Chain: Tasks and Channels for Reliable Intermittent Programs

Alexei Colin
Carnegie Mellon University, USA
acolin@andrew.cmu.edu

Brandon Lucia
Carnegie Mellon University, USA
blucia@andrew.cmu.edu

Abstract

Energy harvesting computers enable general-purpose computing using energy collected from their environment. Energy-autonomy of such devices has great potential, but their intermittent power supply poses a challenge. Intermittent program execution compromises progress and leaves state inconsistent. This work describes Chain: a new model for programming intermittent devices.

A Chain program is a set of programmer-defined *tasks* that compute and exchange data through *channels*. Chain guarantees forward progress at task granularity. A task is restartable and never sees inconsistent state, because its input and output channels are separated. Our system supports language features for expressing advanced data exchange patterns and for encapsulating reusable functionality.

Chain fundamentally differs from state-of-the-art checkpointing approaches and does not incur the associated overhead. We implement Chain as C language extensions and a runtime library. We used Chain to implement four applications: machine learning, encryption, compression, and sensing. In experiments, Chain ensured consistency where prior approaches failed and improved throughput by 2-7x over the leading state-of-the-art system.

Categories and Subject Descriptors D.4.5 [*Reliability*]: Checkpoint/restart; C.3 [*Special-purpose and application-based systems*]: Real-time and embedded systems

Keywords intermittent computing, energy-harvesting

1. Introduction

Simultaneous innovations in wireless and ambient power technology, coupled with a dramatic decrease in power demands of general-purpose microcontrollers (MCUs), has produced a new breed of *energy-harvesting computing devices* (EHDs). An EHD is a type of computing hardware that powers itself exclusively with energy it collects from its environment, such as ambient [22] or directed [34] radio frequency energy (RF). Energy-autonomy makes EHDs an essential technology for next-generation medical [21, 30], extraterrestrial [40], and IoT [38] applications.

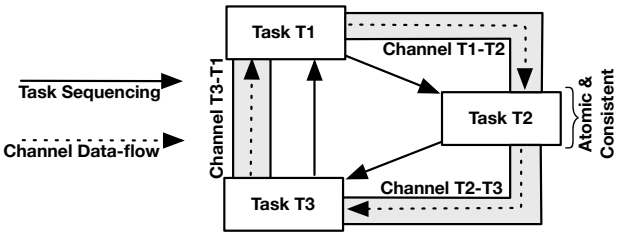


Figure 1: A Chain program. The program has three tasks that execute in sequence and pass data to one another via channels.

A typical EHD buffers the energy it collects in a small energy storage capacitor. Once the capacitor has accumulated sufficient energy, the EHD begins operating and quickly drains the capacitor. When the capacitor is exhausted, the EHD abruptly shuts down. Operation resumes when the capacitor recharges and the device reboots. The charge/discharge cycle is characteristic of energy-harvesting devices [21, 28, 34]. Consequently, EHDs execute software according to the *intermittent execution model* [8, 23, 31]: a program that runs longer than a single charge cycle includes periods of execution perforated by reboots. An intermittent execution *includes the power failures*, in contrast to continuously powered execution that ends at a power failure.

Intermittence is problematic because it can cause a program that is correct in a continuous execution to exhibit unpredictable behavior. When a device experiences a power failure, its volatile state (program counter, registers, SRAM) clears, while its non-volatile state (FRAM, flash) persists. As a result, intermittence can impede forward progress [5, 32] and cause data corruption or crashes [8, 23, 31]. Prior work attempted to address these issues, primarily by checkpointing [23, 32]. Unfortunately, checkpointing is not a viable solution. Checkpointing does not scale to large memory sizes because its time and energy overhead is proportional to the

amount of program state. Checkpointing overheads deprive the application of the EHD's scarce storage and energy. Furthermore, checkpointing is impossible when its energy cost plus the energy cost of application code between checkpoints exceeds the device's energy storage capacitor.

The goal of this work is to make software that runs intermittently reliable without resorting to checkpoints. A programming model that guarantees that an application will execute correctly even when it executes intermittently is *intermittence-safe*. We propose the Chain programming and execution model. Chain is a new programming interface for intermittent EHDs. In a Chain program, the programmer decomposes the computation into a sequence of *tasks*, each of which can perform arbitrary computation and I/O. Chain guarantees that execution progress is preserved at the granularity of tasks. Chain ensures tasks have *atomic* all-or-nothing semantics and that state in volatile and non-volatile memory visible to a task is always consistent. Together these properties make Chain intermittence-safe.

To ensure atomicity and consistency, Chain uses a novel memory access model for an EHD's volatile and non-volatile memory. A task has full access to volatile memory, but Chain requires all volatile variables to be task-local. To convey inputs to and outputs from tasks, the programmer uses Chain's *channel-based* non-volatile memory access model. A task can send a named value to another task (or to a future instance of itself) via a channel dedicated to a pair of tasks (or its own 'self' channel). Chain's channel mechanism guarantees that a task's inputs and outputs are stored in distinct memory locations. The separation of inputs and outputs ensures that a task with any mixture of accesses to volatile and non-volatile memory is arbitrarily restartable after a power failure with *essentially no restoration cost*. From the perspective of the task, its inputs are always immutable, consistent, and available in the channel. Figure 1 shows a schematic representation of a Chain program with three tasks that are sequenced by a task graph and that exchange data via channels.

The Chain execution model implements the semantics of the Chain programming model. To preserve progress, Chain runtime tracks the currently executing task and restarts it on power failure. For atomicity and consistency, in a Chain execution all access to non-volatile memory by the application takes place via a channel I/O operation. Sophisticated data exchange patterns between tasks are expressed through the *synchronized* channel read and *multicast* channel write operations. A synchronized channel read returns the most up-to-date version of a named value among a collection of channels that *may* contain an input value. A multicast channel write allows one task to send a value to many other tasks, without the need for duplicate backing channel storage. The Chain execution model supports abstraction and encapsulation of reusable functionality into *modular task groups*.

We implemented the Chain language as an extension to C and a runtime library. We evaluated Chain by using it to write two real-world sensing applications and two software components, including machine-learning-based activity recognition, temperature sensing and compression, cuckoo filtering, and 1024-bit RSA encryption, the first of which we know to run on an EHD. We show with experimental comparisons to prior work that Chain more effectively preserves progress, keeps data consistent, and significantly improves on run time performance of checkpointing systems with similar goals, namely DINO [23] by 2-7x and Mementos [32] by 10-150x.

To summarize, our main contributions are:

- We propose Chain, an intermittence-safe task-based programming model and channel-based memory model.
- We present an implementation of Chain as an extension to C and a runtime library.
- We use Chain to build applications that perform RSA encryption, LZW compression, activity recognition, and cuckoo filtering on an EHD.
- We show that Chain provides intermittence-safety with *2-150x higher throughput* than checkpointing approaches.

2. Background and Motivation

This section provides background on EHDs, the intermittent execution model, and the state-of-the-art in tolerating intermittence. This section also presents high-level simulation results that motivate Chain in the context of prior approaches.

2.1 Energy-harvesting and Intermittent Execution

EHDs extract energy from their environment and operate according to the intermittent execution model [23]. In contrast to continuous execution, intermittent execution includes periods of computation interspersed with power failures. The failures inherent to an intermittent execution threaten its progress and data consistency. An intermittent execution may fail to make progress, because after a failure control flows back to the application's entry point (*i.e.*, the start of `main`). Intermittence may leave data inconsistent, because in a typical EHD a power failure erases volatile memory but leaves non-volatile memory untouched. The re-initialized volatile

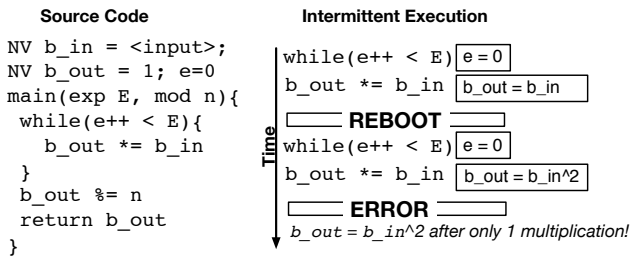


Figure 2: Intermittent execution causes errors. The program computes RSA cyphertext `b_out` from plaintext `b_in` with public key (E,n), all stored in non-volatile memory. The loop index e is in volatile memory. Intermittent execution produces the wrong result with or without a checkpoint of volatile state, because in either case only `e` but not `b_out` is re-initialized on reboot.

memory values may be inconsistent with values persisted in non-volatile memory. A reboot may cause a read-modify-update of non-volatile memory to repeat and produce a memory state impossible in a continuous execution.

Figure 2 shows simplified code for RSA encryption [33] that is correct when executed with continuous power, but that produces incorrect results on intermittent power. The code encrypts plaintext from `b_in` into cyphertext in `b_out`, both of which are in non-volatile memory. The exponent `E` and the modulus `n` are inputs to the algorithm, stored in volatile memory, as is `e`, the loop index. The intermittent execution on the right is interrupted when `e` is zero and `b_out` has been multiplied once. On reboot control flow returns to the top of `main`. Index `e` is again zero but `b_out` is erroneously multiplied a second time. The cyphertext in `b_out` has been irreversibly corrupted as a result of intermittence.

2.2 The High Cost of Checkpointing

Starting with Mementos [32], prior work has resorted to checkpointing [2, 18, 23, 31] to preserve progress and keep data consistent in an intermittent execution. A checkpointing-based approach periodically captures a consistent system state and after a reboot resumes the execution by restoring the state captured in the checkpoint. Note that the intermittence bug in Figure 2 can still manifest with a checkpoint at the end of the loop body, when the checkpoint records only the volatile state [2, 18, 32]. The result of the program will be wrong whenever a reboot happens between the update of `b_out` in non-volatile memory and the checkpoint.

