarily-timed power failures by making each attempt to execute a task idempotent. Task idempotence guarantees that if any attempt has sufficient energy to complete, the effects of a single, atomic execution of the task are made visible in memory. The memory state immediately after a task transition is equivalent to the corresponding state in execution on continuous power. Alpaca guarantees idempotence by privatizing non-volatile variables involved in *W-A-R* dependences and requiring volatile state to be task-local.

5.2.1 Non-volatile Memory Consistency. Taking a cue from prior work [De Kruijf and Sankaralingam 2013; de Kruijf et al. 2012; Lucia and Ransford 2015; Van Der Woude and Hicks 2016], Alpaca privatizes only non-volatile variables involved in *W-A-R* dependencies. We show that privatizing only this subset is sufficient by proving that only memory accesses related by *W-A-R* can cause a value written by the task before a power failure to be read by the same task after the power failure.

Consider one task and assume that control flows along the same path each time the task re-executes, which is true of all code that does not perform I/O operations (we discuss I/O later in this section). Consider one memory location and let R_i^j and W_i^j respectively denote the i^{th} memory read and write to that location during the j^{th} attempt to execute the task. If power fails in attempt j after k accesses and the task re-executes, then the sequence of memory accesses is: $X_0^j, \ldots W_p^j \ldots X_k^j$ – [power failure] – $X_0^{j+1} \ldots R_q^{j+1} \ldots$, where X stands for either read or write and our hypothesis postulates a write W_p^j before the power failure and a read R_q^{j+1} that returns the same value. The hypothesis implies that $q < p$, otherwise, W_p^{j+1} would overwrite the value written by W_p^j before R_q^{j+1} reads it. The order $q < p$ implies that R_q^j precedes W_p^j in the task code, which is the definition of a *W-A-R* dependence.

5.2.2 Volatile Memory Consistency. In Alpaca, the only volatile data are task-local variables. Since all local variables must be initialized before use in a task, local reads after a power failure will never access uninitialized memory. Since volatile memory clears on reboot, local reads will never observe a value written before the power failure.

Like prior work [Van Der Woude and Hicks 2016], Alpaca conservatively assumes that compiler optimizations cannot introduce memory read or write instructions and Alpaca safely interacts with any compiler optimization that adheres to this assumption.

5.3 I/O

Code that interacts with sensors and actuators poses three difficulties: (1) some I/O-related actions must execute atomically, (2) external inputs introduce non-determinism, and (3) actuation or output cannot be undone. Alpaca allows the programmer to express (1) and (2) through careful coding patterns that we describe below. Alpaca targets applications that can tolerate repeated outputs, where (3) is acceptable.

Some applications include I/O-related code that should execute atomically, such as the code in Figure 7. The code reads temperature and pressure sensors and sets the `heaterOn` or `coolerOn` flag, based on the sensed data. The temperature and pressure values should be consistent. Alpaca lets the programmer ensure that the values will be consistent by putting the actions in the same task. In contrast, a system with dynamic [Balsamo et al. 2015; Ransford et al. 2011b] or compiler-inserted [Van Der Woude and Hicks 2016] task boundaries gives the programmer no way to ensure that the input operations execute atomically.

```
TS bool heaterOn, coolerOn;
task measure(){
  int temp = sampleTemp();
  int pres = samplePressure();
  if (pres > THRESHOLD){
    if (temp < 0)
      heaterOn = true;
    else
      coolerOn = true;
  }
  transition_to(actuate);
}
task actuate(){
  assert(heaterOn != coolerOn);
  ...
  transition_to(measure);
}
```

Fig. 7. I/O in Alpaca.

The code in the example asserts that `heaterOn` and `coolerOn` are never both `true`. The code misbehaves if a power failure occurs after assigning one of the flags (e.g., `heaterOn`). If the sensor's result is different in the following execution attempt, the code could set the other flag (e.g., `coolerOn`), violating the assertion. The core issue is that non-volatile memory updates are conditionally dependent on sensed inputs. If control-flow depends on the input, then *conditional* non-volatile memory updates can violate task idempotence. We note that this problem also afflicts prior efforts [Colin and Lucia 2016; Lucia and Ransford 2015; Van Der Woude and Hicks 2016]. A programmer can preserve idempotence by using intermittence-safe I/O programming patterns. Concretely, one programming pattern that avoids the problem in this example is to use a dedicated task to read and store both `temp` and `pres`, and to use another task to do the conditional updates to `heaterOn` and `coolerOn`. Alternatively, a programmer could avoid the problem by ensuring that both execution paths access the same set of memory locations: inserting `coolerOn = false;` on the `if` branch and inserting `heaterOn = false;` on `else` branch.

5.4 Forward Progress

Guaranteeing forward progress in an intermittent, energy-harvesting system is a difficult problem that is orthogonal to the problems solved by Alpaca. The key challenge is that a system buffers a fixed amount of energy before it begins operating and if the energy required by a task exceeds the buffered amount, the task will never complete executing, preventing progress. A task's energy cost can be input dependent, adding further complexity. This progress issue is not unique to Alpaca, afflicting prior task-based systems as well [Colin and Lucia 2016; Lucia and Ransford 2015; Mirhoseini et al. 2013; Ransford et al. 2011b; Van Der Woude and Hicks 2016].

Prior work has used *ad hoc* techniques that attempt to ensure progress, to the detriment of other system characteristics. Ratchet [Van Der Woude and Hicks 2016] inserts a dynamic checkpoint between static checkpoints after repeatedly failing to make progress. Other systems [Balsamo et al.

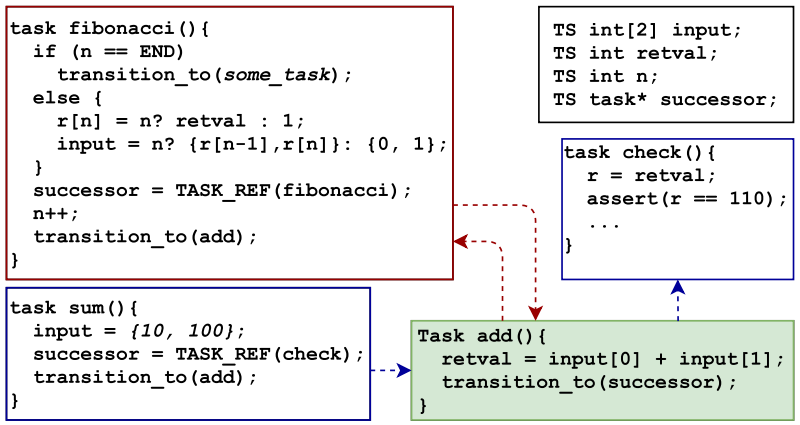


Fig. 8. Task modularity. Task add is parametrized and reused by task `fibonacci` and `sum`. Task-shared variables are used to pass arguments (`input`), return value (`retval`), and the successor task (`successor`).

