SAND2017-0038C

Intermittent Computation without Hardware Support or Programmer Intervention

Joel Van Der Woude
*Sandia National Laboratories**

Matthew Hicks
University of Michigan

Abstract

As computation scales downward in area, the limitations imposed by the batteries required to power that computation become more pronounced. Thus, many future devices will forgo batteries and harvest energy from their environment. Harvested energy, with its frequent power cycles, is at odds with current models of long-running computation.

To enable the correct execution of long-running applications on harvested energy—without requiring special-purpose hardware or programmer intervention—we propose Ratchet. Ratchet is a compiler that adds lightweight checkpoints to unmodified programs that allow existing programs to execute across power cycles correctly. Ratchet leverages the idea of idempotency, decomposing programs into a continuous stream of re-executable sections connected by lightweight checkpoints, stored in non-volatile memory. We implement Ratchet on top of LLVM, targeted at embedded systems with high-performance non-volatile main memory. Using eight embedded systems benchmarks, we show that Ratchet correctly stretches program execution across frequent, random power cycles. Experimental results show that Ratchet enables a range of existing programs to run on intermittent power, with total run-time overhead averaging below 60%—comparable to approaches that require hardware support or programmer intervention.

1 Introduction

Improvements in the design and development of computing hardware have driven hardware size and cost to rapidly shrink as performance improves. While early computers took up entire rooms, emerging computers are millimeter-scale devices with the hopes of widespread deployment for sensor network applications [23]. These rapid changes drive us closer to the realization of smart dust [20], enabling applications where the cost and size of computation had previously been prohibitive. We are rapidly approaching a world where computers are not just your laptop or smart phone, but are integral parts your clothing [47], home [9], or even groceries [4].

Unfortunately, while the smaller size and lower cost of microcontrollers enables new applications, their ubiquitous adoption is limited by the form factor and expense of batteries. Batteries take up an increasing amount of space and weight in an embedded system and require special thought to placement in order to facilitate replacing batteries when they die [20]. In addition, while Moore's Law drives the development of more powerful computers, batteries have not kept pace with the scaling of computation [18]. Embedded systems designers attempt to address this growing gap by leveraging increasingly power-efficient processors and design practices [49]. Unfortunately, these advances have hit a wall—the battery wall; enabling a dramatic change in computing necessitates moving to batteryless devices.

Batteryless devices, instead of getting energy from the power grid or a battery, harvest their energy from their environment (*e.g.,* sunlight or radio waves). In fact, the first wave of energy harvesting devices are available today [4, 50, 9]. These first generation devices prove that it is possible to compute on harvested energy. This affords system designers the novel opportunity to remove a major cost and limitation to system scaling.

Unfortunately, while harvested energy represents an opportunity for system designers, it represents a challenge for software developers. The challenge comes from the nature of harvested energy: energy harvesting provides insufficient power to perform long-running, continuous, computation [37]. This results in frequent power losses, forcing a program to restart from the be-

*Work completed while at the University of Michigan

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

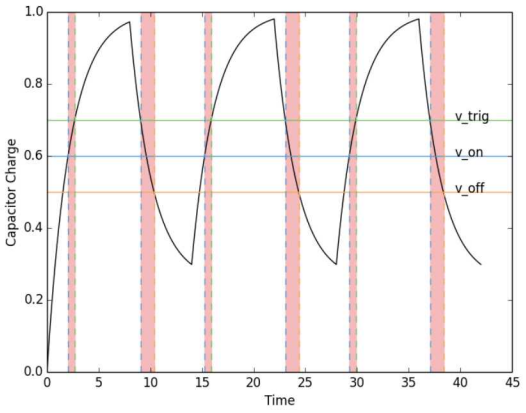

Figure 1: Energy harvesting devices replace batteries with a small energy storage capacitor. This is the ideal charge and decay of that capacitor when given a square wave input power source. The voltage across the capacitor must be above `v_trig` for program computation to take place as that is the minimum energy required to store the *largest* possible checkpoint in the *worst-case* environmental and device conditions. The energy expended going from `v_on` to `v_trig` and from `v_trig` to `v_off` is wasted in what is called the guard band. The guard band is essential for correctness in one-time checkpointing systems [39, 17, 3]. In real systems, the charge and discharge rate is chaotic, depending on many variables, including temperature and device orientation.

ginning, in hopes of more abundant power next time. While leveraging faster non-volatile memory technologies might seem like an easy way to avoid the problems associated with these frequent power cycles, previous work exposes the inconsistent states that can result from power cycles when using these technologies [26, 38].

Previous research attempts to address the problem of intermittent computation using two broad approaches: rely on specialized hardware [39, 29, 3, 17, 27] or require the programmer to reason about the effects of common case power failures on mixed-volatility systems [26]. Hardware-based solutions, rely on a single checkpoint that gets saved just before power runs out. It is critical for correctness that every bit of work gets saved by the checkpoint and that there is no (non-checkpointed) work after a checkpoint [38]. On the other hand, taking a checkpoint too early wastes energy after the checkpoint that could be used to perform meaningful work. This tradeoff mandates specialized hardware to measure available power and predict when power is likely to fail. Due to the intermittent nature of harvested energy, predicting power failure is a risky venture that requires large guard bands to accommodate a range of environmental conditions and hardware variances. Figure 1 depicts how guard-bands waste energy that could otherwise be used to make forward progress. In addition to the guard-bands wasting energy, the power monitoring hardware itself consumes power. Even simple power monitoring circuits (think 1-bit voltage level detector) consume power up to 33% of the power of modern ultra-low-power microcontrollers [5, 43].

An alternative approach, as taken by DINO [26], is to forgo specialized hardware, instead, placing the burden on the programmer to reason about the possible outcomes of frequent, random, power failures. DINO requires that programmers divide programs into a series of checkpoint-connected tasks. These tasks then use data versioning to ensure that power cycles do not violate memory consistency. Smaller tasks increase the likelihood of eventual completion, at the cost of increased overhead. Larger tasks result in fewer checkpoints, but risk never completing execution. Thus, the burden is on the programmer to implement—for all control flows—correct-sized tasks given the program and the expected operating environment. Note that even small changes in the program or operating environment can change dramatically the optimal task structure for a program.

Our goal is to answer the question: *What can be done without requiring hardware modifications or burdening the programmer?* To answer this question, we propose leveraging information available to the compiler to preserve memory consistency without input from the programmer or specialized hardware. We draw upon the wealth of research in fault tolerance [31, 24, 21, 51, 25, 13] and static analysis [7, 22] and construct Ratchet, a compiler that is able to decompose unmodified programs into a series of re-executable sections, as shown in Figure 2. Using static analysis, the compiler can separate code into idempotent sections—*i.e.,* sequences of code that can be re-executed without entering a state inconsistent with program semantics. The compiler identifies idempotent sections by looking for loads and stores to non-volatile memory and then enforcing that no section contains a write after read (WAR) to the same address. By decomposing programs down to a series of re-executable sections and gluing them together with checkpoints of volatile state, Ratchet supports existing, arbitrary-length programs, no matter the power source. Ratchet shifts the burden of reasoning about the effects of intermittent computation and mixed-volatility away from the programmer to the compiler—without relying on hardware support.

We implement Ratchet as a set of modifications to the LLVM compiler [22], targeting energy harvesting platforms that use the ARM architecture [2] and have wholly non-volatile main memory. We also implement an ARM-based energy harvesting simulator that simulates power

failures at frequencies experienced by existing energy harvesting devices. To benchmark Ratchet, we use it to instrument the newlib C-library [45], libgcc, and eight embedded system benchmarks [14]. Finally, we verify the correctness of Ratchet by executing all instrumented benchmarks, over a range of power cycle rates, on our simulator which includes a set of memory consistency invariants dynamically check for idempotence violations. Our experimental results show that Ratchet correctly stretches program execution across frequent, random power cycles, while adding 60% total run-time overhead[1]. This is comparable to approaches that require hardware support or programmer intervention and much better than the alternative of most benchmarks never completing execution with harvested energy.