Prior work [23] noted the importance of checkpointing volatile *and* non-volatile state. A volatile-only checkpointing system challenges the user with a lose-lose proposition: either store *all* application state in scarce volatile memory or risk state corruption. Consequently, in either checkpointing system, the checkpoint size may be as large as the *total* amount of state in the program (i.e., not just the volatile execution context). The size of the checkpoints determines the program's energy and time overhead.

To study the overhead of checkpointing, we built a simulator that models intermittent execution in a system with checkpointing (CP) and with no checkpointing (NCP). The workload consists of a fixed number of read-modify-update (RMU) operations to distinct memory locations. For each trial, we break up the workload into tasks such that the size of each task is randomly distributed within +/-10% of the value chosen for that trial. Our device model emulates energy storage, depletion, and full recharge. If energy runs out in the middle of a task, the energy level is replenished, and the task re-executes from the beginning. A CP system saves a checkpoint of the volatile memory into non-volatile memory at each task boundary. Any NCP system must also use non-volatile memory to persist state. We model the worst-case for an NCP, by making each RMU in the NCP workload an access to non-volatile memory. We model the relative cost of volatile and non-volatile memory accesses based on our own measurements of access costs for SRAM and FRAM in a TI MSP430FR5969 MCU on the WISP [34].

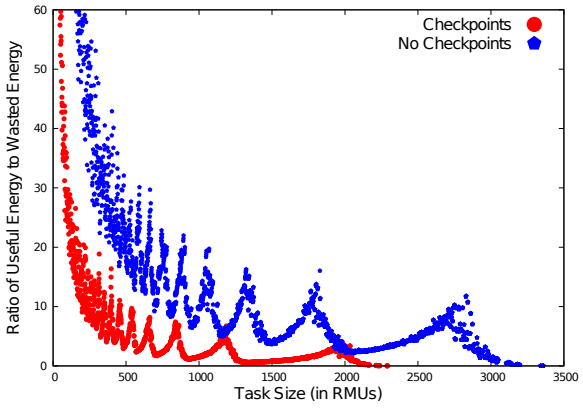


Figure 3: Checkpointing wastes energy. With checkpoints more work is wasted, because a task is less likely to finish restoring, executing, and saving state before energy runs out (left, center). To have any chance of completing at all, tasks must be smaller than they could be without checkpoints (right).

Figure 3 shows the ratio of *useful* to *wasted* energy for the checkpointing and no checkpointing models. Any energy spent on the task is wasted if the task experienced a power failure before it was able to complete (including checkpoint operations in CP). Our simulation shows that across all task sizes, a CP system wastes more work per successful task than an NCP system. The disparity arises because checkpointing overhead increases the likelihood that a task will exhaust energy before completing and waste all the energy that was spent on it. The second observation is that checkpoints prevent progress when the task length grows to around 2200 RMUs. At that point the checkpoint overhead plus the task cost exceeds the energy storage capacity. By eliminating checkpoint overhead, an NCP system allows for a much larger maximum task size, which grants more flexibility in task definition. The simulation illustrates the deficiencies of checkpointing: added overhead, wasted work, and limited task length. The next section describes Chain, a realization of an NCP system that ensures progress and consistency without these checkpointing costs.

3. Chain Programming Model

The Chain programming model uses *task-based control-flow* and a *channel-based memory model* to ensure progress and consistency for intermittent executions without any of the overheads of checkpointing. Task-based control-flow provides a strong notion of progress in the presence of power failures. Channel-based memory allows tasks to exchange data values with guaranteed consistency. Table 1 summarizes the Chain language, and Figure 4 shows an example Chain program that we use to illustrate Chain's features.

Table 1: The Chain Language.

	Feature	Function
Tasks	`task` T, f	Create a task T implemented by function f
	`origin` T	Specify task T to run on first power up
	`self`	Refer to the current task
	`NextTask` T	Transfer control to task T
Channels	`channel` (T_1, T_2), $\{F : type, \ldots\}$	Define channel from task T_1 to task T_2 with a set of fields and specify a type for each field F
	`ChIn` F, T	Read field F from channel $(T$, `self`$)$
	`ChOut` $\{F \leftarrow v\}, T$	Write value v into field F in channel (`self`, T)
	`ChSync` $F, \{T_1, \ldots, T_n\}$	Read the most recent value of field F in channels $(T_1$, `self`$), \ldots, (T_n$, `self`$)$
	`MultiOut` $\{F \leftarrow v\}$, $\{T_1, \ldots, T_n\}$	Write v into field F in channels (`self`, T_1), $\ldots$, (`self`, T_n)
Modules	`module` $M, T_{\text{in}}, T_{\text{out}}$, $\{T_1, \ldots, T_n\}$	Create module M with entry task T_{in}, exit task T_{out} and member tasks $T_1, \ldots, T_n$
	`ModEnter` M, T	Transfer control to the entry task in module M and make task T the successor of module M
	`ModLeave`	Transfer control to successor of current module
	`ModPut` $\{F \leftarrow v\}, M$	Write value v to field F in the input channel of M
	`ModGet` F, M	Read field F from the output channel of M
	`ModIn` F	Read field F from current module's input channel
	`ModOut` $\{F \leftarrow v\}$	Write v to F in current module's output channel

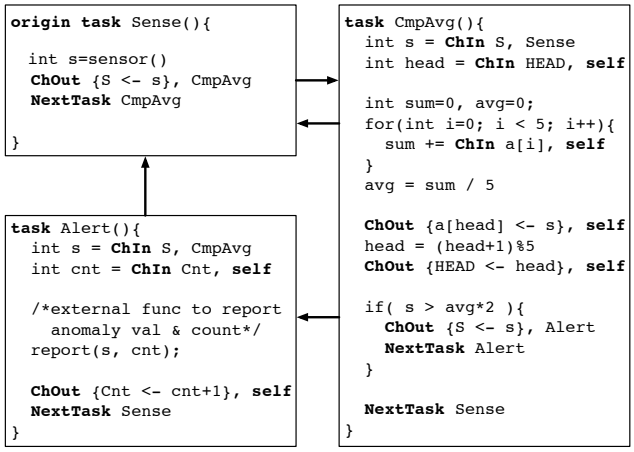


Figure 4: An example Chain program. The program has three tasks, `Sense`, `CmpAvg`, and `Alert`. Sense is the origin task. It reads a sensor and sends the result to `CmpAvg`. `CmpAvg` compares the current sample from `Sense` to twice the average of 5 past samples. If the current sample is greater, `CmpAvg` sends the anomaly to `Alert`. `Alert` counts and outputs anomalies. The code assumes channel fields are statically initialized to zero. All non-channel state (i.e., `int s`) in Chain is task-local.

3.1 Task-based control-flow

A Chain program is written as a collection of *tasks*. The `task` keyword labels a C function as a Chain task. A task can perform arbitrary computation and is free to define task-local volatile variables (e.g. `s` in task `Sense` in Figure 4) and access peripherals (e.g. call to `sensor()`). The set of tasks in an application form a *task graph* that determines how control flows into and out of each task. Each task has at least one *predecessor task* and at least one *successor task*. One task in the graph is marked as the *origin* task and is the task that executes when the device powers up for the first time.

Each task has a single *entry point* and one or more *exit points*. The entry point is at the top of the task function. The programmer syntactically represents an exit point with a `NextTask` statement. Each `NextTask` statement takes the name of another task as an argument and when the statement executes, control is transferred to the entry point of that task. The programmer can include a `NextTask` statement along any control-flow path in a task, terminating that path. The program in Figure 4 has three tasks that are linked into a task graph using `NextTask` statements.

Chain provides the *language-level* guarantee to programmers that tasks are *progress-preserving*. Progress preservation means that control flows from one task to another at a task's endpoint only. Control never jumps discontinuously back to an earlier task and never flows non-deterministically, even in the presence of arbitrarily timed power failures. For example, after a reboot anywhere in the code in Figure 4, the application resumes from one of precisely three locations in the application: the first instruction in `Sense`, `Alert`, or `CmpAvg`. In addition, once execution advances into `CmpAvg`, it cannot enter `Sense` before going through `Alert`; execution follows the task graph. This intuitive control flow behavior cannot be relied on in a system that is not progress-preserving but can experience power failures. In Chain *forward progress* is guaranteed as long as the energy demand of each individual task never exceeds the total energy storage capacity of the device. The programmer can control the energy demands of tasks by choosing the total number of tasks in the application and the amount of work in each task.

3.2 Channel-based Memory Model

A Chain task has unrestricted access to volatile, task-local variables but is not allowed to directly access the system's non-volatile memory. Instead, Chain exposes non-volatile memory to programmers through *channels*. A channel is a named region of non-volatile memory controlled by Chain. Each channel holds a collection of individually-accessible named, typed *fields*. A channel may be declared from any task to any other task using the `channel` statement. A channel is identified by a tuple of its endpoints: the *source task* and the *destination task*. The source task can write a named value into the channel and the destination task can read that named value from the channel. We refer to the channel as a *self-channel* when the source and destination are the same task.