2016, 2015; Ransford et al. 2011a] dynamically checkpoint in response to an interrupt when energy is low. Dynamic checkpointing requires capturing enough state to restart from an arbitrary point, which can take a prohibitive amount of time [Colin and Lucia 2016], especially with hybrid volatile and non-volatile memory. Dynamic checkpointing may also violate I/O atomicity (see Section 5.3).

We opted not to include a dynamic checkpointing fall-back in Alpaca. Instead the programmer must ensure for sizing tasks such that tasks in their program do not require more energy than their target device can buffer. As long as this condition is satisfied, Alpaca always avoids atomicity violations and guarantees correctness. None of the tasks in our test programs have a forward progress problem. It would be straightforward to incorporate a dynamic checkpointing fall-back into our Alpaca prototype.

5.5 Reusability of Tasks

In a task-based programming model for intermittent execution, code re-use via functions is insufficient, because functionality that requires multiple tasks cannot be encapsulated in a function. Any function called from a task must fit within the task's energy budget, which is constrained by the energy capacity of the device. Alpaca supports modular re-use of *task groups* that contain common functionality. Multiple predecessor tasks in the program can invoke the same task group by passing a distinct *successor task* identifier and input data through task-shared variables to an entry task in the group. The task that runs last in the group transfers control to the passed successor task. Alpaca's simple approach to modularity is a contrast to Chain's approach, which requires complex mechanisms for indirect access to channels.

Figure 8 shows an example of how different tasks can reuse code encapsulated in an Alpaca task. In the example, the `add` task is parametrized and invoked from the `fibonacci` task and from the `sum` task. The re-used task accepts two inputs via a task-shared variable and writes their sum into another task-shared variable. A more sophisticated implementation than ours could eliminate the need to explicitly specify task-shared variables for the input arguments and the return value and support syntax similar to traditional function calls.

5.6 Prototype Limitations

Our Alpaca prototype supports a useful subset of the C language, handling most uses of pointers and complex data structures. Our prototype has a few implementation-specific limitations, which we emphasize are not fundamental limitations of Alpaca.

We implemented a limited pointer alias analysis and our prototype requires that a `TS` pointer only ever be assigned the address of a `TS` variable if that address is constant. Allowing `TS` pointers to constant variables permits the especially important case of function pointers.

Our prototype requires the programmer to refer to array elements directly, i.e., writing `A[30]` instead of `*(p + 30)`. Our prototype statically inserts code to maintain version-backed bitmasks on array accesses. Array indirection would require our prototype to use instrumentation that dynamically disambiguates pointers to arrays, to determine which bitmask to update. Our prototype makes the calculated choice to avoid this additional dynamic analysis cost by requiring direct array access. We note that this strategy is similar to DINO [Lucia and Ransford 2015].

6 BENCHMARKS AND METHODOLOGY

We evaluated Alpaca using a collection of applications taken from prior work running on real, energy-harvesting hardware. Our evaluation ran on a TI MSP430FR5969 microprocessor on MSP-TS430RGZ48C project board for experiments with continuous power, and on a WISP5 [Sample et al. 2008] for experiments with harvested energy. We used EDB [Colin et al. 2016] for measurements when running on harvested energy. To power the WISP, we used the ThingMagic Astra-EX RFID reader as an RF energy source with its power parameter set to `-X 50`, and a distance between the WISP and the power source of 20cm.

We evaluated Alpaca using applications ported to run on harvested energy using DINO [Lucia and Ransford 2015], Chain [Colin and Lucia 2016], and Alpaca, allowing for a thorough direct comparison. DINO and Chain versions of four applications were provided by the authors of the Chain [Colin and Lucia 2016] paper: activity recognition (ar), cuckoo filter (cf), rsa encryption (rsa), and cold-chain equipment monitoring (cem). We ported two additional applications from the MIBench [Guthaus et al. 2001] to run with DINO, Chain, and Alpaca.

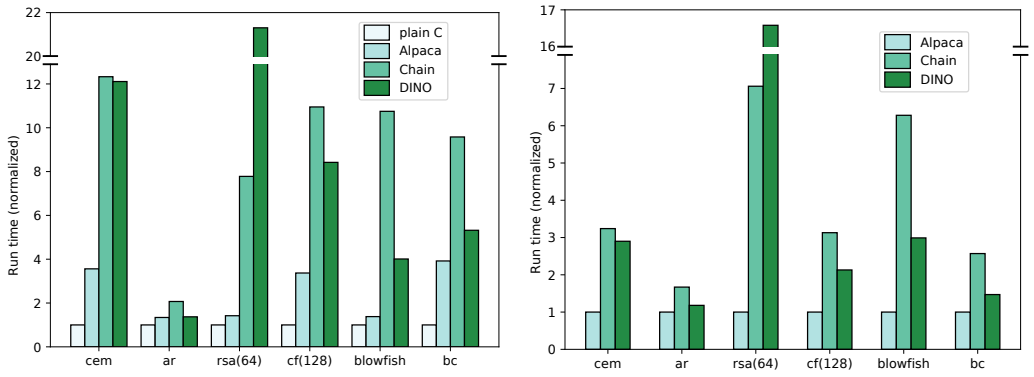
We studied six applications:

Activity Recognition (ar) AR buffers 128 samples from a three-axis accelerometer, featurizes those samples, trains a model, and uses a model to do nearest neighbor classification that determines whether the device is stationary or shaking. The features computed are the mean and standard deviation in each dimension over the window of samples. To make experiments reproducible across trials, we emulated accelerometer readings with a pseudo-random number generator, although Alpaca fully supports AR with an accelerometer. We went through stationary data training phase three times, moving data training phase two times, and testing phase two times.