This paper makes several key contributions:

- We design and implement the first software-only system that *automatically* and *correctly* stretches program execution across random, frequent power cycles. As a part of our design, we extend the notion of idempotency to memory.
- We evaluate Ratchet on a wider range of benchmarks than previously explored and show that its total run-time overhead is competitive with existing solutions that require hardware support or programmer intervention.
- We open source Ratchet, including our energy harvesting simulator [16], our benchmarks [15], and modifications to LLVM to implement Ratchet [44].

2 Background

The emergence of energy harvesting devices poses the question: *How can we perform long-running computation with unreliable power?* Answering this question forces us to go beyond a direct application of existing work from the fault tolerance community due to four properties of energy harvesting devices:

- Power availability and duration are unknowable for most use cases.
- Added energy drain by hardware is just as important as added cycles by software.
- Small variations in the device and the environment have large effects on up time.
- Faults (*i.e.,* power cycles) are the common case, not a special case.

[1]We say total run-time overhead to cover all sources of run-time overhead, including hardware and software. Keep in mind that adding hardware indirectly increases the run time of programs by decreasing the amount of energy available for executing instructions. Because Ratchet is software only, the total run-time overhead is equal to the overhead due to saving checkpoints and re-execution.

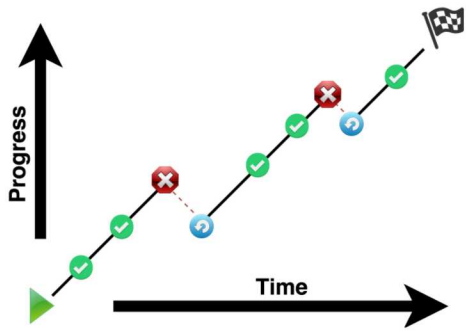

Figure 2: Ratchet-compiled program in operation. The checkmarks represent completed checkpoints, the x's represent power failures, and the dashed lines to the backward rotating arrows represent the system restarting execution at the latest checkpoint. This figure shows how programs execute as normal with added overhead from checkpoints and re-execution.

These four properties dictate how system builders construct energy harvesting devices and how researchers make programs amenable to intermittent computation.

2.1 Prediction vs. Resilience

Given the properties of harvested energy, there are two checkpointing methods for enabling long-running computation: one-time and continuous. One-time checkpointing approaches attempt to predict when energy is about to run out and checkpoint all volatile state right before it does. Doing this requires measuring the voltage across the energy storage capacitor as depicted in Figure 1. Measuring the voltage requires an Analog-to-Digital Converter (ADC) configured to measure the capacitor's voltage. Hibernus [3], the lowest overhead one-time checkpointing approach, utilizes an advanced ADC with interrupt functionality and a configurable voltage threshold that removes the need to periodically check the voltage from software. As Table 1 shows, this produces very low overheads, with the two main sources of overhead being the extra power consumed by the ADC and the energy wasted waiting in the guard bands.

While many energy harvesting systems have ADCs, the program may require use of the ADC, the ADC may not support interrupts, or the ADC may not be configured (in hardware) to monitor the voltage of the energy storage capacitor. Without such an ADC, programs must be able to fail at any time and still complete execution correctly. Making programs resilient to spontaneous power failures is the domain of continuous checkpointing systems. Continuous checkpointing systems must maintain the abstraction of *re-execution memory consistency* (*i.e.,* a section of code is unable to determine if it is being

Property	Commodity Checkpointing [32, 11, 31, 34]	Energy Harvesting HW-assisted [17, 3, 29]	DINO [26]	Idempotence [7, 13]	Ratchet
Failure Rate	Days/Weeks	100 ms	100 ms	Days/Weeks	100 ms
Requires	Varies	HW+Compiler	Programmer+Compiler	Compiler	Compiler
Failure Type	Transient Fault	Power Loss	Power Loss	Transient Fault	Power Loss
Memory type	DRAM+HDD	FRAM	SRAM+FRAM	DRAM+HDD	FRAM
Chkpt. Trigger	Time	Low Voltage	Task Boundary	Register WAR	NV WAR
Chkpt. Contents	Varies	VS	VS+NV_TV	—	VS
Overhead	Varies	0–145% [3]	80–170%	0–30%	0–115%
Primary Factor	Varies	Measurement	Task Size	# Faults	Section Size

Table 1: Requirements and behavior of different checkpointing/logging techniques. `WAR` represents a Write-After-Read dependence, `VS` represents Volatile State (*e.g.,* SRAM and registers), `NV` represents Non-Volatile memory (*e.g.,* FRAM), `NV_TV` represents Task Variables stored in Non-Volatile memory, and `Measurement` represents the added time and energy consumed by using voltage monitoring hardware.

executed for the first time or being re-executed by examining memory)[2]. To maintain re-execution memory consistency, continuous checkpointing systems periodically checkpoint volatile state and guard against inconsistent updates to non-volatile memory. DINO [26] does this through data versioning, while Ratchet does this through maintaining idempotence. Table 1 shows that this class of approach also yields low total overheads, with the primary source being time spent checkpointing.

2.2 Memory Volatility

Another consideration for energy harvesting devices is the type, placement, and use of non-volatile memory. While the initial exploration into support for energy harvesting devices, Mementos [39], focuses on supporting Flash-based systems with mixed-volatility main memory, all known follow-on work focuses on emerging systems with Ferroelectric RAM (FRAM)-based, non-volatile, main memory [17, 3, 26, 27]. This transition is necessary as several properties of Flash make it antithetical to harvested energy. The primary reason Flash is ill-suited is its energy requirements. Flash works by pushing charge across a dielectric. Doing so is an energy intense operation requiring high voltage that makes little sense when the system is about to run out of power. In fact, on MSP430 devices, Flash writes fail at a much higher voltage than the processor itself fails [5]—increasing the energy wasted in the guard band. A second limitation of Flash is that most programs avoid placing variables there, increasing the amount of volatile state that requires checkpointing. Flash writes, beyond being energy expensive, are slow and complex. Updating a variable stored in Flash requires erasing a much larger block of memory and rewriting all data, along with the one updated value. This process adds complexity to applications and increases write latency over FRAM by two-orders of magnitude.

In comparison with Flash memory, FRAM boasts extremely low voltage writes, as low as a single volt [35]. Writes to FRAM are also nearly as fast as writes to SRAM and are bit-wise programmable. The flexibility and low overheads of FRAM allows for processor designers to create wholly non-volatile main memory, decreasing the size and cost of checkpoints. This opens the door to continuous checkpointing systems as the cost of checkpointing is outweighed by the power requirement of the ADC. While, like previous approaches, we focus on FRAM due to its commercial availability, there are competing non-volatile memory technologies (*e.g.,* Magnetoresistive RAM [46] and Phase Change RAM [40]) that we expect to work equally well with Ratchet. Moving program data to non-volatile memory does come with a cost: previous work reveals that mixing non-volatile and volatile memory is prone to error [38, 26]. Ratchet deals with this by pushing such complexity into the compiler, relieving the programmer from the burden of reasoning about mixed volatility main memory and the effects of power cycles.

3 Design

Ratchet seeks to extend computation across common case power cycles in order to enable programs on energy harvesting devices to complete long-running computation with unreliable power sources. Ratchet enables re-execution after a power failure by saving volatile state to non-volatile memory during compiler inserted

[2]Note that this is a relaxation on the requirement for deterministic re-execution [48, 31, 30, 33], where its is required that each re-execution produce the same exact result. Our problem only requires that re-executions produce a semantically correct execution.

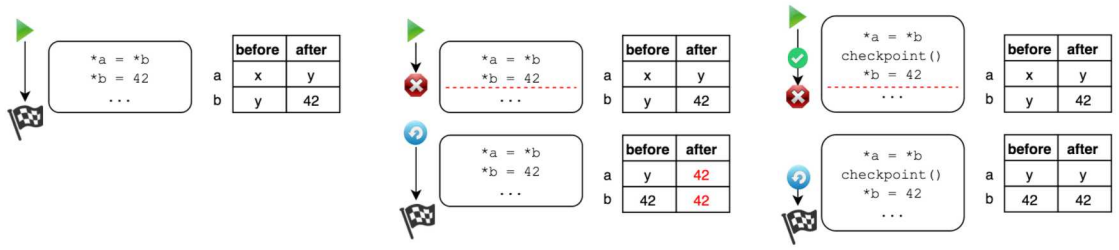

Figure 3: The same code executed without failures, with failures, and with failures with Ratchet. This basic example illustrates one of the difficulties with using non-volatile main memory on intermittently powered computers: failures can create states not possible given program semantics.

checkpoints. However, checkpointing alone is insufficient to ensure correct computation due to the problems with maintaining consistency between volatile and non-volatile memory give unpredictable failures [38, 26]. To ensure correct re-execution, we use compiler analysis to determine sections of code that may be re-executed from the beginning, without producing different results; a property called idempotence. After decomposing programs into idempotent sections, we connect the independent sections with checkpoints of volatile state. This ensures that after a power failure the program will resume with a view of all memory identical to the first (attempted) execution[3].