Channels are the sole mechanism to move data into and out of tasks. The programmer passes data through a channel using the `ChIn` and `ChOut` operations. `ChOut` takes a named value and the name of a channel and writes the value into the matching field in the channel. `ChIn` takes the name of a channel and a field name and returns the value that was most recently written to that field in the channel by another task's `ChOut`. For example, with channel (T_1, T_2) declared, task T_1 can `ChOut` its output values for T_2 to use as inputs via `ChIn`. A self-channel (T, T) allows an instance of task T to send values to future instances of itself. Note that the programmer need only explicitly specify the destination task of a `ChOut` statement and the source task of a `ChIn`: the other task in the channel's name is the task containing the `ChIn` or

ChOut. To refer to a self channel, the programmer can use the self keyword in place of a task name. Figure 4 shows how three tasks exchange data using ChIn and ChOut statements. CmpAvg and Alert both use self channels to maintain data across instances.

3.3 Multi-endpoint Channel Communication

While communication between tasks in a simple program may be expressed with basic channel operators presented in the preceding section, communication patterns in complex programs require generalized operators. Chain defines *multicast channel write* for channeling data to more than one destination and *synchronized channel read* for channeling data from more than one source. Figure 5(a) and (b) illustrate multicast write and synchronized read schematically.

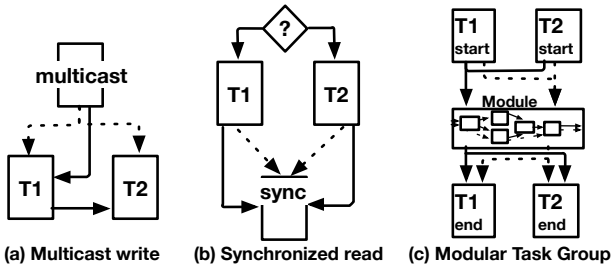


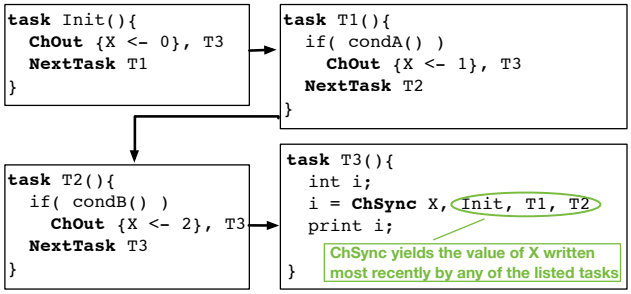
Figure 5: Schematic view of advanced Chain features. Dashed lines are channels and solid lines are task graph edges. (a) A one-to-many multicast channel write to T1 and T2. Multicast enables use of a single, shared channel buffer. (b) A many-to-one synchronized channel read from T1 and T2. Sync enables consuming values conditionally produced in one of many tasks. (c) A modular task group enabling reuse of an encapsulated task sub-graph. T1 and T2 can enter/exit and channel data into/out of the module, which need not refer explicitly to T1 or T2.

3.3.1 One-to-Many Writes with Multicast Channels

Chain allows one task to produce the same value for many other tasks at the same time, using a *multicast channel write*, which is written "MultiOut" in Chain syntax. The semantics of MultiOut is identical to the semantics of a consecutive sequence of normal ChOut operations. Conversely, a ChOut is a special case of the generalized MultiOut. Including both MultiOut and ChOut in the language benefits implementations for two reasons. By retaining ChOut in the language, MultiOut becomes an optional feature that can be omitted by a compliant but minimal implementation of the core language. An implementation that supports MultiOut can leverage it to reduce the channel memory footprint. As we discuss in detail in Section 4, MultiOut allows the Chain implementation to use a single channel buffer for the multicasted values, rather than using multiple channel buffers, one per sequential ChOut.

3.3.2 Many-to-One Reads with Channel Sync

Chain allows one task to consume a value that may come from any one of a set of tasks using a *synchronized channel read*, which is written "ChSync" in Chain syntax. A ChSync takes the name of the value to read and a list of channel names, rather than a single channel name like ChIn. ChSync returns the named value from the channel that *most recently* had a value with that name written into it.



(a) A synchronized channel read always produces the version of a named value most recently written by a task.

```
task T3_no_Sync(){

  int i;
  if( condA() ){
    i = ChIn X, T1 }
  else if(condB()){
    i = ChIn X, T2 }
  else{ i = ChIn X, Init }
  print i;
  exit;

}
```

To consume X without ChSync, T3 must re-evaluate these conditions from T1 and T2.

(b) To find a value without ChSync, T3 must re-evaluate conditions from T1 and T2 which may be impossible after T1 and T2 complete.

Figure 6: ChSync simplifies channel logic. Any of the tasks, Init, T1, and T2, may provide a value to T3. T3 is trying to read the freshest value of X. We show two versions of T3, one with and one without ChSync. The version without ChSync must re-evaluate the conditions from T1 and T2 (in either order, assuming the conditions are exclusive), to decide which of the three task to get the value from. Such logic may require additional values referenced in the conditions to be channeled from T1 and T2 to T3. ChSync is a concise, robust, and efficient solution to this problem.

ChSync is an essential Chain language feature, because a value-consuming task may sometimes need a named value produced by one task and other times need the same named value produced by another task. Similarly, a value-producing task may produce a named value only if some condition is met. Figure 6 shows an instance of this situation. Tasks Init, T1 and T2 each potentially produce a different value for field X. The task consuming that named value, T3 in this example, has no way of knowing which of the tasks, T1 or T2, actually produced a value for X in a particular execution of the program, unless the programmer adds dedicated logic that would generate that information. The extra logic would need to either replicate the code evaluating the condition in the value consuming task (T3), as sketched in Figure 6b, or unconditionally channel the result of the condition evaluation to that task (from T1 and T2). With ChSync such unsustainable logic duplication is avoided.

ChSync eliminates the need for a value-consuming task to reason about which of a set of possible value producers pro-

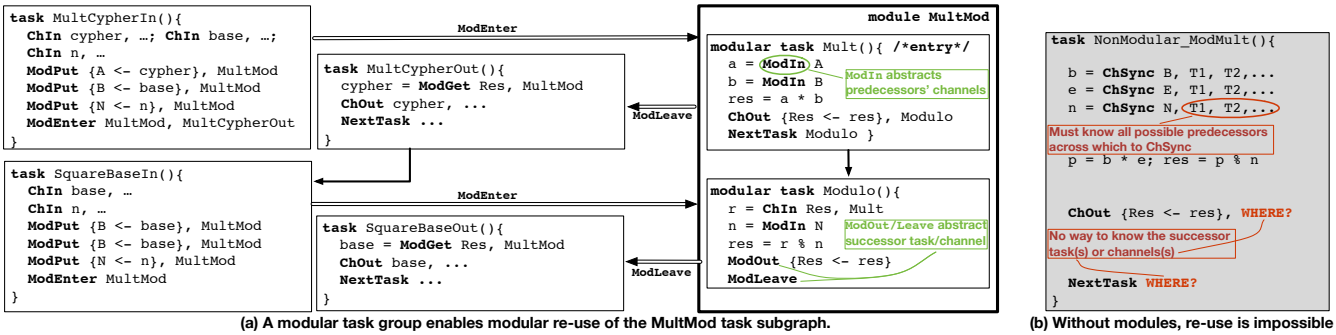


Figure 7: Modular task groups encapsulate code. The `RSA` implementation contains a reusable task graph that computes a product modulo a number. With modules, reused tasks need not explicitly name all predecessors (to get the inputs from) and all successors (to pass the outputs and transfer control to). `NonModular_ModMult` shows why it is impossible to properly encapsulate without Chain's modules.

duced the value it needs. Instead, `ChSync` lets the consuming task observe the value most recently produced by any of the potential value-producing tasks. Yielding the most recently produced value at a `ChSync` is reasonable because there is a global, total order on tasks: yielding any other value would be contradictory to the sequential task order and would be equivalent to reordering executed tasks. Section 4 describes how the Chain runtime ensures that a `ChSync` always sees the latest value.

3.4 Encapsulation of Reusable Functionality

Chain supports encapsulation of reusable functionality with *modular task groups*, or "*modules*". Figure 5(c) schematically illustrates Chain's module support. Like callable functions, modules allow the same code to be executed at many points in a Chain program without requiring the code inside the module to include any information about what those arbitrary points of use are. Without modules, code cannot be parametrized for reuse, because the code would need to include information about each point in the program at which its parameters would be instantiated with values. Specifically, without module support, the channel operations in the code must explicitly refer by name to each task that could produce parameter values for the code and to each task that could consume the result computed by the code. The module interface abstracts tasks that produce parameters and consume results, making it possible to write reusable code that does not contain any information about the sites where it is used.

A set of tasks may be grouped into a module using the `module` keyword. One task is designated as the *entry* task and one as the *exit* task. The encapsulated functionality is the behavior of any execution from the entry task to the exit task. We refer to any task that transfers control to a module as one of the *predecessors* of that module and any task that receives control from a module as one of the *successors* of that module. To transition to a module, a predecessor task uses a `ModEnter` statement. A `ModEnter` takes the name of a module and the name of a successor task and transfers control to the module's entry task. A module's member tasks then execute until the module's exit task executes a `ModLeave` statement. The `ModLeave` statement transfers control to the successor task specified to `ModEnter`.

A module takes its input from its own input channel and produces output into its own output channel. Both channels are allocated as a result of the module's declaration and dedicated to that module. A modules's member tasks can read values from the input channel using `ModIn` and the exit task can write values into the output channel using `ModOut`. Only the exit task is allowed to write into the module's output channel, because it is never safe for more than one task to write into the same channel (cf. Section 3.5). For the same reason, only a predecessor task can put values into the module's input channel. A predecessor task channels values into the module using the `ModPut` statement. `ModPut` takes the name of a module and a named value, and associates the value with that name in the module's input channel. A module's successor can read a value from the module's output channel using a `ModGet` statement. A `ModGet` statement takes a module's name and a field name and returns the value from the module's output channel. To eliminate a potential source of bugs, only a successor task is allowed to access a module's output channel.