RSA Encryption (rsa) RSA encrypts a fixed, in-memory input string of arbitrary size, using a fixed encryption key. We used an 11-byte input string and a 64 bit key in our experiments. With RSA's existing task definitions, the Chain version of RSA got stuck in a task that always exhausts its energy budget with keys larger than 64 bits.

Cuckoo Filtering (cf) CF uses a cuckoo filter to store a sequence of pseudo-random numbers, then queries the filter to recover the sequence. We use 128-entry filter filled to one-fourth capacity in each trial.

Cold-chain Equipment Monitoring (cem) CEM logs and periodically LZW-compresses temperature sensor data. For repeatability, we emulated the sensor with pseudo-random numbers because LZW has input-dependent run time. We used a 512-entry dictionary and 64-byte compressed block size.



(a) On continuous power

(b) On harvested energy

Fig. 9. Run time performance. Data are normalized to performance of (a) plain C and (b) Alpaca.

Blowfish Encryption (blowfish) Blowfish encrypts a string of arbitrary size using an encryption key. We encrypted a 32-byte string using five sub-key arrays, excluding the derivation of the sub-keys. We used a P array with 18 32-bit entries and S0, S1, S2, and S3 arrays each 256 32-bit entries. Our code is ported from MIBench [Guthaus et al. 2001].

Bitcount (bc) Bitcount counts the set bits in a random string using seven different algorithms and compares their result to ensure they executed correctly [Guthaus et al. 2001]. We ran each test with 100 pseudo-random inputs.

To ensure a fair comparison, applications use identical task definitions for both Chain and Alpaca, and we inserted task boundaries at equivalent code points for DINO (We study the impact of task boundary placement in Section 7.5). Some system parameters in our evaluation vary from the defaults provided by the authors of Colin and Lucia [2016]; compiler and device energy buffer size changes required us to modify some input and task sizes to ensure applications run to completion and we used these parameters consistently across systems in our evaluation.

7 EVALUATION

Our evaluation compares directly to Chain and DINO and illustrates several findings about Alpaca. The data show that Alpaca outperforms existing systems while running natively on existing hardware both on harvested energy and running on continuous power. Our evaluation characterizes these findings, showing that Alpaca avoids the costliest time and memory overheads of prior approaches. We note that consequently Alpaca is applicable to more devices and less expensive hardware options than other systems. We qualitatively and quantitatively show that programming with Alpaca is simple compared to other approaches. We also contrast our Alpaca implementation with an alternative Alpaca design that privatizes data to volatile memory, showing that our baseline design is usually more efficient because of additional overheads required by volatile privatization.

7.1 Run Time Performance

Figure 9 shows Alpaca's run time performance, measured *on real hardware* on both continuous power and on harvested RF energy. Performance on continuous power is an upper bound on performance because it avoids reboot-related overhead. Performance on harvested energy includes all reboot-related overheads and is representative of a real-world deployment.

Figure 9a shows performance on continuous power for each system, normalized to a plain C implementation that implements each application without considering intermittence. As expected, Alpaca has an overhead compared to plain C code, with the slowdown from 1.3x to 3.6x. However, Alpaca outperforms both previous state-of-the-art systems Chain and DINO, by 1.5x to 7.8x and 1.02x to 15x respectively.

Figure 9b shows performance on harvested energy. Here, we omit the plain C variant because it does not handle intermittence and cannot run correctly on harvested energy. Alpaca outperforms all other systems, by 1.7x to 7.1x for Chain and 1.2x to 16.6x for DINO. The performance gap is larger on harvested energy because power failures introduce reboot-related overheads and Alpaca's reboot overhead is extremely low. DINO has a higher overhead because it must restore the stack and register file on every reboot.

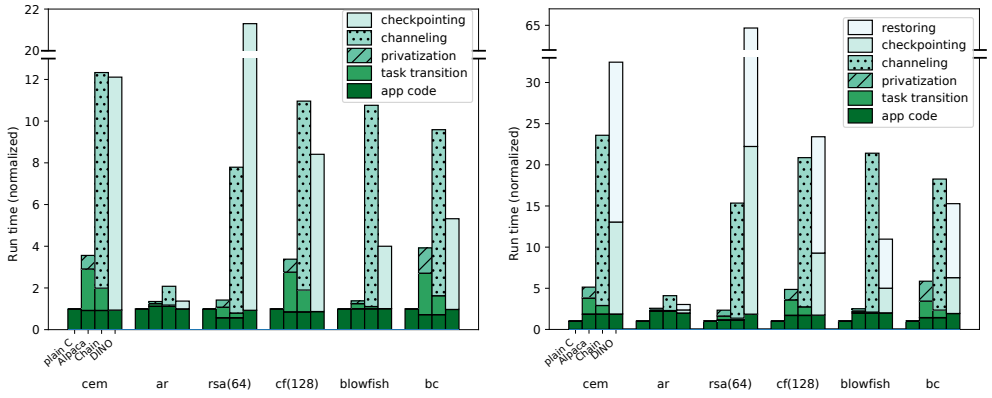
7.2 Characterizing Alpaca's Runtime Overhead

To better understand Alpaca's performance, we made detailed measurements of each system's major overheads. Alpaca's main overheads are privatization and task transitioning. Chain's major overheads are channel manipulation and task transitioning. Alpaca task transitions commit privatized state and Chain's task transitions commit all data written to "self" channels. DINO's major overheads are checkpointing and versioning, and restoring the stack and register files on reboot.

We measured each system's overheads using the microcontroller's internal timer. Our measurements are conservative because Alpaca's privatization is often a single-cycle `mov` operation which may take less time than the minimum resolution of the timer; our measurements may over-estimate Alpaca's overheads. Alpaca's task transitioning overhead is larger than the timer's resolution, allowing for precise measurement. Measuring Chain's overhead requires distinguishing between the inherent read and write functionality of channel manipulation and the channel overhead, which is difficult. Instead of measuring channel manipulation overheads directly, we assume that because Alpaca and Chain have the same code structure, they should have the same run time, except for their overheads. We compute overhead-free application-only run time for Alpaca by subtracting its measured overhead from its total run time. We then compute Chain's channel manipulation overhead by subtracting the application-only run time and measured transition overhead from Chain's total run time. We directly measured DINO's checkpointing and restoration overhead using the microcontroller's timer.