3.1 Idempotent Sections

Idempotent sections are useful because they are naturally side-effect free. Nothing needs to be changed about them in order to protect against memory consistency errors that may arise from partial execution. By recognizing code sections with this property, Ratchet is able to find potentially long sequences of instructions that can be re-executed without any additional work required to ensure memory consistency.

De Kruijf *et al.* identify idempotent sections by looking for instruction sequences that perform a Write-After-Read (WAR) to the same memory address [7]. Under normal execution, overwriting a previously read memory location is inconsequential. However, on systems that roll back to a previous state for recovery (due to potential issues such as power failures), overwriting a value that was previously read will cause a different value to be read when the code section is re-executed. Figure 3 shows an example of how re-executing a section of code with a WAR dependency may introduce execution that diverges from program semantics.

[3]We are not saying the memory must be identical. We are saying that the values that a given idempotent section of code reads are identical to the initial execution. Idempotency enables this relaxation.

In order to prevent these potential consistency problems Ratchet inserts a checkpoint between the write and the read. This breaks the dependency, separating the read and the write in different idempotent sections. This ensures that the read always gets the original value and the write will be contained in a different idempotent section, where it is free to update the value. Note that a sequence of instructions that contains a WAR may still be idempotent if there exists a write to the same memory address before the first read. For example a WARAW dependency chain is idempotent since the first write initializes the value stored at the memory address so that even if it is changed by the last write, it will be restored upon re-execution before the address is read again. Note that this holds for a potentially infinite sequence of writes and reads, the sequence will be idempotent if there is a write before the first read.

It is important to remember that in order for a load followed by a store cause consistency problems, they must read and modify the same memory. In order to determine which instructions rely on the same memory locations, Ratchet uses intraprocedural alias analysis due to its availability, performance, and precision. Alias analysis *conservatively* identifies instructions that *may* read or modify the same memory locations. Since the alias analysis is intraprocedural we conservatively assume all stores to addresses outside of the stack frame may alias with loads that occurred in the caller. This forces Ratchet to insert a checkpoint along any control flow path that includes a store to non-local memory.

After finding all WARs we use a modified hitting set algorithm to insert the minimum number of checkpoints between the loads and stores. The algorithm works by assigning weights to different points along the control flow graph based upon metrics such as loop depth and the number of other idempotency violations intersected. It uses these metrics to identify checkpointing locations that prevent all possible idempotency violations while

trying to avoid locations that will be re-executed more often than necessary. For example, do not checkpoint within a loop if a checkpoint outside the loop will separate all WARs. For a more in-depth discussion of this algorithm, we refer you to de Kruijf *et al.* [7].

3.2 Implicit Idempotency Violations

While looking for WARs identifies the majority of code sequences that violate idempotence, some instructions may implicitly violate idempotence. A pop instruction can be modeled by a read and subsequent update to the stack pointer. This update immediately invalidates the memory locations just read by allowing future push instructions to overwrite the old values. On a system with interrupts, this scenario occurs when an interrupt fires after a pop instruction. In this case, the pop instruction will read from the stack and update the stack pointer. When the interrupt occurs it will perform a push instruction to callee save registers, in order to preserve the state of the processor before the interrupt fired. However, the state saved by the interrupt is written to the stack addresses that were read by the initial pop. If the system re-executes the pop instruction, it will read a value from the interrupt handler—not the original value! This behavior forces Ratchet to treat all pop instructions as implicit idempotency violations since interrupts are not predictable at compile time.

In order to enable a checkpoint before the slots on the stack are freed Ratchet exchanges pop instructions for a series of instructions that perform the same function. The first duty of the pop instruction is to retrieve a set of values from the stack and insert them into registers. Ratchet emulates this by inserting a series of load instructions to retrieve these values from the stack and place them in their respective registers. Note that the load instructions do not update the stack pointer so any interrupt that fires will push the new values above these values. After the data is retrieved from the stack, Ratchet inserts a checkpoint. Finally, Ratchet inserts an update to the stack pointer to free the space previously occupied by the values that we just loaded into registers. By emitting this sequence of instructions we have deconstructed an atomic read write instruction that is an implicit idempotency violation and replaced it by a series of instructions that enable separation of potential idempotency violations.

3.3 Checkpoints

In between each naturally occurring idempotent section, we insert checkpoints in order to save all volatile memory necessary to restart from the failure. In emerging systems we observe non-volatile memory moving closer and closer to the CPU, so far that it has non-volatile RAM [42]. In such a system, all that is needed to restore state are the values stored in the registers. In fact, not all registers are necessary, only registers that are used as inputs to instructions that occur after the checkpoint location. These registers are denoted live-in registers.

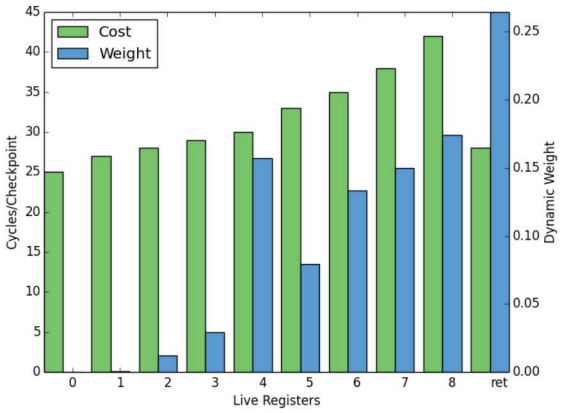

Figure 4: Shows the relationship between checkpoint overhead and live registers. A checkpoint's cost is the number of cycles it takes to commit and its weight refers to how often they occur in our benchmarks relative to the total number of checkpoints.

Traditionally, compilers keep a list of live-in and live-out registers to determine which registers are needed in a basic block to perform some computation and which ones are unused and can be reallocated to reduce register spilling. This information is available to the compiler after registers have been allocated. We are interested in which registers are live-in to a checkpoint because they denote the volatile memory needed to correctly restart from a given location in a program. Figure 4 shows the relationship between checkpoint overhead and number of live-in registers.

In order to prevent power failures during a checkpoint from causing an inconsistent state, we use a double buffering scheme. One buffer holds the previous checkpoint, while the other is used for writing a new checkpoint. A checkpoint is committed by updating the pointer to the valid checkpoint buffer as the last part of writing the checkpoint. We tolerate failures even while taking a checkpoint by never overwriting a checkpoint until a more recent checkpoint is available in the other buffer. The atomicity of the store instruction for a singe word ensures that we always have a valid checkpoint regardless of when we experience a power failure.

3.4 Recovery

In order to recover from a power failure, we insert code before the main function to check to see if there ex-

ists a valid checkpoint. If so, we determine that we have experienced a power failure and need to restore state, otherwise we begin executing from main. Restoring state consists of moving all saved registers into the appropriate physical registers. Once the program counter has been restored, execution will restart from the instruction after the most recently executed checkpoint.

In the case that some idempotent sections are too long, that is, power may repeatedly fail before a checkpoint is reached, we use a timer that triggers an interrupt in order to ensure forward progress. Each interrupt checks to see if a new checkpoint has been taken since the last time it was called. It does this by zeroing-out the program counter value in the unset checkpoint buffer each time it is called. It can then tell if a checkpoint has been taken since its last call and only checkpoint if the program counter of the unused checkpoint buffer is still zero.

When checkpointing from the timer interrupt it is impossible to foresee which registers are still live and we must instead conservatively save all of the registers to non-volatile memory. There exists a trade off when selecting the timer speed. A timer that is short increases the overhead due to checkpointing, while a timer that is long increases re-execution overhead. Without additional hardware to measure environmental conditions, the timer can be set on the order of experts estimation of average lifetime for transiently powered devices [39]. Note that for our benchmarks, a timer was not needed in order to make forward progress.