Figure 7 illustrates modules in a simplified snippet from our implementation of RSA. The figure shows a partial RSA task graph (left) that in two places computes a product modulo a number. The tasks inside the bold box, `Mult` and `Modulo`, compose a module called `MultMod`. `Mult` is the module's entry task. The `MultCypherIn` task `ModPuts` the two factors and the modulus into the `MultMod` module and enters the `Mult` task. `Mult` uses `ModIn` to receive the inputs. `ModIn` is key to modularity, because it eliminates the need for `Mult` to `ChSync` across all possible predecessor tasks to find its inputs. Such a `ChSync` would break encapsulation by requiring all predecessors of the module to be explicitly enumerated. `Mult` `ChOuts` the multiplication result to `Modulo`, another member of the module, which computes the remainder. `Modulo` uses `ModOut` and `ModLeave` to yield output and transition to the module's *current* successor, which may be either

MultCypherOut or SqBaseOut. ModOut and ModLeave are also essential to reusability, because they abstract the successor. These operations send output and transition to the successor registered in the module by ModEnter. Without modularity support, Modulo would be forced to break encapsulation by enumerating all successors and conditionally choosing among them based on a control value channeled into the task by its predecessor. Figure 7(b) shows an unsuccessful attempt to reuse code for the same calculation through complex control logic instead of Chain's modularity support.

We also note that Chain allows reusing code by encapsulating it into a conventional function and calling that function from within a task. However, this method severely limits the maximum size of a reusable component (i.e., the function). Computation that requires more energy than can be stored in the capacitor on the device cannot be encapsulated into a function, because any task that would call that function would *always* run out of energy before completing. This limit does not apply to a Chain module, because computation encapsulated into a module is decomposable into as many member tasks as necessary.

3.5 Correctness

Chain is correct because its execution model ensures progress and its memory access model ensures that every task is *atomic* and *idempotent*. Together these properties imply a task can arbitrarily lose power and reboot without compromising progress or consistency. Chain's progress guarantee follows trivially from the task-based execution model definition, assuming no task's energy demand exceeds the maximum energy storage of the capacitor on the device. Chain's consistency guarantee follows from the way Chain constrains accesses to non-volatile and volatile state.

3.5.1 Task Atomicity and Isolation

Channel Exclusion for Non-volatile Memory Consistency. Chain channels are strictly, statically subject to *access control*. A task may not write into any channel for which it is not the source and a task may not read from any channel for which it is not the destination. The key property that Chain guarantees about channels is *input/output channel exclusion*: by construction, a single task cannot both read and write the same non-volatile memory location. Channel exclusion occurs trivially for non-self channels. Chain statically requires that when a task executes a ChIn on a channel, the executing task must be the channel's destination. Likewise Chain statically requires that a task executing a ChOut on a channel be the channel's source. For self channels, channel exclusion is enforced by the Chain runtime (Section 4).

Channel exclusion guarantees that a task's accesses to non-volatile memory are *idempotent*. The task's input values are always available in channels for which it is the destination and the task cannot alter those inputs. The task's output values are always written into channels for which that task is the source and it cannot read those outputs. The inability for a task to read a non-volatile value *after* a failure that it wrote *before* a failure precludes visibility of partial or repeated non-volatile data structure updates.

Task-locality for Volatile Memory Consistency. Chain's volatile memory model requires that all volatile variables are task-local. This ensures that a volatile variable is initialized in the task before it is used. Task-locality ensures that a task can arbitrarily reboot from its entry point without losing any volatile state: any path through the task must include a re-initialization assignment. Task-locality of the Chain volatile memory access model ensures that a task's accesses to the volatile memory are idempotent.

I/O in Chain Tasks A Chain task can interact with the world by using I/O operations to manipulate sensors and actuators. The Chain tasks with I/O behave differently from normal Chain tasks. An I/O operation in a Chain task may execute repeatedly as a task re-executes due to intermittence. Repeated executions of an I/O operation may produce different behavior if the repeated operation is non-idempotent. Without careful programming, repeated, non-idempotent input operations can violate task atomicity and repeated, non-idempotent output operations may repeat external behavior that should not repeat.

A non-idempotent input operation violates a task's idempotence, because the input operation may produce a different value each time the task re-executes. Exposing the input operation's non-idempotence to the program is desirable, because the re-execution of an input operation (e.g.,sensor read) should always produce the latest available values. A task containing a non-idempotent input operation is safe *only if* the task does not write to a channel *conditionally*, based on the value produced by the input operation.

In the absence of an input-conditional write to a channel, every re-execution of the task performs writes to the same fields of the same channels. Note that these operations may write different *values* into these fields, depending on the result of the input operation. After the task completes, channels contain the consistent result of its last successful execution.

In contrast, a task that conditionally writes to a channel, depending on the value produced by an input operation may leave channel data inconsistent. The programmer must leverage Chain channels to eliminate this possibility. When control flow involves conditional channel output operations and a task is interrupted, it may leave its output channels in an inconsistent state that may be observed by a successor task. A program that may exhibit this behavior is illustrated on the left in Figure 8. Consider the execution where task T1 reads a positive value from the sensor and is interrupted after writing S but before writing SS. Then, T1 executes from the beginning, reads a negative value from the sensor, and transitions to T2. Task T2 will observe the value of S written by T1, but the value of SS written by T0, which may be mutually inconsistent.

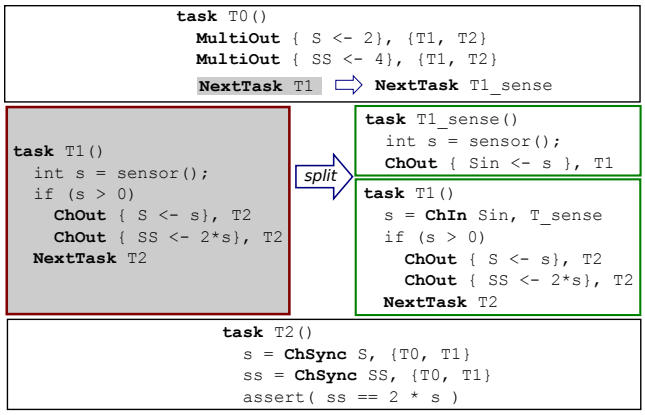


Figure 8: I/O in Chain Tasks. On the left, in task `T1`, an input operation (`sensor()`) is followed by a conditional output into a channel to `T2`. Because the input operation is non-idempotent, the condition may evaluate differently when `T1` is interrupted and re-executed and `T2` may observe `S` written by `T1` and `SS` written by `T0`, which violates atomicity of `T1`. On the right, the program is re-written according to a programming pattern to perform I/O safely: `T1` is split such that the input operation is confined to a dedicated task (`T1_sense`).

The problem arises because control-flow is affected by an input into the task that does not come from a channel and, therefore, is not covered by Chain's atomicity and idempotence guarantees. This observation suggests a straightforward programming pattern, which prevents I/O non-idempotence from affecting control-flow. The pattern for safe I/O is to confine the input operation to a dedicated task that acquires and channels the value to its successor(s). All downstream tasks have to obtain the sensor value from a channel, which ensures their idempotence. Applying the pattern to the above example, we obtain an equivalent program, shown on the right in Figure 8, which uses sensor input safely.

Once an output operation has activated an actuator, its physical effect cannot be undone. Chain does not attempt to provide at-most-once semantics for output operations. We assume that in an application destined for an environment where power is intermittent, the interfaces to actuators, the actuators themselves, or the physical world can tolerate incomplete and repeated output operations from the software.

Chain Tasks are Idempotent and Atomic. Channel exclusion ensures that a task's non-volatile memory accesses are idempotent. Making all volatile variables task-local ensures that a task's volatile accesses are idempotent. Together, these facts guarantee that a task's computation and memory operations are completely idempotent. Idempotent tasks can fail repeatedly and be arbitrarily re-executed from their start without any risk of inconsistency. Furthermore, Chain guarantees forward progress at the granularity of tasks, as long as each task can complete on one capacitor charge. The progress guarantee, combined with the idempotence guarantee implies that Chain tasks are also *atomic*, exhibiting all-or-nothing behavior with respect to observable memory state. We emphasize that Chain imparts idempotence and atomicity guarantees to the application *by construction* at language level. Section 4 describes how our Chain implementation provides these strong guarantees without the need for costly checkpoint/restart mechanisms.

3.5.2 Generality of the Channel-based Memory Model

For any program written in a sequential language with a conventional memory model, e.g. C, there is a program written using Chain task and channel abstraction that has equivalent behavior. [1] We define *program behavior* as a sequence of observed variable values, similar in spirit to the model in [24]. To simplify the discussion, we only discuss operations that manipulate non-volatile memory. This simplification is reasonable, because Chain-specific operations (`ChOut` and `ChSync`) only manipulate non-volatile memory, and there is a trivial correspondence between volatile variable manipulations in a conventional execution and in a Chain execution. Without loss of generality, we consider `ChSync`, but not `ChIn`, because `ChSync` is a generalization of `ChIn`, and `ChOut`, but not `MultiOut`, because `MultiOut` can be expressed in terms of multiple `ChOut` statements.