7.2.1 Best Case Overhead. Figure 10a shows overheads measured on continuous power, which represents the best case, with no restart overheads and re-execution costs. The data show that Alpaca has high performance because it imposes few overheads. Privatization requires many fewer operations than Chain's channel manipulations and DINO's checkpointing and versioning. Alpaca's frequent committing makes its task transition overhead twice as large as in Chain, but the performance gain from efficient privatization amortizes this overhead.

7.2.2 Worst Case Overhead. Based on the measured best case data, we computed worst case overheads. Assuming no task requires more energy than the device can buffer, a task will suffer at most one power failure before successfully completing. We compute the worst case overhead by assuming a power failure on the first execution attempt of each task, just before the task completes. In this worst case, the system must re-execute from the previous task boundary or checkpoint, roughly doubling the application's execution time and maximally repeating operations like initialization after reboot, privatization, and channel operations. We computed the worst case run time for Alpaca and Chain by doubling the application run time, and privatization and channel overheads. To compute the worst case run time for DINO, we invoked restore manually for each checkpoint and measured the overhead.



(a) Best case overhead.

(b) Worst case overhead.

Fig. 10. Attributing slowdown to sources of overhead. Bars for each application are plain C, Alpaca, Chain, DINO. We baseline both plots to plain C to allow direct comparison, even though plain C *cannot execute* on intermittent power.

Figure 10b shows the worst case overhead results, in this case including the plain C version with the caveat that *the plain C version cannot actually run because it does not handle intermittence*. In the worst case, Alpaca has high performance, outperforming Chain by 1.6x to 7.3x and DINO by 1.2x to 27.6x. Comparing with Figure 10a, DINO's performance degrades considerably because, while Chain and Alpaca have no additional overhead on reboot, DINO must restore a checkpoint on each reboot, imposing an overhead.

Directly measuring actual per-operation overheads on harvested energy is experimentally difficult, but the overheads will be between our measured best case and modeled worst case overheads. The worst case model is adversarial, interrupting every task once before allowing it to complete, leading to an artificially high reboot count. In our experiments with real hardware on harvested energy, we observed few reboots (around 10 per test), suggesting that real-world overhead figures are more likely to resemble the best case than the worst case (i.e., Figure 9b).

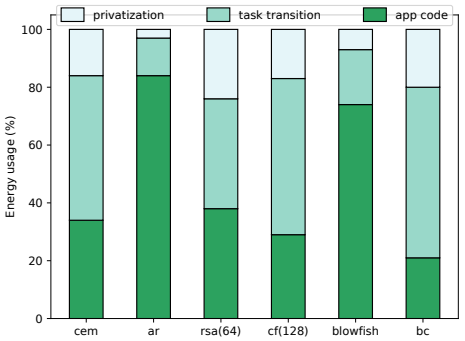


Fig. 11. Breakdown of energy overheads.

7.2.3 Energy Overhead. We measured Alpaca's energy overhead using TI's *MSP EnergyTrace* tool. On continuous power, we removed privatization code and transition code and measured total energy use to deduce the energy use of each.

Figure 11 shows a breakdown of Alpaca's energy overheads, including application code execution, task transitioning including commit, and privatization including pre_commit. The trend is consistent with the breakdown of run time overhead. Applications in which tasks exchange smaller blocks of data tend to have lower overhead. For example, AR has low overhead because different tasks only share input data, unlike RSA, in which tasks share large arrays. Section 7.4 evaluates the energy impact of privatizing data to SRAM instead of FRAM.

7.3 Non-Volatile Memory Consumption

We measured non-volatile memory consumption by inspecting each application binary, which is sufficient because these applications do not dynamically allocate non-volatile memory, as is typical in embedded systems. DINO reserves double-buffered checkpointing space equal to twice the maximum stack size of 2KB, i.e., 4KB total. Figure 12 shows that Alpaca uses less non-volatile memory in all applications, by factor of 2.1x to 4.4x and 1.3x to 5.5x for Chain and DINO, respectively. Alpaca uses less non-volatile memory than Chain mainly because Chain creates multiple versions of variables that exist in different channels. Alpaca uses less non-volatile memory than DINO because DINO checkpoints all volatile state and versions some non-volatile state, while Alpaca never checkpoints and only selectively privatizes non-volatile state.

Using less non-volatile memory than other systems makes Alpaca applicable to different and cheaper hardware configurations with different (i.e., smaller) non-volatile memory sizes and dollar cost. We compared the total non-volatile memory requirement for each application and configuration to the entire MSP430FRxxxx product line, microcontrollers in which vary primarily in price and memory configuration. Alpaca enables applications to be deployed on cheaper, smaller hardware. For example, Alpaca allows AR, RSA, and bitcount to deploy on product line's bottom-end MCU, the MSP430FR2110, with only 2KB of non-volatile memory. However, DINO requires more non-volatile memory, limiting hardware options to microcontrollers with at least 8KB of non-volatile memory, such as the MSP430FR2302, which is 1.8x more expensive [TI Inc. 2017b] than the MSP430FR2110. By a similar analysis, deploying blowfish using Chain requires hardware that is 2.4x more expensive than hardware available to blowfish using Alpaca.

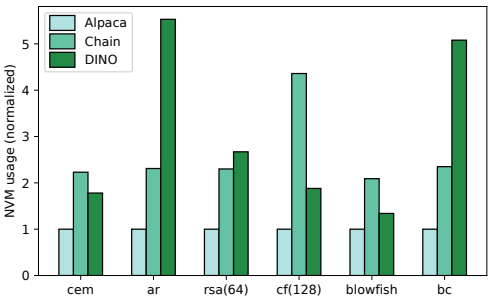


Fig. 12. Non-volatile memory use.

7.4 Privatizing Data to Volatile Memory

We evaluated an alternative implementation of Alpaca, called Alpaca-VM that uses volatile memory to store privatized values, motivated by the fact that volatile memory accesses require less energy than non-volatile memory accesses. Privatizing to volatile memory instead of non-volatile is not straightforward because a consistent set of data privatized to volatile memory must be atomically committed, even if power fails during commit. To ensure that volatile values commit atomically despite failures, Alpaca-VM must make a full copy of all privatized *values* to a non-volatile commit buffer during pre-commit, along with the usual memory address and data size. In contrast, Alpaca avoids writing values to the commit buffer, saving a considerable number of non-volatile memory writes, especially when privatizing large `struct` type data.