3.5 Challenges

During implementation of Ratchet we encountered a number of design challenges with actually implementing our ideal design. Most of these challenges related to being entrenched at different levels of abstraction within the compiler. While the code is in the compiler's intermediate representation (IR), powerful alias analysis and freedom from architecture level specifics made for a logical location to identify WAR dependencies and insert checkpoints. However, these decisions are dependent on the choices made during the translation from the compiler's IR to machine code, such as which calls are tail calls and information about when register pressure causes register spilling.

This semantic gap causes conservative decision making about where to place checkpoints, since the decisions made in the front end rely on assumptions about where checkpoints will be inserted in the back-end.

3.6 Optimizations

As we began to implement our design, it became clear that we were inserting checkpoints more frequently than needed. As a result of profiling our initial design we implemented several optimizations to remove redundant checkpoints.

```
r1 = mem[sp + 4]
checkpoint()
mem[sp + 4] = r2
...
r3 = mem[r2 + 8]
checkpoint()
mem[r2 + 8] = r4
       a.
```

```
r1 = mem[sp + 4]
r3 = mem[sp + 8]
checkpoint()
mem[sp + 4] = r2
...
checkpoint()
mem[sp + 8] = r4
       b.
```

Figure 5: Two possible code sequences. In (a) each WAR dependency is separated by a checkpoint, but the two checkpoints cannot be combined without violating idempotency. In (b), the second checkpoint could be moved to the same line as the first checkpoint since there are no potentially aliasing reads separating the two checkpoints.

3.6.1 Interprocedural Idempotency Violations

Because of the limits on interprocedural alias analysis, we initially conservatively inserted a checkpoint on function entry. This protects against potential WAR violations between caller and callee code. We observed that this was often conservative and could be relaxed. A checkpoint is only necessary on function entry if there exists a write that may alias with an address outside of the function's local stack frame. In the presence of an offending write, the function entry is modeled as the potentially offending read (since an offending load could have occurred in the caller).

In addition, we noticed that some tail calls could be implemented without any checkpoints. Since tail calls operate on the stack frame of their caller, a checkpoint on return is unnecessary, assuming that the tail call does not modify non-local memory. However, opportunities for this optimization were observed to be limited due to the difficulty of determining where to put checkpoints for intraprocedural WAR dependencies. This is a result of the semantic gap between compiler stages, when identifying WAR dependencies, we do not yet have perfect information about which calls can be represented as tail calls. We imagine a more extensive version of Ratchet that takes information from each stage of the compiler pipeline and iteratively adjusts checkpoint locations to find the near-minimal set of checkpoints that maintain correctness.

3.6.2 Redundant Checkpoints

Due to the semantic gap between our alias analysis and insertion of checkpoints in the front-end of the compiler (while the code is still in IR), and the instruction scheduling of the back-end, we observed cases where optimiza-

tions or other scheduling decisions caused redundant checkpoints. We consider redundant checkpoints to be a pair of checkpoints where any potential idempotency violations could be protected with a single checkpoint. In general, a checkpoint can be relocated within its basic block to any dominating location does not cross any reads and any dominated location that does not cross any writes. This conservative rule follows even without knowing alias information, which allows us to reorder instructions after machine code has been generated and we no longer know which instructions generated the WAR dependency. Figure 5 shows an example of how checkpoints can be combined safely by relocation.

3.6.3 Special Purpose Registers

Since all volatile state must be saved during a checkpoint, all live special purpose registers must be saved along with the live general purpose registers. Some of the special purpose registers have higher costs to save than the general purpose registers. In our experience implementing Ratchet, we found the cost of checkpointing condition codes to be high. Instead of paying this overhead, we ensured checkpoints were placed after the condition code information was consumed, while still ensuring all non-volatile memory idempotency violations were cut. Ratchet does this by reordering instructions to ensure a checkpoint is not placed between a condition code generator and all possible consumers. This instruction reordering is done with the same constraints as combining checkpoints.

3.7 Architecture Specific Tradeoffs

There are a number of architectural decisions that influence the overhead of our design. Register-rich architectures reduce the number of idempotent section breaks required by reducing the frequency of register spills, at the cost of increasing checkpoint size. Atomic read-modify-write instructions are incompatible with our design since there is no way to checkpoint between the read and the write. On such a architecture, Ratchet could separate the instruction into separate load and store operations by our compiler implementation.

4 Implementation

We implement Ratchet using the LLVM compiler infrastructure [22]. Beyond verifying that Ratchet output executes correctly on an ARM development board [41], we build an ARMv6-M [2] energy harvesting simulator, with wholly non-volatile memory. The simulator allows for fine-grain control over the frequency, arrival time, and effects of power cycles, as well as allowing us to verify Ratchet's correctness. The benchmarks we use for evaluation all depend on libc, so we also use Ratchet to instrument newlib [45], and link our benchmarks against our instrumented library.

4.1 Compiler

We build our compiler implementation on top of the LLVM infrastructure [22]. We add a front end, IR level pass that detects idempotent section breaks by tracking loads and stores to non-volatile memory that may alias (based on earlier work targeted at registers [7]). This top level pass inserts checkpoint placeholders that are eventually passed to the back end where machine code is eventually emitted. The back end replaces pop instructions with non-destructive reads and a checkpoint followed by an update to the stack pointer. Next, the back end relocates the inserted placeholders to minimize the number of checkpoints required by combining them and avoiding bisecting condition code def-use chains. After register allocation, each placeholder is replaced by a function call to the checkpointing routine that saves the fewest possible registers. We determine the minimal set of registers to save using the liveness information available in the back end.

One compiler-inserted idempotency violating construct that we modify the compiler to prevent is the use of shared stack slots for virtual registers. If the compiler were able to re-assign the same stack slot to a different virtual register, it would create an idempotency violation as the original virtual register value is overwritten by a different virtual register's value. While it is possible to include some backend analysis to uncover such situations, we choose to sacrifice some stack space and prevent the sharing of stack slots. Achieving this in LLVM is as simple as adding the flag `-no-stack-slot-sharing` to the compile command. In addition, we include the `mem2reg` optimization that causes some variables that would otherwise be stored in memory to be stored in registers. This reduces the number of idempotency violations thereby reducing the number of checkpoints and increasing idempotent section length.

4.2 Energy Harvesting Simulator

We also implement a cycle accurate ARMv6-M simulator. Many of the coming internet of things class devices are choosing to use ARM devices as they are the performance per Watt leader. As the price of new non-volatile memory technologies decreases and their speed increases, we expect ARM to follow in the footsteps of the MSP430 [42] and move to a wholly non-volatile main memory. In fact, even Texas Instruments, the maker of the MSP430, recently moved to ARM for their newest MSP devices [43].

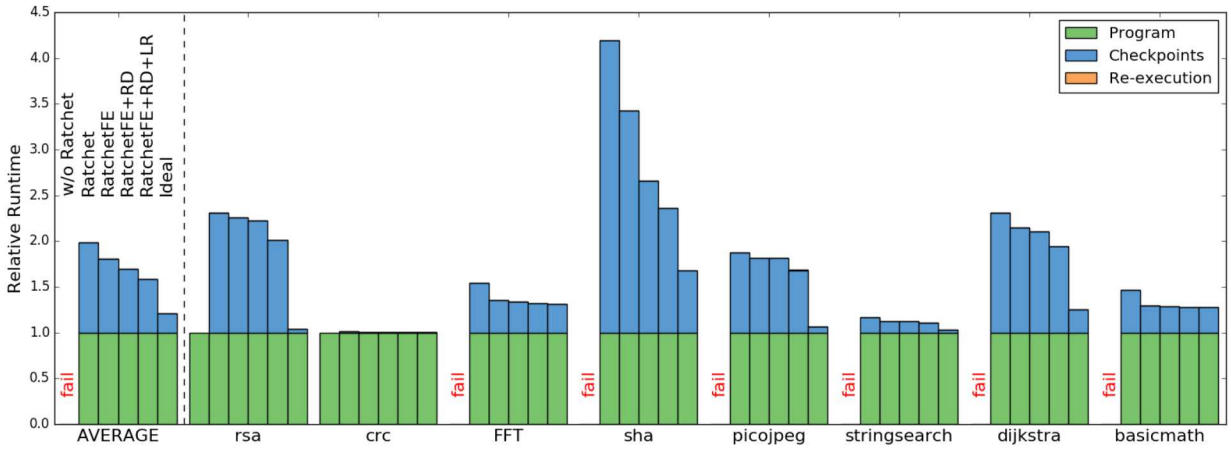

Figure 6: Runtime overhead for several versions of Ratchet.

