Clank: Architectural Support for Intermittent Computation

Matthew Hicks
Virginia Tech
mdhicks@gmail.com

ABSTRACT

The processors that drive embedded systems are getting smaller; meanwhile, the batteries used to provide power to those systems have stagnated. If we are to realize the dream of ubiquitous computing promised by the Internet of Things, processors must shed large, heavy, expensive, and high maintenance batteries and, instead, harvest energy from their environment. One challenge with this transition is that harvested energy is insufficient for continuous operation. Unfortunately, existing programs fail miserably when executed intermittently.

This paper presents Clank: lightweight architectural support for correct and efficient execution of long-running applications on harvested energy—without programmer intervention or extreme hardware modifications. Clank is a set of hardware buffers and memory-access monitors that dynamically maintain idempotency. Essentially, Clank dynamically decomposes program execution into a stream of restartable sub-executions connected via lightweight checkpoints.

To validate Clank's ability to correctly stretch program execution across frequent, random power cycles, and to explore the associated hardware and software overheads, we implement Clank in Verilog, formally verify it, and then add it to an ARM Cortex M0+ processor which we use to run a set of 23 embedded systems benchmarks. Experiments show run-time overheads as low as 2.5%, with run-time overheads of 6% for a version of Clank that adds 1.7% hardware. Clank minimizes checkpoints so much that re-execution time becomes the dominate contributor to run-time overhead.

CCS CONCEPTS

• **Computer systems organization** → **Embedded hardware**; **Reliability**; **Processors and memory architectures**; • **Hardware** → *Memory and dense storage*;

KEYWORDS

Energy Harvesting, Intermittent Computation, Batteryless Devices, Idempotence

ACM Reference format:
Matthew Hicks Virginia Tech. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080238

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080238

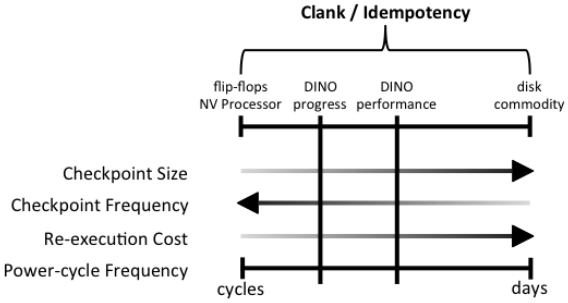

Figure 1: Overview of the tradeoff space for continuous checkpointing approaches to intermittent computation. Previous approaches operate at fixed points in the tradeoff space, depending on the location of non-volatility [28] or programmer defined boundaries [27]. Clank applies to the entire spectrum of volatile/non-volatile memory mixes with appropriate tradeoffs in checkpoint size and frequency and responds dynamically to power-cycle frequency.

1 INTRODUCTION

Architects and circuit designers continually push the boundaries of hardware, creating processors that are smaller, more energy efficient, yet more computationally powerful than those of previous generations. This drive towards smaller chips allows the server rooms of the 1970's to fit on the tip of your finger with recent millimeter-scale devices [24]. The next major transition for processors is smart dust [19], where processors are small enough to be embedded everywhere from shoes [41], to groceries [4], to paint.

Unfortunately, batteries stand in the way of this progress. Batteries are often the largest, heaviest, most expensive, and highest maintenance part of Internet-of-Things (IoT) systems. For example, consider a modern smartphone. In these phones, the main processor is the size of an eraser while the battery is the size of a deck of cards. Waiting for improvements in battery technology is a non-starter as the gap between processor technology and battery technology is growing, not declining. Thus, it is clear that mobile, embedded, and IoT devices of today and the future must shed their batteries if we are to see a dramatic shift in their use.

Energy harvesting techniques have emerged as a viable battery replacement, drawing power from the environment rather than a fixed source. In fact, many devices already integrate these methods to power themselves [4, 11, 44]. Unfortunately, developers for these intermittently powered computers observe that energy harvesting provides insufficient power to perform long-running, continuous, computation [35]. This results in frequent power losses, forcing the program to restart, in hopes of more abundant power next time. Most programs fail to execute completely or correctly with such frequent power failures and requiring programmers to reason about the effects or frequent power failures is error prone [36] and neglects existing programs.