In a conventional execution, a memory write assigns a value to a named variable, and a memory read observes the value most recently written to a named variable. The sequence of variable values observed by the reads in an execution defines the program behavior. In a Chain execution, a `ChOut` operation assigns a value to a named field in a specified channel. A `ChSync` operation observes the most recent value of a named field in a collection of specified channels. The sequence of variable values observed by the `ChSync` operations in an execution defines the program behavior. We show that for all conventional executions, there exists a Chain execution that has the same behavior. We start from an arbitrary execution trace generated by an arbitrary conventional program and we construct a Chain execution trace that has the same behavior.

We consider an arbitrary conventional execution trace, E_{conv}, defined by a sequence of operations, $\mathsf{write}_i(M[v], x)$ and $\mathsf{read}_j(M[v])$, where indexes i and j denote positions in the sequence and the operations are respectively a write of value x to memory location v and a read from memory location v. We begin constructing a Chain execution trace, E_{Chain}, by initializing it with a copy of E_{conv}. At this point, E_{Chain} is not yet a valid Chain execution trace, because it contains direct accesses to non-volatile memory. We assume an arbitrary assignment of operations to tasks in the Chain execution and define a convenience function, $\mathsf{Task}(c)$, that reports the task containing operation c. In E_{Chain}, we replace each $\mathsf{write}_i(M[v], x)$ with a `ChOut` of value x to field $F[v]$ from task $\mathsf{Task}(\mathsf{write}_i)$ to each subsequent task that reads

[1] We argue about the generality of uninterrupted execution, because a conventional program does not complete in the presence of intermittence.

that value before it is overwritten by another, later write. That is, the ChOut associated with write_i accesses channels

$$\{(\mathsf{Task}(\mathsf{write}_i), \mathsf{Task}(\mathsf{read}_j)) \mid j > i \text{ and } \nexists \mathsf{write}_k,\ i < k < j\}.$$

Similarly, we replace each $\mathsf{read}_j(M[v])$ with a ChSync of field $F[v]$ from each preceding task that wrote to that field. That is, the ChSync associated with read_j accesses channels

$$\{(\mathsf{Task}(\mathsf{write}_i), \mathsf{Task}(\mathsf{read}_j)) \mid i < j \text{ and } \exists \mathsf{write}_i\}.$$

Note that in the case of ChSync, we consider *all* prior writes, because, as we note in Section 3.3.2, a task that reads a field using ChSync cannot know which ChOut had produced the observed value and must rely on ChSync to retrieve the most recent value. We next argue that the constructed E_{Chain} has the same behavior as E_{conv}.

To show behavioral equivalence, we show that the sequence of variable values produced by the memory read operations in E_{conv} is the same as that produced by the ChSync operations in E_{Chain}. A $\mathsf{read}_j(M[v])$ in E_{conv} observes value x written by the *unique* $\mathsf{write}_i(M[v],x)$ that most recently precedes that read — i.e. $i < j$ and $\nexists \mathsf{write}_k(M[v],y)$ for $i < k < j$. The write is unique, because a conventional execution is sequentially totally ordered. In the constructed E_{Chain}, each ChSync accesses a *collection* of channels, each of which may contain a value for field $F[v]$. As observed by the ChSync, the channels are not sequentially ordered, unlike writes with reads in E_{conv}. However, by the semantics defined in Section 3.3.2, a ChSync on field $F[v]$ observes the value of the ChOut to $F[v]$ that was the *most recent* according to an explicitly maintained timestamp. By our construction, this ChOut operation corresponds to $\mathsf{write}_i(M[v],x)$ in E_{conv}, where $i = \max\{k \mid k < j \text{ and } \exists \mathsf{write}_k(M[v],\cdot)\}$. This correspondence implies that the value of the ChOut operation to field $F[v]$ is also x. Extending this argument to all read operations in E_{conv} and the corresponding ChSync operations in E_{Chain}, we conclude that the sequence of variable values produced by the execution traces is identical, implying that the behavior of programs producing these traces is equivalent.

4. Chain Implementation

We implemented the Chain programming language primitives described in Section 3 using a combination of compile-time macros and a runtime library. The compile-time features declare and allocate memory for tasks and channels. The runtime features implement task sequencing and channel operations. Figure 9 depicts the state that Chain uses internally to implement execution context, tasks, and channels.

Hardware Assumptions. Our implementation makes few assumptions about the underlying energy-harvesting hardware. We assume some (but not necessarily all) memory is non-volatile. This assumption matches existing energy-harvesting devices (e.g., the Wireless Identification and Sensing Platform (WISP) [34]). Chain runtime implementation assumes

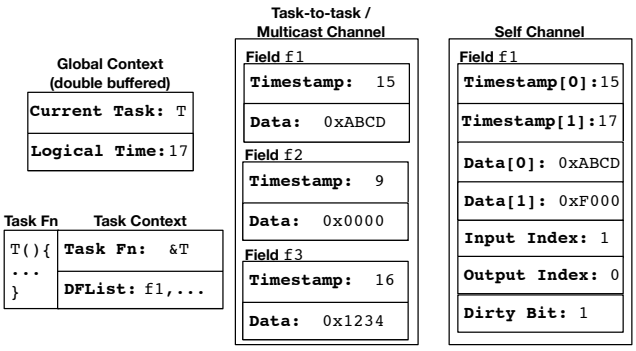


Figure 9: State used in our Chain implementation. The double-buffered execution context tracks time and the current task. The task context keeps a pointer to the task code and a "Dirty Field List" (DFList) containing updated fields in the task's self channel. Task-to-task channels and multicast channels have the same representation and each of their fields contains a timestamp and data. A self channel field contains two timestamps and data buffers, one for input and one for output. A self channel field tracks which timestamp and data buffer is input and which is output using the input/output indices; the dirty bit is set if the field was updated.

single-word writes to non-volatile memory are atomic. This assumption is reasonable for the single-cycle 16-bit microcontrollers and the FRAM memory technology common in energy-harvesting hardware. While atomicity of memory writes given arbitrarily-timed power failures is not explicitly guaranteed by manufacturers, we have never observed partially written words in non-volatile memory. For the runtime library, the compiler must not be allowed to re-order writes to non-volatile memory. This requirement does not concern application code, because all Chain operations are sequencing points.

4.1 Tasks

A task is composed of a *task function* that contains its code and a *task context object* that contains its runtime state. A task function is a C function with no arguments and no return value. A task function can contain calls to arbitrary C code (i.e., legacy/third-party code), but Chain's consistency guarantee does not extend to any such code that writes to non-volatile memory. A task's context consists of a pointer to its function and state related to maintaining its self channel, which we describe in Section 4.3. Each task object is statically allocated and initialized in non-volatile memory.

4.2 Task Sequencing

The Chain runtime maintains a non-volatile *global execution context* that stores the pointer to the current *task execution context* and the current logical time; both objects are depicted on the left in Figure 9. The NextTask control-flow directive updates the current task context pointer in the global execution context to point to the context object of the next task. Chain must atomically update the multi-word global execution context despite intermittence. Atomicity is ensured by *double buffering* the global execution context and indirecting

accesses to it through a pointer. While the current global execution context is in one buffer, the `NextTask` routine sets up the updated context in the other buffer. To commit the transition, Chain sets the context pointer to point to the updated buffer atomically using a single instruction. After a reboot, the runtime transfers control to the task pointed to by the current global execution context, which is retrieved from non-volatile memory. On each task transition, but not on reboot, the runtime increments the current *logical time* in the global execution context, which clocks application progress and is used to implement channel operations described in the next section.

A key property provided by the Chain language implementation is that all state visible to a program after a task transition is *exactly the same* as after a reboot. This property frees Chain from the need for costly restore operations after reboots that are characteristic of checkpointing systems. After a transition, the Chain runtime invokes a *task prologue* that idempotently sets up a task's channel structures. Section 4.3 provides a detailed explanation of channel setup in the prologue. Importantly, Chain guarantees that the prologue completes exactly once for each task transition. To guarantee a single successful prologue execution the runtime saves the logical time at the end of the prologue into the task context. Chain only executes the prologue if the current time exceeds the saved timestamp. After the prologue, Chain jumps to the task's function entry point.

4.3 Channels

In our implementation of Chain, channels are defined statically and accessed dynamically. A channel is defined by specifying its two end-points and a set of named, typed data fields. We refer to a channel between different tasks as a *task-to-task* channel to distinguish it from a *self* channel (Section 3.2). A channel's data field may hold a scalar value or an array value. Chain implements array fields as a collection of scalar fields, each of which can be referred to by an index. The fields of each channel are specified by the programmer at compile time using syntax defined in the Chain library header.

Each channel definition translates to a C structure type. Internally, each member field of a channel is a nested structure. The nested field structure in a task-to-task channel has a buffer to hold the data value of the channel field and a member for channel metadata. The channel metadata field consists of a last-modified timestamp, which is used by the `ChSync` implementation described in Section 4.6.

The implementation of a self-channel is different from a task-to-task channel. A self-channel field has *two* buffers for data values — one for incoming and one for outgoing data — and a member for metadata. The duplicated data buffers in a self-channel field are used to implement the channel exclusion principle introduced in Section 3.5. The metadata field of a self-channel holds the timestamp of its last update and the state Chain needs to decide which data buffer is its input and which is its output (Section 4.4).

A channel declaration statically allocates a channel as a non-volatile C `struct` (in FRAM). A channel's symbol name is the concatenation of the names of the channel end point tasks. Consequently, `ChIn` and `ChOut` can resolve a channel's name using tasks' names at compile time.