An energy harvesting simulator is required because of the difficulties associated with developing and debugging intermittently powered devices. Using a cycle accurate simulator we are able to simulate failures with a probability distribution that can model the true frequency and effects of power failures experienced by devices in the real world. Using a simulator also allows us to take precise measurements of how much progress a program makes per power cycle and the cycles consumed by re-execution. Lastly, our simulator allows for us to verify the correctness of Ratchet for every benchmark run.

4.3 Idempotent Libraries

In order to ensure that each section of code that runs is idempotent, we instrument all of the libraries needed by the device. In order to facilitate real applications we compile newlib [45], a basic libc and libm library aimed at embedded systems, with Ratchet. This requires a modifying three lines in newlib's makefile to prevent it from building these optimized versions of libc calls. We did this because any uninstrumented code could cause memory consistency to be violated (due to idempotency violations) if power fails after a write-after-read dependency and before a checkpoint.

Lastly, we produce an instrumented version of the minimum runtime functions expected by clang that are included in the compiler-rt project [1]. These functions implement the libgcc interfaces expected by most compilers. As with newlib, we use only the bare minimum optimized assembly implementations. Those that we use, we insert checkpoints by hand between potential write-after-read dependencies. Thankfully, this only needs to be done once by the library's author.

Note that Ratchet supports assembly as long as the assembly is free of idempotence violations or all potential idempotency violations are separated by checkpoints. With additional engineering effort, it is possible to create a tool that inserts these checkpoints automatically through static analysis of the assembly [8].

5 Evaluation

In order to provide a comparison against other checkpointing solutions for energy harvesting devices, we evaluate Ratchet on benchmarks common to these approaches, namely, RSA, CRC, and FFT. For a more complete analysis, we port[4] several benchmarks from MiBench, a set of embedded systems benchmarks categorized by run-time behavior [14]. Expanding the benchmark set used to evaluate energy harvesting systems is crucial, because testing with a wide range of program behaviors and truly long-running programs is more likely to expose an approach's tradeoffs.

Unless otherwise noted, we compile all benchmarks with `-O2` as the optimization level. We choose the `-O2` level since it includes most optimizations while avoiding code size versus speed tradeoffs. As Section 5.3 illustrates, Ratchet supports all of LLVM's optimization levels. We use an average lifetime of 100 ms to match the setup of previous works. With an average lifetime of 100 ms running with a 24 MHz clock, this gives us a mean lifetime of 2,400,000 cycles. Before each bout of execution, the simulator samples from a Gaussian whose mean is the desired mean lifetime (in cycles) and uses that value as the number of clock cycles to execute for

[4] Some of these applications require input that is read in from a file. Since many energy harvesting systems do not include an operating system or file system, we instead compile the input into the binary.

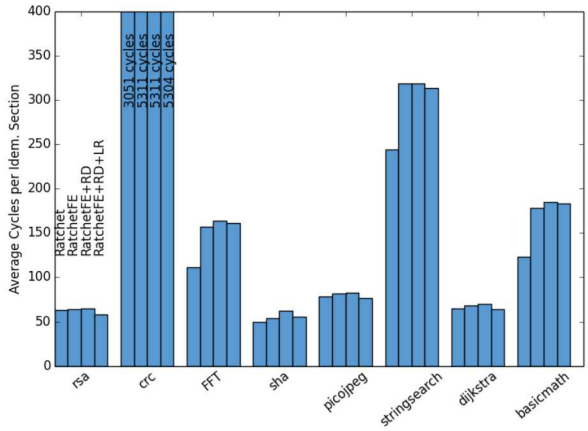

Figure 7: The average number of cycles per idempotent section break.

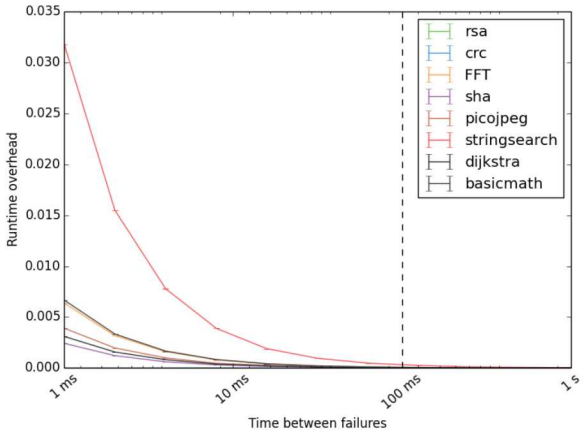

Figure 8: Re-execution overhead decreases as failure frequency increases. Note that CRC and RSA have zero re-execution overhead throughout since they are short enough to complete in a single power cycle.

before inducing the next power cycle. To simulate a power cycle, we clear the register values.

Given this experimental setup, we set out to answer several key questions about Ratchet:

1. Does Ratchet stretch computation across frequent, unpredictable losses of power correctly?
2. What is the overhead of running Ratchet due to checkpoints and re-execution?
3. Is Ratchet compatible with compiler optimizations?
4. What impact does Ratchet have on code size?

We use the results of this evaluation to compare Ratchet against alternative approaches that require hardware support or programmer intervention. See the results of this comparison in Table 1 and Section 7.2.

5.1 Performance

To understand the effects of power cycles on long-running programs and the overhead of Ratchet, we perform 10 trials of each benchmark with power failures as described earlier. Figure 6 displays the results of this experiment for each benchmark, averaged and normalized to the run time of the benchmark executed without power failures. The first thing to note is that 6 out of 8 of the benchmarks fail to complete execution on harvested energy without Ratchet (`w/o Ratchet`). There are several other results for each benchmark that represent successive Ratchet optimizations (all levels except `Ideal` maintain correctness): `Ratchet` shows the performance from our naive implementation; `RatchetFE` denotes placing a checkpoint at function entry only when there is a store to a non-local variable that is not preceded by a checkpoint; `RatchetFE+RD` is `RatchetFE`, but with duplicate checkpoints removed in LLVM's backend; `RatchetFE+RD+LR` adds a further optimization of only checkpointing the live-in registers; and finally, `Ideal` represents a lower bound on Ratchet's overhead that assumes perfect intraprocedural alias analysis and zero idempotence violations. `Ideal` bounds what is possible with more compiler engineering.

We observe an average run-time overhead of 58.9% using `Ratchet+FE+RD+LR`—a 20.1% improvement over `Ratchet`. The `Ideal` result suggests that further compiler engineering can reduce this overhead by over 60%. The total overhead includes run-time overhead due to saving checkpoints and re-execution, but checkpoint overhead dominates total overhead, because re-execution overhead approaches zero.

Looking at Figure 6, we can see that overhead varies dramatically between benchmarks. This shows that performance of our method is highly program dependent. Intuitively, this makes sense. If one program includes an implicit WAR dependence buried deep in the hot sections of code and another has very few WAR dependencies, we would expect their run times to vary dramatically. In order to determine the effect of idempotent section length of a benchmark on performance we measure the number of cycles between each checkpoint commit. To measure this, we instrument our simulator to measure the number of cycles[5] between each call to one of our checkpointing functions. We then run each benchmark to completion, without power cycles. The average number of cycles per idempotent section for each benchmark is shown in Figure 7. By comparing Figures 6 and 7 we notice that programs with shorter idempotent sections have higher

[5]Most instructions take a single clock cycle to complete in the ARMv6-M instruction set.