Copying from volatile memory into the non-volatile commit buffer adds time and energy overhead to committing. Privatizing data to volatile memory is only a net benefit if the time and energy saved by using volatile memory in the task are more than the time and energy consumed by copying to the commit buffer. In general, if tasks often re-use privatized data before committing, then Alpaca-VM will use less energy and time than Alpaca.

We experimented with a microbenchmark to measure how many accesses to volatile privatized data are required to amortize the increased pre-commit cost of using volatile privatization buffers. The benchmark is a repeated task that does a fixed number of read-modify-write (RMW) operations. The benchmark varies the number of RMWs in the task to find the "tipping point" number, when using volatile privatization buffers has better performance. The data show that Alpaca-VM's performance relative to Alpaca improves as the number of RMWs per task grows. When the task grows to around 110 RMWs, Alpaca-VM begins to outperform Alpaca. If an application has a number of accesses to privatized data greater than this "tipping point" number, volatile privatization pays off. We quantified data re-use in our real applications and Table 3 shows the average number of reads and writes to each privatized variable per task for each application. The data show that the average number of reads and writes is much smaller than the tipping point, suggesting that volatile privatization is unlikely to pay off.

Table 3. Average per-task re-use of privatized data.

	cem	ar	rsa	cf	blowfish	bc
Read	2	2.33	4.29	1.44	2.17	1
Write	1	1	1.29	1	1	1

We ran each application using Alpaca-VM and Figure 13 shows actual performance on benchmarks. The data show that Alpaca-VM's performance is often worse than, or negligibly different from Alpaca's performance, which is consistent with our "tipping point" characterization. While Alpaca-VM does not show any improvement in our experimental system, it is not an uninteresting design: volatile privatization is likely to be viable and beneficial in a system with a much larger energy buffering capacitor that accommodates more (i.e., hundreds of) reads and writes in each task.

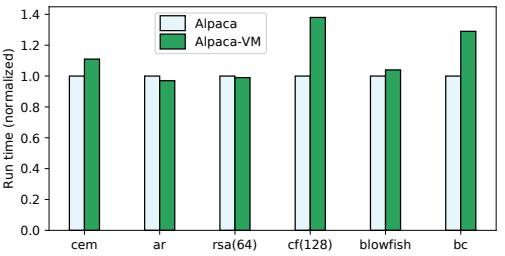


Fig. 13. Run time of Alpaca-VM versus Alpaca.

7.5 Comparing Programmer Effort

We compared the programming effort required by Alpaca to the effort required by Chain and DINO and found that Alpaca requires reasonable code changes compared to plain C code, requires less change than writing Chain code, and is often easier to optimize for performance than DINO code. Like Alpaca, Chain also requires the programmer to decompose code into tasks, which is different from writing typical C code and we characterize task sizing next. Unlike Alpaca, Chain also requires additional effort to re-write memory access code in terms of channel operations, which is different from a typical C programming. Alpaca instead allows code to manipulate task-shared variables like ordinary C variables using loads and stores.

7.5.1 Quantifying Programmer Effort. We quantified the difference in programmer effort between systems by comparing the differences in the number of lines of code (LoC) and the number of keywords by each system. Keywords are divided into three types: boundary, declaration, and read/write. Boundary keywords (Bnd) represent task boundaries (i.e., `transition_to`) in Alpaca and Chain, and checkpoints in DINO. Declaration keywords (Decl) modify function and data declarations: `task` and `TS` for Alpaca, and `task` and `channel` declaration for Chain. Read/Write keywords (R/W) access memory and only occur in Chain (channel in and channel out), because Alpaca and DINO use a standard C read/write memory interface.

Table 4. Lines of code and number of keywords.

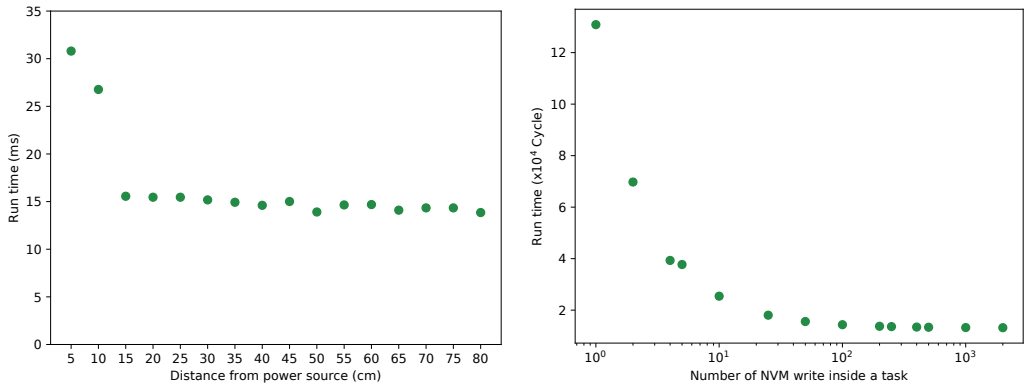
App	Alpaca			Chain				DINO	
	LoC	# Bnd.	# Decl.	LoC	# Bnd.	# Decl.	# R/W	LoC	# Bnd.
CEM	372	19	28	721	19	40	63	338	13
AR	466	19	26	713	19	34	57	439	8
RSA	765	27	40	1197	27	53	123	722	35
CF	397	19	29	707	19	41	72	335	11
Blowfish	614	18	24	740	18	29	75	556	9
BC	313	23	26	588	23	26	57	276	10

Table 4 summarizes the data. On average, the number of lines of Alpaca code is 10% higher than DINO code, but Alpaca requires 37% fewer lines then Chain. The number of keywords used by Alpaca code is 240% more than the number used by DINO code, but is only 27% of the number used by Chain code. Although these data are only a rough indicator of programming complexity, the data suggest that Alpaca's complexity lies somewhere between Chain and DINO.