4.4 `ChIn` and `ChOut`

A task writes a value into a field of a channel using `ChIn` and reads a value from a field of a channel using `ChOut`. These directives resolve the channel's memory location at compile time by concatenating the names of the source and destination tasks into the channel structure's symbol name. A `ChOut` to a field in a task-to-task channel writes a value into the field's data buffer and sets the field's last modified timestamp to the current logical time (from the global execution context). A `ChIn` from a task-to-task channel's field returns the value in the field's data buffer.

A self-channel field has an input and an output data buffer. Its field metadata consists of two timestamps, an *output buffer index*, an *input buffer index*, and a *dirty bit*. A `ChOut` to a field of a self-channel writes the given value to the data buffer identified by the output index, sets the dirty bit, and adds the field offset to a *list of dirty fields* in the task object. A `ChIn` from a self-channel returns the contents of the data value buffer identified by the input index. On the next transition the roles of the input and output value locations are reversed for all fields that were marked as dirty. This takes place in the prologue routine that runs once after a task transition, as explained in Section 4.2. For each field in the dirty field list with its dirty bit set, the prologue does an atomic *swap-and-clear* that *swaps* the input index with the output index and *clears* the dirty bit. We pack the index and dirty bits into a single 16-bit word, making the atomic *swap-and-clear* a single write instruction. Even if the prologue executes repeatedly, each field undergoes exactly one swap, because a swap only occurs if the dirty bit is set and the dirty bit is cleared by the swap.

4.5 `MultiOut`

With a `MultiOut` primitive a task can channel a value to multiple recipients using as much memory as a single task-to-task channel. A *multicast channel* is like a task-to-task channel in its compile-time declaration and field structure. The channel's name is the concatenation of its source with a *destination ID* that uniquely identifies its destination list.

A `MultiOut` statement can refer to a multicast channel only if its source task is the calling task, because `MultiOut` constructs the channel's name using the name of the calling task. The `ChIn` and `ChSync` statements use the name of the calling task and the multicast channel's destination ID string to refer to the channel. Our prototype does not prohibit reads from a multicast channel by tasks that are not members of the destination set. This limitation may be unintuitive, but does not jeopardize Chain's correctness.

4.6 ChSync

A ChSync operation reads a value that may reside in one of a set of channels. The ChSync primitive accepts a field name and a set of sources. A source can be a task name, self, or a multicast destination ID. At compile time ChSync resolves each source into a channel name and locates the field's last-modified timestamp by the field's name. At runtime, ChSync compares the timestamps associated with the fields and returns the data value of the field with the latest timestamp.

4.7 Modular Task Groups

A *modular task group*, or a "*module*", encapsulates a group of tasks for re-use. A module contains an entry task, an exit task, an input channel, and an output channel. A module's input and output channels are implemented as augmented task-to-task channels. The input channel stores the name of the module's successor task. ModEnter saves the name of the successor task into the input channel and transfers control to the entry task. A module's entry task's name is constructed at compile time from the module's name. ModPut and ModIn translate into ChOut and ChIn on the module's input channel. ModGet and ModOut translate into ChIn and ChOut on the module's output channel.

Any task in the module can ModIn from the input channel, because the input channel symbol name does not include the calling task's name. The output channel's name includes the name of the exit task, allowing only the exit task to ModOut to the output channel. Our prototype implementation of modules has a limitation that is not fundamental to Chain's design. We do not check that only a module's member tasks access its channels, and the programmer is responsible for correctly using ModIn, ModOut, ModPut, ModEnter, and ModGet.

4.8 Release

Our implementation of Chain is distributed as a static library with headers. It amounts to 644 lines of C code, which compiles to 412 bytes or 0.6% of program memory on the WISP platform. The source code is available at http://intermittent.systems.

5. Applications

We used Chain to implement two complete applications and two software components, each representative of a practical domain with varied control flow, compound data types, and complex data structures. We built our systems using the WISP5 energy-harvesting platform, which has a TI MSP430FR5969 16-bit MCU with one core clocked at 8 MHz, 2KB of RAM, 64KB of non-volatile FRAM, an accelerometer, and an RF energy-harvesting power system [34]. In addition to implementing the applications using Chain, we also implemented each using DINO [23] and Mementos [23], which are state-of-the-art runtime systems for intermittence. We used the publicly released DINO implementation. We wrote two variants for Mementos. One variant, Mem-NV, uses volatile and non-volatile memory, but may experience data corruption because Mementos does not keep non-volatile memory consistent. The other variant, Mem-V, restricts mutable state to volatile memory, which is kept consistent by Mementos, but limits the total size of the program state to the small capacity of the volatile memory. Section 6 evaluates these systems and Chain in correctness, performance, memory profile, and developer effort.

Activity Recognition (AR). AR is a machine-learning physical activity classification system used in prior work [23]. AR collects accelerometer samples into a sliding window and filters out samples below a noise threshold. AR converts the 3-axis samples into feature vectors and classifies the window as moving or stationary using a nearest neighbor classifier. After classifying, AR updates the classification statistics for each class and stores them in non-volatile memory for later inspection. AR trains its model by having the user generate reference activity for each class. In a correct execution, the classification statistics must be mutually consistent: the class counts must sum to the total count.

Cold-Chain Equipment Monitoring (CEM). A CEM system continuously monitors a temperature-controlled environment (e.g., vaccine storage), logging temperature over time. Our CEM system collects a stream of temperature sensor readings and compresses them using LZW compression [39] to maximize the capacity of the log. The compressed stream is recorded in non-volatile memory for later decompression and inspection. In a correct execution, the resulting log is a valid, LZW-compressed data stream.

Data Encryption (RSA). This application encrypts a message using RSA [33]. The public key of up to 2048 bits (configurable at compile time) is stored in non-volatile memory, and can be changed after deployment. To the best of our knowledge, ours is the first RSA implementation on an energy-harvesting device using such a strong (large) key. Our implementation thus enables an energy-harvesting device to securely communicate with any base station without the need to share a secret ahead of time.

Cuckoo Filtering (CF). A cuckoo filter is a general-purpose data structure that approximately encodes set membership and supports element deletion. This data structure is well-suited for filtering out redundant samples from a sensor. Like a Bloom filter, a cuckoo filter may return a false positive but not a false negative when queried for a value. Our CF implementation inserts a fixed-length sequence of pseudo-random values into a large filter. CF then looks up the sequence of values in the filter. In a correct execution, the count of affirmative lookups matches the sequence length.

6. Evaluation

We compare Chain to state-of-the-art runtime systems in terms of correctness, performance, memory profile, and developer effort. We deployed each application described in Section 5 on the WISP [34] and ran it on harvested-

Table 2: Correctness of observed application output.
Legend: ✓= correct, ✗= incorrect, ∗= correct if application fits in RAM.

App.	Chain	DINO	Mem-NV	Mem-V
AR	✓	✓	✗	∗
CEM	✓	✓	✗	∗
RSA	✓	✓	✗	∗
CF	✓	✓	✗	∗

energy. The input to AR was generated by the accelerometer as we flipped the orientation of the WISP from vertical to horizontal throughout the experiments. The AR model was trained in the vertical and horizontal orientation for each class, respectively. The input to CEM originated from the temperature sensor without any deliberate manipulation of the surrounding temperature. The plaintext for RSA was a fixed 11-byte string stored in non-volatile memory. The input values in CF were produced by a simple pseudo-random-number generator with a fixed seed. We used the Energy-interference-free Debugger (EDB) [9] to record the output without affecting the energy state of the device. In our lab setup the WISP harvested energy from a ThingMagic Astra-EX RFID reader from a distance of 20 cm (10 cm for CEM).

6.1 Correctness

An application may produce an incorrect result on an intermittently-powered platform if the runtime system does not guarantee memory consistency (cf. Section 2.1). Table 2 summarizes the outcome of running each application on harvested energy. Applications written in Chain and DINO always produced correct output. This result follows from the memory consistency guarantee made by these systems.

The Mem-NV version of every application either returned an incorrect output or failed to complete on at least one trial. AR generated percentages for moving and stationary classes that did not add up to 100%. CEM entered an infinite loop or produced compressed text with more compressed indexes than uncompressed samples, which violates an invariant in LZW algorithm. RSA produced undecryptable cyphertext. CF reported false negatives to membership queries. Mem-V guarantees correctness for an application only if its state (stack and global variables) completely fits within *volatile* memory. We discuss the implications of restricting state to volatile memory in Section 6.3.

6.2 Performance

We ran each application in Section 5 on harvested energy and measured the time it took to complete a fixed amount of work. The amount of work was defined by application-specific parameters that control the number of classifications in AR, the size of the compressed log in CEM, the size of the plaintext (equal to key size) in RSA, and the number of buckets in the cuckoo filter in CF. We configured AR to 128 classifications, CEM to a dictionary of 280 entries, RSA to 128-bit keys, and CF to a filter with 256 buckets. These are the

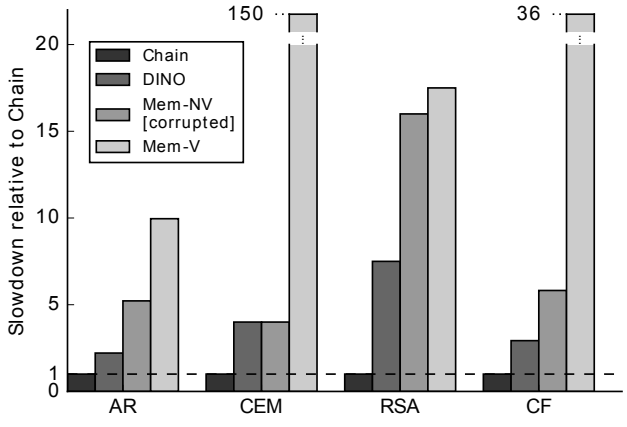


Figure 10: Application performance with Chain and state-of-the-art.

largest workloads that Mem-V can handle due to its memory size limitation. In our trials each application completed its workload within several seconds. We present detailed results from a representative trial run in Figure 10.