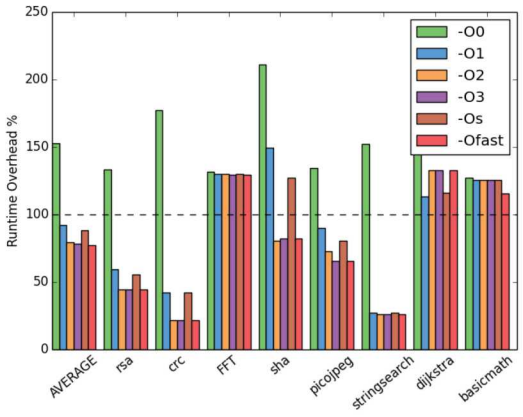

Figure 9: The runtime overhead of each benchmark compiled with Ratchet at various LLVM optimization levels normalized to the runtime of the uninstrumented benchmark compiled at `-O0`.

overheads.

We also investigate the relationship between idempotent section length and re-execution overhead. To do this, we instrument our simulator to measure the number of cycles from the last checkpoint to a checkpoint restore. This includes the cycles spent executing code that occurs after the last checkpoint and the cycles spent restarting and restoring the last checkpoint. Figure 8 shows the fraction of run-time overhead due to re-execution for a range of average power-on times. While short idempotent sections tend to cause higher overall overhead due to checkpoint overhead dominating the total overhead, Figure 8 combined with Figure 7 shows that benchmarks with shorter idempotent sections have lower re-execution costs. This is reasonable considering that a failure halfway through a idempotent section requires the program to re-execute from the last checkpoint. The more cycles since the last checkpoint, the higher the re-execution overhead. This suggests that increases in idempotent section length eventually will expose re-execution time as a key component of overhead.

5.2 Correctness

We validate Ratchet's correctness using both formal and experimental approaches. First, to test that Ratchet enforces idempotency with respect to non-volatile memory, we instrument the simulator to log reads and detect WAR dependencies that occur during execution. We consider a program to fail if there exists any load and subsequent store, to the same address, that are not separated by a checkpoint[6]. Second, to test that Ratchet enables long-running execution even with power-cycle-induced volatile state corruption, we simulate random power failures like those experienced in energy-harvesting devices and verify the results. We check the validity by running different sequences of failures and hashing memory contents and registers at the completion of each benchmark run to compare the hash to the hash of the ground truth run without power failure. Lastly, to ensure Ratchet works even in the most energy starved environments we repeat this experiment with lifetimes as short as 1 ms.

[6]This was especially helpful in debugging, as it exposed missed WAR dependencies, such as ones caused by spilling registers onto the stack due to register pressure, in early prototypes.

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

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

5.3 Impact of Compiler Optimizations

In order to understand the performance of Ratchet under varying compiler optimizations, we benchmark Ratchet across each LLVM optimization level. Figure 9 shows the performance of the benchmarks compiled, with Ratchet, at different optimization levels relative to uninstrumented benchmarks compiled at `-O0`. We observe that in general, traditional compiler optimizations improve the performance of Ratchet. However, it also shows that in some cases, aggressive optimization results in higher perceived overheads. This suggests that breaking the program into idempotent sections can not only reduce the efficiency of optimizations, but cause them to be detrimental (see Dijkstra).

5.4 Code size increase from Ratchet

Code size is a critical constraint for many energy harvesting devices. In order to evaluate Ratchet's practicality with respect to code size, we measure the effect Ratchet has on code size. On average, Ratchet increases the size of the program by 1.79% or 2896 bytes. This increase is caused by adding our checkpoint recovery code, a number of optimized checkpointing functions, checkpoint calls throughout the program, and exchanging pop instructions for loads. Table 2 shows the change in code size for each benchmark. We notice that about

1356 bytes are a result of the additional function calls and reserved areas in memory. The rest of the code size increase can be attributed to inserted code or code that has been rewritten to support Ratchet, namely the translation of pop instructions. Note that the FFT program actually sees a decrease in size. We suspect this might be a result of Ratchet limiting optimization opportunities such as loop unrolling.

6 Discussion

There are several issues that represent corner-cases to Ratchet's design, such as avoiding repeating outputs on re-execution, instrumenting hand-written assembly, and dealing with the effects of power cycles too short to make forward progress. Instead of distracting from the core design, we discuss them here.

6.1 Output Commit Problem

In any replay-based system, there is a dilemma called the output commit problem, which states that a process should not send data to the outside world until it knows that that data is error free. The output commit problem takes on new meaning for energy harvesting devices; the problem is one of sending multiple outputs when only one should have been sent in a correct execution. This problem occurs during the re-execution of a code section that created an output during an earlier execution. Imagine an LCD interface on an embedded system, where there is re-execution while printing to the screen. We suggest placing checkpoints immediately before and after these output instructions to minimize the chance of re-execution due to the instruction being at the end of a long idempotent section. We believe that there is a wealth of future work on making protocols themselves robust against the effects of intermittent computation. One step in this direction is delay/disruption-tolerant networking [12].

6.2 Hand-Written Assembly

Hand-written assembly poses another challenge for Ratchet. Because it is never transformed into LLVM IR we cannot run our passes to determine if there are idempotency violations, and if so, where they occur. One naive approach is to simply checkpoint before and after the assembly. While that is a good start, there still remains the possibility of idempotency violations between loads and stores within a section of assembly. While we hand-process the assembly files required for newlib, and libgcc it is possible to create a tool to perform this processing automatically [8]. We choose not to write such a tool, since only three assembly files have idempotency violations. Instead we instrument those by hand.

6.3 Ensuring Completion

Extremely short power cycles pose a problem in that they prevent programs from making forward progress and thus never complete execution. Ratchet handles this problem both at compile time and at run time. At compile time, the programmer informs Ratchet of the expected on-time for the device. Ratchet uses this information to make sure that no idempotent section is longer than the expected on time. Note that adding arbitrary checkpoints (*i.e.,* artificial idempotent section breaks) only affects overhead, *not* correctness. The second approach is to use the watchdog timer (commonly available on embedded systems) to insert checkpoints dynamically, when these quick power cycles prevent forward progress. That being said, the longest idempotent sections in our benchmarks (roughly 5000 cycles) require on-times of less than 0.2 ms to expose this issue.

7 Related Work

Ratchet builds upon previous work from three broad categories of research: rollback recovery, intermittently powered computing, and idempotence. We start by examining the history of general-purpose checkpointing and logging schemes, followed by how previous approaches to stretching program execution across frequent power cycles have so far adapted those general-purpose approaches. Lastly, since Ratchet builds upon previous work on idempotence, we cover that previous work.

7.1 Rollback Recovery

Research from the fault tolerance community presents the idea of backward error recovery, "backing up one or more of the processes of a system to a previous state which is hoped is error-free, before attempting to continue further operation" [36]. Backward error recovery systems often choose checkpointing as the underlying technique [11]. CATCH is an example of a checkpointing system that leverages the compiler to provide transparent roll-back recovery [24].

The challenge of checkpointing is identifying and minimizing the state that needs to be protected by a checkpoint [36]. BugNet [31] eliminates the need for full-state checkpoints, while maintaining correctness, by logging the values loaded after an initial checkpoint. This represents a dramatic reduction in overhead as it is unlikely that a process reads the entire contents of memory. Revive [34] reduces overhead further (up to 50%) by using undo logging to record stores instead of loads—assuming a reliable memory.

Ratchet further reduces the bandwidth required by logging/checkpointing by extending idempotency to main

memory. Idempotency tells us that only a subset of program stores actually require logging—those that earlier loads depend on. By checkpointing at stores that alias with earlier loads (since the last checkpoint), Ratchet is able to increase, by orders of magnitude, the number of instructions between log entries/checkpoints.

7.2 Intermittently Powered Computing

Research into fault tolerant computing traditionally targets persistently powered computers. However, new ultra-low-power devices challenge traditional power requirements, making them ideal candidates for energy harvesting techniques. Unfortunately, these techniques provide unreliable power availability which results in fragmented and incorrect execution—violating the assumptions of continuous power and infrequent errors. Because of this, previous work on intermittent computation adopts known rollback recovery techniques, but must adapt them to the properties of this new domain.