7.5.2 Choosing a Task's Size. Dividing a program into tasks is a key part of Alpaca development, and we we experimentally characterize the process to show that it is reasonable. Alpaca preserves forward progress at the granularity of a task assuming the system eventually buffers sufficient energy to complete each task. However, a real, energy-harvesting system with a fixed-size energy buffer, may never be able to buffer sufficient energy for a very long task to complete, preventing progress. If a task is too short, its privatization, commit, and transition overhead will be relatively very high, impeding performance. Based on knowledge of the device and the energy cost of program tasks, the programmer must assign work to an Alpaca task.

While it is a non-trivial programming task, defining the extents of Alpaca tasks requires only modest programmer effort. We observed that on today's energy-harvesting hardware, the task decomposition problem is independent of input power and depends only on the device's energy buffer size. Figure 14a shows data for a microbenchmark that runs a loop on a WISP5 [Sample et al. 2008] device harvesting energy from an RF power supply. The x-axis shows the distance to the RF power supply, which corresponds to input power. The y-axis shows the time to the first brown out, at which point the device has exhausted energy accumulated in its capacitor and must slowly recharge. Except for distances so small that the RF supply effectively continuously powers the device (~10cm), the amount of work that the system can execute before browning out is invariant to input power; *the energy buffer is constant*. Forming Alpaca tasks is thus a reasonable (albeit non-trivial) task because the programmer need only reason about the total energy cost of a task. The programmer need not reason about instantaneous input power, nor the power envelope of particular hardware operations, which would be difficult.

We also experimentally observed that choosing a task size that amortizes privatization, commit, and transition costs is not overly challenging. On a WISP5 device, we studied the effect of task size on the run time of a microbenchmark that executes a fixed amount of work across a varying size of tasks. The microbenchmark executes a fixed total number of read-modify-write operations on entries in an array. We varied the number of accesses per task, and Figure 14b shows the relationship between task size and total run time. Run time decreases as task size grows because tasks better amortize commit and transition cost. However, the effect saturates as tasks grow, revealing that even relatively small tasks of around 100 read-modify-write amortize Alpaca's overheads well. The data suggest that choosing a task size that amortizes task overheads will not be prohibitively challenging to a programmer.



(a) One charge cycle run time for various distance

(b) Run time for various task size

Fig. 14. Choosing task size. (a) *Energy* availability does not vary with input *power*. (b) Task overhead varies with task size.

7.5.3 Manual Optimization Complexity. Unlike Alpaca, DINO does not require code to be written in task functions. Instead, it requires boundaries which may be simpler but complicate performance optimization and predictability. DINO complicates reasoning about performance because the cost of a DINO task boundary is not explicit, instead varying dynamically with call depth and stack allocation. DINO puts modularity at odds with performance: passing function arguments increases checkpointing cost of a task boundary. We observed subtle performance effects introduced by DINO that are unlikely to be obvious to a typical performance-conscious programmer.

We conducted a case study to illustrate that writing code in DINO is not necessarily simpler than writing task-based code because DINO compromises performance predictability. This study revealed to us that Alpaca is immune to some performance-specific programming issues that afflict DINO. We hand-optimized the cost of DINO task boundaries for CEM. We carefully removed stack-allocated variables and function arguments and made them non-volatile, global variables to reduce the stack size and decrease checkpointing cost. The optimization process was trial-and-error and decreased the modularity of the code. As a result, we got 36.8% performance improvement (which remained slower than Alpaca) by tediously removing 18 local variables, introducing 14 global non-volatile variables, and modifying the interfaces of 6 functions in 400 lines of code. The performance improvement suggests that a simple translation from naive C style to a checkpointing system may not produce performant code because programmers are not thinking about the costs imposed by checkpointing. Alpaca's task model does not checkpoint the stack and requires tasks to initialize task-local variables, eliminating the need for this optimization.

8 RELATED WORK

Alpaca relates to prior work in several areas. Most related are prior efforts studying intermittent computing, some of which discussed in Section 2. We also relate Alpaca to work on idempotent compilation, systems with non-volatile memory, transactions and transactional memory, and continuation-passing style.

8.1 Energy-Harvesting and Intermittent Computing

There is a large body of work on intermittent execution and other support for intermittent systems. Mementos [Ransford et al. 2011b] was among the first system to address computing on

intermittent power, by monitoring the device energy level and checkpointing registers and stack. Idetic [Mirhoseini et al. 2013] optimized where to poll and used a custom circuit to checkpoint volatile state.

Periodic checkpointing approaches are correct with a purely volatile memory system. However, if the program manipulates non-volatile memory, which is common in widely available platforms [Sample et al. 2008; Zhang et al. 2011a], periodic checkpointing alone is incorrect [Ransford and Lucia 2014]. DINO [Lucia and Ransford 2015] keeps non-volatile and volatile memory consistent by versioning non-volatile memory and checkpointing volatile state. Ratchet [Van Der Woude and Hicks 2016] assumes that main memory is entirely *non-volatile* and keeps memory consistent using idempotence analysis and register checkpoints. These prior approaches all use some form of periodic volatile state checkpoints. Alpaca avoids volatile state checkpointing, eliminating its time and space overhead.

Instead of periodic checkpointing, QuickRecall [Ransford et al. 2011a], Hibernus [Balsamo et al. 2015], and Hibernus++ [Balsamo et al. 2016] do on-demand checkpointing of volatile state when supply voltage is below a threshold. This approach is effective, but requires continuous supply voltage measurement hardware, which is not typically available [Sample et al. 2008; Zhang et al. 2011a]. Also, choosing a threshold voltage is not straightforward. Too high a threshold makes the system checkpoint and wait for energy, even if there is ample energy to continue. Too low a threshold may fail to guarantee that checkpointing completes, which is especially problematic with a variable size call stack and arbitrary global variables. Alpaca is energy agnostic, avoiding hardware requirements and threshold voltage assignment issues. Chain [Colin and Lucia 2016] avoids checkpointing using a task-based execution strategy, like Alpaca. Chain uses static multi-versioning to keep non-volatile data consistent at a high cost in time and space (see Section 7). Non-volatile processors propose architectural non-volatility [Ma et al. 2015a,b] making intermittent software simple, but precluding the use of existing hardware and imposing a performance and complexity overhead. Dewdrop [Buettner et al. 2011] runs small, "one-shot" tasks on intermittent hardware, optimizing task scheduling to maximize task completion likelihood given limited energy. Dewrop, however, does not support computations that span failures.