Figure 10 shows the slowdown relative to Chain, defined as a ratio of the respective execution times. Chain outperforms DINO, the state-of-the-art, by 2x to 7.6x. We include Mem-NV performance results for completeness, but emphasize that its output is *incorrect* in trials that generate any output at all (cf. Section 6.1). Mem-V running time is one or more orders of magnitude above the alternatives. Mem-V spends most of the time in saving and restoring its disproportionally large checkpoints, each of which must include *all* program state. Applications run faster with Chain, because Chain does not use checkpoints. Chain eliminates the cost of saving and restoring checkpoints as well as the work wasted on checkpoints that are started without sufficient energy to complete them.

6.3 Memory Profile

We characterized how Chain utilizes memory and showed that Chain is always comparable with DINO and Mementos, and in some cases Chain makes better use of memory. The memory footprint of the runtime is not prohibitively high for any of DINO, Mementos, or Chain. Chain's footprint was 412 bytes (0.6% of memory on the WISP), compared to 582 bytes for DINO (30% larger than Chain) and 340 bytes for Mementos (21% smaller than Chain). The most significant memory cost for all three systems is the non-volatile memory consumed by checkpoints and channels. Table 3 summarizes the non-volatile memory footprint that we collected from the application binaries. For Chain, the footprint consists of the channel memory buffers. For checkpointing-based systems (Mem-V, Mem-NV, and DINO) the footprint includes the space reserved for a *double-buffered* checkpoint, as well as non-volatile variables in Mem-NV and their versioned copies in DINO. Mementos and DINO both conservatively use a checkpoint buffer that can accommodate a checkpoint of all of

RAM (i.e., 2KB), amounting to double-buffered checkpoint storage of 4KB.

The data show that Chain's memory use is sometimes less than, and sometimes more than that of the other systems for the same applications. Chain uses less memory than checkpointing systems when the size of channel state is less than the size of checkpoints and versions. Chain allocates exactly as much memory as it needs, since its channel declarations are available at compile time. In cases where Chain uses more non-volatile memory (CF, CEM), the overhead is due to replication of data in channels. In these cases, Chain trades non-volatile memory consumption for the often disproportionately large improvement in throughput and energy efficiency that comes with eliminating checkpoints.

We also analyzed volatile memory consumption. Volatile and non-volatile memory usage affect deployment cost disproportionately. Non-volatile memory is orders of magnitude cheaper than volatile memory, e.g., the largest MCU in the MSP430FR family has 128 KB of FRAM but only 2KB of SRAM. Chain does not change a system's volatile memory consumption. DINO requires enough volatile memory to hold versioning data for non-volatile variables on the stack before adding them to a checkpoint. Mem-V requires enough volatile memory to hold all program state, making it impossible to use for non-trivial applications on some MCUs.

Mem-V's exclusive dependence on volatile memory compromises application performance, constrains the choice of MCU, and limits the maximum distance to the energy-source. For example, in CEM the compression rate is a function of the size of LZW dictionary. The 2048-bit RSA would require an MCU with at least 2KB of SRAM. However, purchasing more RAM is not a solution if the RAM contents is part of every checkpoint, as it is in Mem-V. Once the checkpoint becomes large enough, the energy required to reach the next checkpoint will exceed the energy available during one capacitor charge-discharge cycle and application progress will halt. We directly observed this failure mode for our CEM system at $>$10 cm from its energy source, where Mem-V stopped working completely.

To illustrate the value of Chain's multicast channel feature, we calculated the memory saved in Chain applications by using multicast channels instead of standard channels. As explained in Section 4.5, our implementation of the multicast channel shares one memory buffer between all destination endpoints. Table 3 shows that using multicast channels is valuable in all of our applications and using multicast channels on average consumes less than half as much memory as a collection of standard channels would use to serve the same purpose.

6.4 Developer Effort

To implement an application in Chain, the developer decomposes it into tasks and connects the tasks with control flow statements and channel statements. A decomposition follows naturally from a modular application design. A loop may need to be converted into a task with the loop body and a transition to itself. To reuse code in a decomposed application, with moderate effort tasks can be encapsulated into modules (cf. Section 3.4). Decomposing into Chain tasks is similar to placing DINO boundaries. We list the number of tasks and modules in Chain and task boundaries in DINO for each application in Table 4.

We compare the amount of additional code each application requires in both Chain and DINO implementations in Table 4. For Chain, lines are categorized into channel declarations, task transition statements, and writes to and reads from channels. Across our applications Chain code is larger than DINO code on average by 42%, of which 60% are straightforward declarations of tasks and channel fields. A task declaration specifies the name of the task and its implementation function, and a channel declaration specifies the channel's endpoints and the types of each of its fields.

Although they burden the programmer, explicit specifications of channels are self-documenting and provide the necessary information to statically check the usage of `ChIn`/`ChOut` statements for correctness. An implementation alternative could trade off the advantage of explicit specifications in favor of reducing the amount of code required. Such an implementation could infer channel declarations from their usage in `ChIn`/`ChOut` statements. A graph of inter-task data exchange could then be constructed from the `ChIn`/`ChOut` statements, and a channel allocated per each edge. The type of channel fields can be inferred from the type of values that are being written and read from the channel, with the exception of sizes of array fields, which would be specified by explicit type declarations. Channel field, type, and structure inference is an especially compelling direction for future work on Chain.

Table 3: Non-volatile memory usage (KB) with Chain and state-of-the-art, measured when deployed on TI MSP430FR5949 MCU that features 2 KB of SRAM (volatile) and 64 KB of FRAM (non-volatile). Right-hand columns show the benefit of Chain's channel multicast feature: the number of multicast operations used, the average number of destinations, and the memory saved on multicast channels relative to an equivalent set of per-task channels with `ChOut` statements.

App.	Memory Consumption (KB)				Multicast Benefit		
	Chain	MemV	MemNV	DINO	Ops.	Avg. Dests.	Savings
AR	2.5	4.1	4.2	4.2	1	2	50%
CEM	9.4	4.1	5.8	5.8	5	2.2	37%
RSA	4.9	4.2	4.4	4.4	11	3.6	90%
CF	16.2	4.1	4.6	4.6	5	3.4	73%

Table 4: Size of application implementations in Chain and DINO.

App.	Chain					DINO	
	Tsk. / Mod.	LOC				Tsk. Bnd.	LOC
		Decl.	Flow	Chan.	Tot.		
AR	11 / 0	61	19	49	519	8	435
CEM	12 / 0	82	19	63	412	13	264
RSA	20 / 2	103	28	119	831	34	644
CF	14 / 1	109	20	74	432	13	262

7. Related Work

Related prior efforts addressed the progress [2, 18, 25, 32] and data consistency problems of intermittence using checkpoints [18, 32], versioning [23, 31], and duty-cycling [2, 5]. We also discuss work on idempotent compilation [11, 12, 41], non-volatile memory management [7, 10, 13, 26, 27, 37], transactions [14], and actor models [1, 3, 4, 20, 29].

7.1 Checkpointing

Mementos [32] first identified the need to preserve state across failure periods to execute long-running programs on energy-harvesting devices, like the WISP [34]. Subsequent work [18, 25] explored the space of volatile state checkpointing with architecture and circuit support. All of these efforts preserve progress by retaining volatile execution context. Unfortunately, volatile-only techniques are insufficient to ensure correctness, since non-volatile memory accesses can give rise to inconsistency [31], as we witnessed in our experiments in Section 6. Based on this observation, DINO [23] used task boundaries and non-volatile versioning to keep all of memory consistent.

The main difference between Chain and these prior approaches is that Chain preserves progress and consistency without the high cost of checkpointing or versioning. Chain eliminates the time and space overheads of checkpointing. Chain also enables strictly static memory allocation, not requiring checkpoints, the size of which varies with call stack depth and working set.

7.2 Duty-cycling and Scheduling

Other prior work used duty-cycling to tolerate intermittence. Hibernus [2] uses energy in a system's existing decoupling capacitor to preserve a limited amount of critical state, then drops to an extreme low power state just before a power failure. Hibernus assures limited consistency and progress, as it collects some volatile state before a failure, but the size of that state is limited by the size of the decoupling capacitor (which is typically very small). DewDrop [5] is a scheduler that analyzes available energy and the expected cost of executing code regions. DewDrop executes a code region if it expects to complete the code region without failing. DewDrop compromises on progress and correctness, because it does not persist any volatile state.

The principle difference between Chain and these approaches is that Chain assumes failures are inevitable and uses idempotence to make tasks running at full duty-cycle robust to arbitrary failures. In contrast, these approaches spend long periods of time sleeping, using their sparse duty-cycle to execute only when failure is unlikely.

7.3 Idempotent Compilation

Prior work on *idempotent processing* [11, 12, 41] aimed to make bounded code regions arbitrarily restartable. This line of inquiry revolves around the idea of extracting idempotent regions from a sequential program and leveraging their idempotence for fault recovery.