7.2.1 Hardware-assisted Checkpointing

Mementos [39] is the first system that attempts to tackle the intermittent computation problem through the use of a one-time (ideally) checkpoint. Mementos uses periodic voltage measurements of the system's energy storage capacitor to estimate how much energy remains. The goal is to checkpoint as late as possible. Mementos provides three ways to control the periodicity of voltage measurement: (1) function triggered: voltage measurement after every function return; (2) loop triggered: voltage measurement after every loop iteration; and (3) timer assisted: a countdown timer added to either of the first two variants that gates whether a voltage measurement is needed. Unfortunately, while Mementos works well when programs write to volatile state only, recent research shows that Mementos is incorrect in the general case [26, 38]. The problem is that Mementos allows for uncheckpointed work to occur. If uncheckpointed work updates both volatile and non-volatile memory, only the non-volatile memory will persist, creating an inconsistent software state. Applying the idea of Mementos to wholly non-volatile memory, as done by QUICKRECALL [17], actually makes the problem worse: in contrast to Flash-based systems, in systems with wholly non-volatile main memory, all program data is stored in non-volatile memory. This makes it a near certainty that a program will write to non-volatile memory after a checkpoint, leading to an inconsistent software state.

Hibernus [3] addresses QUICKRECALL's correctness issues through the introduction of guard bands. A guard band is a voltage threshold that represents the amount of energy required to store the *largest* possible checkpoint to non-volatile memory in the *worst-case* device and environmental conditions. Execution occurs while the voltage is above the threshold, but hibernates when the voltage is below the threshold. Doing this ensures that all work gets checkpointed, at the cost of wasting energy waiting for voltage to build up to the threshold and in the time after a non-worst-case checkpoint—there is no safe way to scavenge unused energy.

Hibernus also improves upon Mementos in terms of performance. Hibernus employs a more advanced analog-to-digital converter (ADC) (for voltage measurement) that allows software to set a threshold value that triggers an interrupt when the voltage goes below the threshold. This removes all software overhead caused by periodically checking the voltage (similar to polling versus interrupts in software).

Comparing voltage-triggered checkpointing systems to Ratchet is difficult. Fortunately, Hibernus provides a detailed performance comparison between itself and Mementos by implementing Mementos on their wholly non-volatile development platform. Their experimental results at 100ms on-time show that Mementos's total overhead varies between 117% and 145%, approximately, depending on the variant of Mementos. Hibernus's polling reduces total overhead to 38%, approximately. In comparison, Ratchet's overhead for the same benchmark program (potentially different inputs and configuration) is a comparable 32%.

In deciding between a one-time, voltage-triggered checkpointing scheme and Ratchet, the biggest factor is the ADC. Mandating an ADC is a non-starter for many existing systems that either do not have one already or systems that have one, but not connected in a way that supports power monitoring. This is a problem even for Mementos, "Voltage supervisors are common circuit components, but most—crucially, including existing prototype RFID-scale devices—-do not feature an adjustable threshold voltage." For future systems and those existing systems that have an ADC capable of voltage monitoring, even the most power efficient ADCs consume as much as 1/3 of the power of today's ultra-low-power microcontrollers [5]. The future is more dire as performance/Watt tends to scale at 2x every 1.57 years for processors (known as Dennard scaling [10]), while ADC's performance/Watt tends to scale at 2x every 2.6 years [19].

7.2.2 Software-only Checkpointing

DINO [26], is a software-only, programmer-driven checkpointing scheme for tolerating an unreliable power source. Programmers leverage the DINO compiler to

decompose their programs into a series of independent transactions backed by data versioning to achieve memory consistency in the face of intermittent execution. To accomplish this, the authors rely on programmer annotated logical tasks to define the transactions. Contrast this with Ratchet which automatically decomposes programs using idempotency information inherent to a program's structure—allowing programmers to ignore the effects of power cycles and mixed volatility memory.

DINO enforces a higher-level property than does Ratchet, namely task atomicity, *i.e.,* tasks either complete or re-execute from the beginning, as viewed by other tasks on the system. This contrasts Ratchet, which allows intermediate writes as long as they maintain idempotency. Enforcing task atomicity happens to also solve the problem of intermittent computation (assuming the programmer defines short tasks), but enforcing a more restrictive property incurs more run-time overhead, 80% to 170%. Note that even though DINO is evaluated on a mixed-volatility system, these numbers are comparable to Ratchet's because DINO's overhead is more dependent on the amount of state a task may update than the volatility of that state.

7.3 Idempotence

Mahlke *et al.* develops the idea of idempotent code sections in creating a speculative processor [28]. The authors construct restartable code sections that are broken by irreversible instructions, which, "modifies an element of the processor state which causes intolerable side effects, or the instruction may not be executed more than one time." They use this notion to define how to handle a speculatively executing instruction that throws an exception and show that they can use the idempotence property to begin execution from the start of the interrupted section and still follow a valid control-flow path.

Kim *et al.* applies the idea of idempotency to data storage, showing that idempotency is useful for reducing the amount of data stored in speculative storage on speculatively multithreaded architectures [21]. The authors note that there are idempotent references that are independent of the memory dependencies that result in errors in non-parallizable code.

Encore is a software-only system that leverages idempotency for probabilistic rollback-recovery [13]. Targeted at systems using probabilistic fault detection schemes, Encore provides rollback recovery without dedicated hardware. The key insight of Encore is probabilistic idempotence: the length of idempotent sections can be increased by ignoring infrequently executed instructions that break idempotence. While this violates correctness—something Ratchet must maintain—the authors realized a performance improvement at the cost of not recovering 3% of the time.

De Kruijf *et al.* [7] presents an algorithm for identifying idempotent regions of code and show that it is possible to segment a program into entirely idempotent regions with minimal overhead. In this initial work, the authors focus their attention on soft faults that do not mangle the register state, noting that registers are usually protected by other means. While soft faults may not corrupt register state, power failures cause the entire register file to be lost.

In follow-on work, de Kruijf *et al.* [6] presents algorithms that utilize the idempotence information generated by the idempotent compiler to better inform the register allocation step of compilation. This allows the compiler to extend the live range of registers. Extending the live range of register values that are live-in to a given idempotent section all the way to the end of that section creates a free checkpoint that enables recovery from side-effect free faults. In contrast, faults on energy harvesting come with significant side effects.

8 Conclusion

Ratchet is a compiler that *automatically* enables the correct execution of long-running applications on devices that run on harvested energy, *without* hardware support. Ratchet leverages the notion of idempotence to decompose programs into a series of checkpoint-connected, re-executable sections. Experiments show that Ratchet stretches program execution across random, frequent, power cycles for a wide range of programs—that would not be able to run completely otherwise—at a cost of less than 60% total run-time overhead. Ratchet's performance is similar to existing approaches that require hardware support or programmer reasoning. Experiments also show that, with more engineering, it is possible to reduce run-time overheads to around 20%.

Ratchet shows that it is possible for compilers to reason about frequent failures and volatile versus non-volatile memory in languages not designed with either in mind. Pushing these burdens on the compiler opens the door for non-expert programmers to code for energy harvesting devices.

Acknowledgment

We thank our shepherd Y. Charlie Hu for his guidance and the anonymous reviewers for their feedback and suggestions. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] compiler-rt runtime [computer software]. Retrieved from http://compiler-rt.llvm.org/, Mar 2015.

[2] ARM. *ARMv6-M Architecture Reference Manual*, Sept 2010.

[3] BALSAMO, D., WEDDELL, A., MERRETT, G., AL-HASHIMI, B., BRUNELLI, D., AND BENINI, L. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters 7*, 1 (2014), 15–18.

[4] BUETTNER, M., PRASAD, R., SAMPLE, A., YEAGER, D., GREENSTEIN, B., SMITH, J. R., AND WETHERALL, D. RFID sensor networks with the Intel WISP. In *Conference on Embedded Networked Sensor Systems* (2008), SenSys, pp. 393–394.

[5] DAVIES, J. H. *MSP430 Microcontroller Basics*, 1 ed. Elsevier Ltd., 2008.

[6] DE KRUIJF, M., AND SANKARALINGAM, K. Idempotent code generation: Implementation, analysis, and evaluation. In *International Symposium on Code Generation and Optimization* (2013), CGO, pp. 1–12.

[7] DE KRUIJF, M. A., SANKARALINGAM, K., AND JHA, S. Static analysis and compiler design for idempotent processing. In *Conference on Programming Language Design and Implementation* (2012), PLDI, pp. 475–486.

[8] DEBRAY, S., MUTH, R., AND WEIPPERT, M. Alias analysis of executable code. In *Symposium on Principles of Programming Languages* (1998), POPL, pp. 12–24.