Other work addresses intermittent computation, like Alpaca, but unlike Alpaca, these efforts are not programming or execution models. Wisent [Aantjes et al. 2017; Tan et al. 2016] addresses intermittence, but is not a computing model, instead enabling reliable software updating of *in situ* intermittent devices. Ekho [Zhang et al. 2011b] helps test intermittent devices with support to collect and replay representative power traces from a realistic environment. EDB [Colin et al. 2016] is a hardware/software tool that allows programmers to profile and debug intermittent devices without interfering with their energy level. Federated energy [Hester et al. 2015] is a disaggregated energy buffering mechanism that decouples the energy storage of different hardware components. Some earlier work addresses computing using harvested energy, but unlike Alpaca, these systems to not explicitly address intermittent computation. Eon [Sorber et al. 2007] is one of the earliest efforts to target harvested-energy computation, scheduling prioritized tasks based on energy availability. ZebraNet [Juang et al. 2002] dealt with the challenges of solar energy in an adversarial environment.

8.2 Idempotent Code Compilation

Several prior efforts [De Kruijf and Sankaralingam 2013; de Kruijf et al. 2012; Zhang et al. 2013] noted that a program decomposed into idempotent sections is robust to a number of failure modes because idempotent sections can be safely re-executed. Idempotence systems break *W-A-R* dependances by dividing dependent operations with a checkpoint (or section boundary). Like these systems, Alpaca leverages the fact that eliminating *W-A-R* dependences makes tasks idempotently re-executable. Unlike other systems, however, Alpaca does not make code sections idempotent by inserting

checkpoints. Instead Alpaca ensures task atomicity by using task-based execution to avoid the need for volatile state checkpoints, and privatization of non-volatile data involved in *W-A-R* dependences to make tasks idempotently restartable.

As discussed in Section 2, Ratchet [Van Der Woude and Hicks 2016] uses compiler idempotence analysis to insert checkpoints to make inter-checkpoint regions idempotent, assuming main memory is entirely non-volatile. Alpaca makes no assumption about memory volatility making it applicable to more varied hardware, and its tasks' sizes are free from idempotence analysis, unlike Ratchet.

8.3 Memory Persistency and Non-Volatile Memory Systems

The increasing availability of non-volatile memory creates a need for models defining the allowable reorderings of non-volatile memory updates and *persist* actions, which ensure data become persistent [Pelley et al. 2014, 2015]. Relaxing the ordering of updates and persist actions to different locations may expose a re-ordering to code resuming execution after a failure and persistency models describe which of these re-orderings are valid. Other, earlier work developed mechanisms for managing data structures in non-volatile memory, and for building consistent memory and file systems out of byte-addressable non-volatile memory [Coburn et al. 2011; Condit et al. 2009; Doshi and Varman 2012; Dulloor et al. 2014; Moraru et al. 2013; Narayanan and Hodson 2012; Venkataraman et al. 2011; Volos et al. 2011]. Alpaca relates to these efforts because both aim to keep non-volatile memory consistent across power failures. The prior work differs from Alpaca, however, in purpose and mechanism. Alpaca is programming model and run-time implementation that keeps data consistent across extremely frequent failures in intermittent executions. These prior efforts focused on large-scale systems and are only peripherally applicable to intermittent devices.

8.4 Transactions and Transactional Memory

Transactions [Gray and Reuter 1992] and, in particular, transactional memory [Hammond et al. 2004; Harris et al. 2005; Herlihy and Moss 1993; Shavit and Touitou 1995] (TM) systems are also related to Alpaca. Transactional memory targets multi-threading systems. A transaction speculatively updates memory until a (usually) statically defined atomic region ends. Transactions commit when they complete execution, updating globally visible state, or aborting their speculative updates due to a conflicting access in another thread, and beginning execution again. Transactions are similar to Alpaca because Alpaca buffers a task's updates privately, committing them to global memory when a task ends. Moreover, when a power failure interrupts a task, its privatized updates are aborted and it begins again from its start. However, Alpaca differs in that it targets intermittent systems with potentially extremely frequent failures. Unlike TM, Alpaca does not target multi-threaded programs, instead aiming to keep memory consistent between re-executions across power failures.

8.5 Continuation-Passing Style

Continuation-passing style [Appel and Jim 1989] (CPS) is a programming style in which each function explicitly calls the next function in a sequence as a *continuation* rather than returning. CPS code makes control and state more explicit, often leading to clearer semantics and simpler analysis of, e.g., intermediate state. CPS is similar to Alpaca in that tasks are like continuations that execute in sequence, avoiding in Alpaca's case the need to track the stack or handle task nesting. CPS code is often written in a functional language, which, like Alpaca tasks makes continuations idempotent without the need for explicit privatization of mutable state. We plan to explore the role of functional and explicitly continuation passing programming models in the intermittent computing context in our future work. One promising direction is continuation-passing C [Kerneis and Chroboczek 2010] (CPC), which uses continuation-passing style in the C language for efficient concurrent execution. CPC is similar to Alpaca because CPC executes short code regions one after

another without exposing intermediate state, similarly to Alpaca tasks. Both system restrict the stack's lifetime to the small scope of a task or continuation, eliminating the need for checkpoints in Alpaca, and reducing the cost of concurrent conflicts in CPC. The goals of these systems are, however, fundamentally different. CPC is for explicitly concurrent programs, while Alpaca is for (implicitly concurrent) intermittent programs. Moreover, CPC uses a run time scheduler, while Alpaca uses explicit inter-task control flow via `transition_to`.

9 CONCLUSION AND FUTURE WORK

This work proposed Alpaca, a programming model for low-overhead intermittent computing that does not require checkpointing, using a task-based execution model and a data privatization scheme built on idempotence analysis. Compared to competitive systems from prior work, our Alpaca prototype, which will be made available after publication achieves up to 7.8x and 15x improvement in run time and 1.3x-5.5x improvement in non-volatile memory consumption on a real, energy-harvesting device for a variety of applications from the literature. Along with Alpaca's effectiveness and high performance, our evaluation showed that Alpaca is reasonable to program and allows applications to run on more diverse and cheaper hardware options, increasing its applicability and impact. Looking to the future, Alpaca emphasizes a need raised by Chain, Ratchet, and DINO for a system to aid, or automate the decomposition of a program into tasks, which is currently a reasonable task, but mostly a manual process.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their valuable feedback and to Vignesh Balaji and Emily Ruppel for contributing to discussions about the work. This work was supported by National Science Foundation Award *CNS-1526342*, a Google Faculty Research Award, and a gift from Disney Research. Kiwan Maeng was partially supported by a scholarship from the Korea Foundation for Advanced Studies. Visit `http://intermittent.systems`.