The similarity between this work and Chain is the goal of idempotence: *idempotent regions* for this prior work, and *tasks* for Chain. There are several important differences. First, this prior work assumes a continuously powered execution in which consistency, but not progress can be compromised. The difference in fault model is critical – a power failure ends an idempotent region's execution for this prior work, losing all associated state and progress. Chain instead embraces power failures of an intermittent execution through idempotence and accommodates arbitrary power failures gracefully. Second, this prior work focuses on volatile memory, limiting the category of faults from which they can recover to those that affect volatile storage. By contrast, Chain targets systems with mixed-volatility memory and must ensure that volatile variables are recomputable and non-volatile memory remain consistent across failures. Third, this prior work explicitly targets surviving hardware value faults, recovering from concurrency errors in multi-threaded software, and handling exceptions. Chain differs greatly in purpose, focusing on providing a new, reliable *programming and execution model* for intermittence, rather than addressing these existing reliability problems.

7.4 Mixed-volatility Memory Consistency

A raft of prior work [7, 10, 13, 26, 27, 37] addressed memory consistency in the presence of mixed volatility and failures. Memory persistency [27] proposed a memory-fence-like mechanism for ensuring the consistency of persistent data. The work also studied relaxations of persistent data consistency, framing all of their work in the context of explicitly multi-threaded, parallel programs. NV-heaps [7] presented new language and runtime support to ensure non-trivial data structures stored in non-volatile memory remain consistent. The work restricted pointer use and controlled some data flow to provide strong correctness guarantees. Other work provided whole system persistence [26] and file-system-like [10, 13] correctness for non-volatile state.

Chain is similar to these systems because Chain, like all of these systems, aims to keep the persistent storage on a device consistent in an adversarial execution environment. Chain is fundamentally different from these systems because it targets devices in which failures are the common case and consequently costly restart actions (like checkpointing or scanning a log) are unacceptable. Instead, Chain enforces *channel exclusion*, making tasks idempotent to keep non-volatile memory consistent, despite arbitrary failures.

7.5 Transaction Processing

The fault tolerance demands of an intermittent system are superficially similar to the fault tolerance demands of a transaction processing system (TPS) [14]. However, the problem for intermittent systems is fundamentally different in domain and purpose from TPS. A TPS expects a transaction (e.g.,

a database update) to complete atomically, but allows it to occasionally fail, as long as the database remains consistent. In contrast, a Chain task on an intermittent computer is interrupted by power failure repeatedly, but eventually completes – i.e., TPS-like failure is not an option. Some prior work such as Spheres of Control [14], transaction chopping [35], and geo-distributed storage [42] decomposed transactions into chains of sub-transactions, breaking the atomicity of the larger transaction. In constrast, in Chain, each task is independently defined and atomic. Chains of tasks are loosely analogous to chained transactions, but in Chain, by construction, no work ever needs to be "rolled forward" [14] or undone on restart.

Transactional memory (TM) in hardware [17] and software [6, 16, 36] also aims to make code atomic. Unlike Chain, TM primarily targets parallel systems, and is does not exist for intermittently-powered hardware. There is also a superficial similarity between Chain, which requires all code to be in some task, and TCC-like memory models [15, 19] which require all code to be in some transaction.

7.6 Actor Models of Computation

Chain defines a computational model and a language for intermittent systems as Actors did for concurrent distributed systems [1]. The Actor model has inspired languages for computationally constrained devices, such as embedded systems [3], sensor nodes [20], and spacecraft [4], but not for intermittently-powered energy-harvesting devices. The Chain model differs in purpose from the Actor model as Chain targets intermittence. Actors represent concurrency, are created dynamically, and carry out computation only in response to messages. In contrast, Chain tasks represent a sequential program, are defined statically, and execute in a fixed order specified by the task graph. Despite their differences, both models encounter some of the same challenges. For example, modules in Chain (Section 3.4) address the same composition challenge that motivated "external actors" [1] and, more recently, flexible interfaces between actors in the form of dynamically-created message channels [29]. Finally, the differences in the models have a direct effect on their implementations: an actor system requires a communication network and a dynamic actor instance manager, while Chain performs most of its work at compile time.

8. Conclusion and Future Work

This work developed Chain, the first programming model to provide intermittence-safety without the need for costly checkpoints. Chain provides a task-granular progress guarantee and its channel memory model keeps data consistent. Channels dispatch with the need to save and restore checkpoints on reboots. Chain ensures consistency and progress with 2-7x higher throughput than prior systems. Such a throughput increase *enables* compute-intensive applications that demand correctness, like 1024-bit RSA and LZW compression, on energy-harvesting devices. Chain opens a promising new research direction around analyzing, optimizing, and refining task-based intermittent programs, as well as in developing new, more sophisticated applications for intermittent systems. An essential question for our future work is to develop system support to size tasks to ensure that their energy demand does not exceed the energy buffer on the device.

Acknowledgments

We thank our anonymous reviewers for their insightful and supportive comments. We thank Ben Ransford and Alanson Sample for their valuable feedback and discussions. This work was generously supported by NSF Grant CNS-1526342 and a gift from Disney Research.

References

[1] G. A. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, Mass, 1986.

[2] D. Balsamo, A. Weddell, G. Merrett, B. Al-Hashimi, D. Brunelli, and L. Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *Embedded Systems Letters, IEEE*, PP(99):1–1, 2014. ISSN 1943-0663. doi: 10.1109/LES.2014.2371494.

[3] T. W. Barr and S. Rixner. Medusa: Managing concurrency and communication in embedded systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 439–450, Berkeley, CA, USA, 2014. USENIX Association.

[4] R. L. Bocchino, E. Gamble, K. P. Gostelow, and R. R. Some. Spot: A programming language for verified flight software. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 97–102, New York, NY, USA, 2014. ACM.

[5] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: An energy-aware task scheduler for computational RFID. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[6] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 1–13, New York, NY, USA, 2006. ACM.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, Mar. 2011. doi: http://dx.doi.org/10.1145/1950365.1950380.

[8] A. Colin, A. P. Sample, and B. Lucia. Energy-interference-free system and toolchain support for energy-harvesting devices. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '15, pages 35–36, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4673-8320-2. URL `http://dl.acm.org/citation.cfm?id=2830689.2830695`.

[9] A. Colin, G. Harvey, B. Lucia, and A. Sample. An energy-interference-free hardware/software debugger for intermittent energy-harvesting systems. In *ASPLOS*, 2016.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Symposium on Operating Systems Principles (SOSP)*, Oct. 2009. doi: http://dx.doi.org/10.1145/1629575.1629589.

[11] M. de Kruijf and K. Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the International Symposium on Code Generation and Optimization "'(CGO)"'*, 2013.

[12] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of 33rd International Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[13] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys '14*, Apr. 2014. doi: http://doi.acm.org/10.1145/2592798.2592814.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, June 2004.

[16] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 388–402, New York, NY, USA, 2003. ACM.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[18] H. Jayakumar, A. Raha, and V. Raghunathan. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*, Jan. 2014.

[19] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 671–690, New York, NY, USA, 2010. ACM.

[20] Y. Kwon, K. Mechitov, and G. Agha. *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, chapter Design and Implementation of a Mobile Actor Platform for Wireless Sensor Networks, pages 276–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[21] Y. Lee, G. Kim, S. Bang, Y. Kim, I. Lee, P. Dutta, D. Sylvester, and D. Blaauw. A modular 1mm3 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*, pages 402–404, Feb 2012. doi: 10.1109/ISSCC.2012.6177065.

[22] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 39–50, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486015. URL `http://doi.acm.org/10.1145/2486001.2486015`.

[23] B. Lucia and B. Ransford. 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, pages 575–585, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737978. URL `http://doi.acm.org/10.1145/2737924.2737978`.

[24] N. A. Lynch and M. Fischer. On describing the behavior and implementation of distributed systems, 1981.

[25] A. Mirhoseini, E. M. Songhori, and F. Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *IEEE Pervasive Computing and Communication Conference (PerCom)*, Mar. 2013. URL `http://aceslab.org/sites/default/files/Idetic.pdf`.

[26] D. Narayanan and O. Hodson. Whole-system persistence with non-volatile memories. In *ASPLOS*, Mar. 2012.

[27] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA*, June 2014.

[28] Powercast Co. Development Kits - Wireless Power Solutions. `http://www.powercastco.com/products/development-kits/`. Visited July 30, 2014.

[29] A. Prokopec and M. Odersky. Isolates, channels, and event streams for composable distributed programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 171–182, New York, NY, USA, 2015. ACM.

[30] Proteus Digital Health. Proteus Digital Health. `http://www.proteus.com/`, 2015.

[31] B. Ransford and B. Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 5:1–5:3, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2917-0. doi: 10.1145/2618128.2618136. URL `http://doi.acm.org/10.1145/2618128.2618136`.

[32] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ASPLOS*, Mar. 2011.

[33] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978. ISSN 0001-0782. doi: 10.1145/359340.359342. URL `http://doi.acm.org/10.1145/359340.359342`.

[34] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an RFID-based battery-free pro-

grammable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, Nov. 2008.

[35] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, Sept. 1995.

[36] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[37] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, Feb. 2011. URL `https://www.usenix.org/legacy/events/fast11/tech/full_papers/Venkataraman.pdf`.

[38] N. Villar and S. Hodges. The Peppermill: A human-powered user interface device. In *Conference on Tangible, Embedded, and Embodied Interaction (TEI)*, Jan. 2010. doi: http://dx.doi.org/10.1145/1709886.1709893.

[39] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984. ISSN 0018-9162. doi: 10.1109/MC.1984.1659158. URL `http://dx.doi.org/10.1109/MC.1984.1659158`.

[40] Zac Manchester. KickSat. `http://zacinaction.github.io/kicksat/`, 2015.

[41] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam. Conair: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems "'(ASPLOS)"'*, 2013.

[42] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.