[9] DEBRUIN, S., CAMPBELL, B., AND DUTTA, P. Monjolo: An energy-harvesting energy meter architecture. In *Conference on Embedded Networked Sensor Systems* (2013), SenSys, pp. 18:1–18:14.

[10] DENNARD, R. H., GAENSSLEN, F. H., RIDEOUT, V. L., BASSOUS, E., AND LEBLANC, A. R. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (Oct 1974), 256–268.

[11] ELNOZAHY, E. N., AND ZWAENEPOEL, W. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers 41*, 5 (May 1992), 526–531.

[12] FARRELL, S., CAHILL, V., GERAGHTY, D., HUMPHREYS, I., AND MCDONALD, P. When TCP breaks: Delay and disruption tolerant networking. *IEEE Internet Computing 10*, 4 (July 2006), 72–78.

[13] FENG, S., GUPTA, S., ANSARI, A., MAHLKE, S. A., AND AUGUST, D. I. Encore: Low-cost, fine-grained transient fault recovery. In *International Symposium on Microarchitecture* (2011), MICRO, pp. 398–409.

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

[15] HICKS, M. Mibench port targeted at IoT devices. https://github.com/impedimentToProgress/MiBench2, 2016.

[16] HICKS, M. Thumbulator: Cycle accurate ARMv6-m instruction set simulator. https://github.com/impedimentToProgress/thumbulator, 2016.

[17] JAYAKUMAR, H., RAHA, A., AND RAGHUNATHAN, V. QUICKRECALL: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers. In *International Conference on Embedded Systems and International Conference on VLSI Design* (2014), pp. 330–335.

[18] JOGALEKAR, A. Moore's law and battery technology: No dice. *Scientific American* (Apr 2013).

[19] JONSSON, B. E. A survey of A/D-converter performance evolution. In *International Conference on Electronics, Circuits, and Systems* (Dec 2010), ICECS, pp. 766–769.

[20] KAHN, J. M., KATZ, R. H., AND PISTER, K. S. J. Next century challenges: Mobile networking for smart dust. In *International Conference on Mobile Computing and Networking* (1999), MobiCom, pp. 271–278.

[21] KIM, S. W., OOI, C.-L., EIGENMANN, R., FALSAFI, B., AND VIJAYKUMAR, T. N. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Trans. Program. Lang. Syst. 28*, 5 (2006), 942–965.

[22] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization* (2004), CGO, pp. 75–86.

[23] LEE, Y., KIM, G., BANG, S., KIM, Y., LEE, I., DUTTA, P., SYLVESTER, D., AND BLAAUW, D. A modular 1mm3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *International Solid-State Circuits Conference Digest of Technical Papers* (2012), pp. 402–404.

[24] LI, C.-C., AND FUCHS, W. Catch-compiler-assisted techniques for checkpointing. In *International Symposium on Fault-Tolerant Computing* (1990), pp. 74–81.

[25] LOWELL, D. E., CHANDRA, S., AND CHEN, P. M. Exploring failure transparency and the limits of generic recovery. In *Symposium on Operating Systems Design & Implementation* (2000), OSDI.

[26] LUCIA, B., AND RANSFORD, B. A simpler, safer programming and execution model for intermittent systems. In *Conference on Programming Language Design and Implementation* (2015), PLDI, pp. 575–585.

[27] MA, K., ZHENG, Y., LI, S., SWAMINATHAN, K., LI, X., LIU, Y., SAMPSON, J., XIE, Y., AND NARAYANAN, V. Architecture exploration for ambient energy harvesting nonvolatile processors. In *International Symposium on High Performance Computer Architecture* (2015), HPCA, pp. 526–537.

[28] MAHLKE, S. A., CHEN, W. Y., BRINGMANN, R. A., HANK, R. E., MEI W. HWU, W., RAMAKRISHNA, B., MICHAEL, R., AND SCHLANSKER, S. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems 11* (1993), 376–408.

[29] MIRHOSEINI, A., SONGHORI, E., AND KOUSHANFAR, F. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered asics. In *International Conference on Pervasive Computing and Communications* (2013), PerComm, pp. 216–224.

[30] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), ASPLOS, pp. 73–84.

[31] NARAYANASAMY, S., POKAM, G., AND CALDER, B. BugNet: Continuously recording program execution for deterministic replay debugging. In *International Symposium on Computer Architecture* (2005), ISCA, pp. 284–295.

[32] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under Unix. In *USENIX Technical Conference* (1995), TCON, pp. 18–29.

[33] POKAM, G., DANNE, K., PEREIRA, C., KASSA, R., KRANICH, T., HU, S., GOTTSCHLICH, J., HONARMAND, N., DAUTENHAHN, N., KING, S. T., AND TORRELLAS, J. QuickRec: Prototyping an Intel architecture extension for record and replay of multithreaded programs. In *International Symposium on Computer Architecture* (2013), ISCA, pp. 643–654.

[34] PRVULOVIC, M., ZHANG, Z., AND TORRELLAS, J. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture* (2002), ISCA, pp. 111–122.

[35] QAZI, M., CLINTON, M., BARTLING, S., AND CHANDRAKASAN, A. A low-voltage 1 Mb FRAM in 0.13 mu m CMOS featuring time-to-digital sensing for expanded operating margin. *IEEE Journal of Solid-State Circuits 47*, 1 (Jan 2012), 141–150.

[36] RANDELL, B., LEE, P., AND TRELEAVEN, P. C. Reliability issues in computing system design. *ACM Computer Surveys 10*, 2 (1978), 123–165.

[37] RANSFORD, B., CLARK, S., SALAJEGHEH, M., AND FU, K. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *Conference on Power Aware Computing and Systems* (2008), HotPower, pp. 5–10.

[38] RANSFORD, B., AND LUCIA, B. Nonvolatile memory is a broken time machine. In *Workshop on Memory Systems Performance and Correctness* (2014), MSPC, pp. 5:1–5:3.

[39] RANSFORD, B., SORBER, J., AND FU, K. Mementos: System support for long-running computation on RFID-scale devices. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS, pp. 159–170.

[40] RAOUX, S., BURR, G. W., BREITWISCH, M. J., RETTNER, C. T., CHEN, Y.-C., SHELBY, R. M., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H.-L., AND LAM, C. H. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev. 52*, 4 (July 2008), 465–479.

[41] SILICON LABS. EFM32ZG-STK3200 zero gecko starter kit.

[42] TEXAS INSTRUMENTS. *MSP430FR59xx Datasheet*, 2014.

[43] TEXAS INSTRUMENTS. MSP432P401x Mixed-Signal Microcontrollers, 2015.

[44] VAN DER WOUDE, J., AND HICKS, M. Ratchet source code repository. `https://github.com/impedimentToProgress/Ratchet`, 2016.

[45] VINSCHEN, C., AND JOHNSTON, J. Newlib [computer software]. Retrieved from https://sourceware.org/newlib/, Mar 2015.

[46] WANG, J., DONG, X., AND XIE, Y. Enabling high-performance lpddrx-compatible mram. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2014), ISLPED '14, ACM, pp. 339–344.

[47] WU, Y.-C., CHEN, P.-F., HU, Z.-H., CHANG, C.-H., LEE, G.-C., AND YU, W.-C. A mobile health monitoring system using RFID ring-type pulse sensor. In *International Conference on Dependable, Autonomic and Secure Computing* (2009), pp. 317–322.

[48] XU, M., BODIK, R., AND HILL, M. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture* (2003), ISCA, pp. 122–135.

[49] ZHAI, B., PANT, S., NAZHANDALI, L., HANSON, S., OLSON, J., REEVES, A., MINUTH, M., HELFAND, R., AUSTIN, T., SYLVESTER, D., AND BLAAUW, D. Energy-efficient subthreshold processor design. *Transactions on Very Large Scale Integration Systems 17*, 8 (Aug 2009), 1127–1137.

[50] ZHANG, H., GUMMESON, J., RANSFORD, B., AND FU, K. Moo: A batteryless computational RFID and sensing platform. Tech. Rep. UM-CS-2011-020, Department of Computer Science, University of Massachusetts Amherst, Amherst, MA, 2011.

[51] ZHANG, W., DE KRUIJF, M., LI, A., LU, S., AND SANKARALINGAM, K. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS, pp. 113–126.