REFERENCES

Henko Aantjes, Amjad Y Majid, Przemyslaw Pawelczak, Jethro Tan, Aaron Parks, and Joshua R Smith. 2017. Fast Downstream to Many (Computational) RFIDs. *IEEE INFOCOM 2017 - The 36th Annual IEEE International Conference on Computer Communications* (2017).

A. W. Appel and T. Jim. 1989. Continuation-passing, Closure-passing Style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 293–302. DOI: http://dx.doi.org/10.1145/75277.75303

Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. 2016. 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.

Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* 7, 1 (2015), 15–18.

Michael Buettner, Ben Greenstein, and David Wetherall. 2011. Dewdrop: An Energy-aware Runtime for Computational RFID. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 197–210.

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. 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 (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. DOI: http://dx.doi.org/10.1145/1950365.1950380

Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. In *Proceedings of the Twenty-First International Coonference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 577–589. DOI: http://dx.doi.org/10.1145/2872362.2872409

Alexei Colin and Brandon Lucia. 2016. 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 (OOPSLA 2016)*. ACM, New York, NY, USA, 514–530. DOI: http://dx.doi.org/10.1145/2983990.2983995

Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 133–146.

Marc De Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 1–12.

Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 475–486. DOI: http://dx.doi.org/10.1145/2254064.2254120

Kshitij Doshi and Peter Varman. 2012. WrAP: Managing byte-addressable persistent memory. In *Memory Archiecture and Organization Workshop.(MeAOW)*.

Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 15.

Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 3–14.

Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News*, Vol. 32. IEEE Computer Society, 102.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 48–60. DOI: http://dx.doi.org/10.1145/1065944.1065952

Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free synchronization. In *Proc. of the 20th Annual International Symposium on Computer Architecture*. 289–300.

Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15)*. ACM, New York, NY, USA, 5–16. DOI: http://dx.doi.org/10.1145/2809695.2809707

Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. 2002. 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 (ASPLOS X)*. ACM, New York, NY, USA, 96–107. DOI: http://dx.doi.org/10.1145/605397.605408

Mustafa Emre Karagozler, Ivan Poupyrev, Gary K Fedder, and Yuri Suzuki. 2013. Paper generators: harvesting energy from touching, rubbing and sliding. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 23–30.

Gabriel Kerneis and Juliusz Chroboczek. 2010. Continuation-Passing C, compiling threads to events through continuations. *Computing Research Repository* abs/1011.4558 (2010). http://arxiv.org/abs/1011.4558

Yoonmyung Lee, Gyouho Kim, Suyoung Bang, Yejoong Kim, Inhee Lee, Prabal Dutta, Dennis Sylvester, and David Blaauw. 2012. A modular 1mm 3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*. IEEE, 402–404.

Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 575–585. DOI: http://dx.doi.org/10.1145/2737924.2737978

Kaisheng Ma, Xueqing Li, Shuangchen Li, Yongpan Liu, John Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015a. Nonvolatile processor architecture exploration for energy-harvesting applications. *IEEE Micro* 35, 5 (2015), 32–40.

Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015b. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 526–537.

Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. 2013. 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*. IEEE, 216–224.

Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. 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*. ACM, 1.

Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 401–410. DOI: http://dx.doi.org/10.1145/2150976.2151018

Joseph A Paradiso and Mark Feldmeier. 2001. A compact, wireless, self-powered pushbutton controller. In *International Conference on Ubiquitous Computing*. Springer, 299–304.

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276.

Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2015. Memory Persistency: Semantics for Byte-Addressable Nonvolatile Memory Technologies. *IEEE Micro* 35, 3 (2015), 125–131.

Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. ACM, New York, NY, USA, Article 5, 3 pages. DOI: http://dx.doi.org/10.1145/2618128.2618136

Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011a. 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 (ASPLOS XVI)*. ACM, New York, NY, USA, 159–170. DOI: http://dx.doi.org/10.1145/1950365.1950386

Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011b. Mementos: System Support for Long-running Computation on RFID-scale Devices. (2011), 159–170. DOI: http://dx.doi.org/10.1145/1950365.1950386

Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126. DOI: http://dx.doi.org/10.1145/359340.359342

Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement* 57, 11 (2008), 2608–2615.

Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 204–213. DOI: http://dx.doi.org/10.1145/224964.224987

Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*. ACM, New York, NY, USA, 161–174. DOI: http://dx.doi.org/10.1145/1322263.1322279

Daisaburo Takashima, S Shuto, I Kunishima, H Takenaka, Y Oowaki, and S Tanaka. 1999. A sub-40 ns random-access chain FRAM architecture with a 768 cell-plate-line drive. In *Solid-State Circuits Conference, 1999. Digest of Technical Papers. ISSCC. 1999 IEEE International*. IEEE, 102–103.

Jethro Tan, Przemysław Pawełczak, Aaron Parks, and Joshua R Smith. 2016. Wisent: Robust downstream communication and storage for computational RFIDs. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 1–9.

TI Inc. 2017a. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf. (2017).

TI Inc. 2017b. 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.

Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 17–32.

Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 5–5.

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. DOI: http://dx.doi.org/10.1145/1950365.1950379

Zac Manchester. 2015. KickSat. http://zacinaction.github.io/kicksat/. (2015).

Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. 2011a. Moo: A batteryless computational RFID and sensing platform. *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep* (2011).

Hong Zhang, Mastooreh Salajegheh, Kevin Fu, and Jacob Sorber. 2011b. 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 (HotPower '11)*. ACM, New York, NY, USA, Article 9, 5 pages. DOI: http://dx.doi.org/10.1145/2039252.2039261

Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. 2013. ConAir: Featherweight Concurrency Bug Recovery via Single-threaded Idempotent Execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 113–126. DOI: http://dx.doi.org/10.1145/2451116.2451129