To address this fundamental limitation of running on harvested energy, previous research looks into an array of checkpointing schemes that store partial computation to non-volatile memory. As Figure 1 shows, the key difference in previous work is the location of the non-volatile memory. DINO [27] targets the majority of today's mixed-volatility platforms where Flash/FRAM is the first level of non-volatility. DINO requires the programmer to decompose programs into a series of tasks backed by data versioning. This decomposition is *static*, requiring the programmer to target a specific deployment environment while programming. This forces the programmer to error on the side of forward progress by creating small tasks or on the side of low overhead by creating larger tasks.

Researchers have also investigated future, wholly non-volatile, energy harvesting systems. In the non-volatile processor [28], all memory—down to the flip-flops—is non-volatile memory. In this scenario (as shown in Figure 1), checkpoints are small but frequent. The tradeoff here is that while the programmer does not have to reason about the potential impact of power cycles, the cost in terms of hardware (increased power consumption, increased area, and decreased frequency) results in significant software slowdowns compared to systems with SRAM-based flip-flops.

We envision a generalizable approach that works across a range of volatile/non-volatile memory compositions. To fulfill this vision, we offer Clank. Clank leverages the notion of idempotency to *automatically* and *dynamically* make programs robust against frequent, random, power cycles. Using idempotency means that Clank applies to devices with off-chip non-volatile memory and to wholly non-volatile devices—and the range of devices in-between. Idempotency is a property that a sequence of instructions has that means that they are arbitrarily restartable. Previous work shows that maintaining the value of the inputs to a sequence of instructions is all that is required to make that sequence restartable [29]. We adapt these previous insights, targeted at registers, to non-volatile memory. Clank includes lightweight hardware that dynamically tracks idempotency, buffering idempotency violations in volatile memory. When the buffer is full with idempotency-violating memory writes, Clank creates a checkpoint (i.e., saves all modified volatile state to non-volatile memory), essentially locking-in the effects of the previous instruction stream. By doing this, Clank dynamically decomposes unmodified programs into a continuous sequence of idempotent instruction streams, connected by lightweight checkpoints. Clank extends the natural idempotency of a program through careful use of volatile buffers. To maximize the benefit of Clank's buffers, Clank includes a modified compiler that encodes in memory access instructions if there is no risk of them violating idempotency. Clank applies to a range of devices, with more frequent, but smaller checkpoints as non-volatility moves towards the core and larger, but fewer checkpoints as non-volatility moves off-chip.

To show that Clank is an effective way to stretch program execution across power cycles, we implement it for the next wave of energy harvesting devices [3], where main memory is non-volatile, but the rest of the system is volatile. This represents a node somewhere in the middle of the spectrum created by the non-volatile processor and DINO, also targeted by Hibernus [2] and Ratchet [40]. We implement Clank for an ARM Cortex M0+ processor. Experiments with 23 embedded systems benchmarks show that Clank is able to limit software run-time overhead to as little as 2.5% with the aid of in-hardware buffers and compiler support.

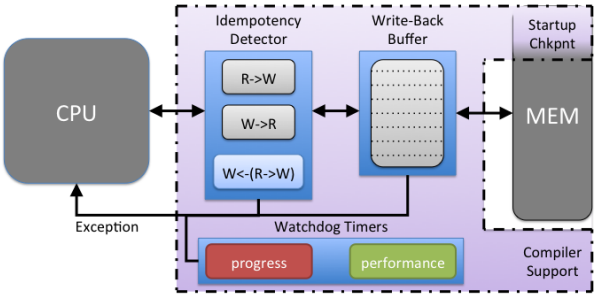

Figure 2: The central component of Clank is a set of hardware buffers that track memory accesses, identifying idempotency violations. Clank also has the ability to condense several idempotency violations into a single checkpoint by storing idempotency-violating writes in a write-back buffer. Optional components include two watchdog timers to ensure forward progress and prevent overly large idempotent sections and a compiler to insert checkpointing routines and identify memory accesses that the hardware can ignore for idempotency purposes.

Clank reduces *total* (see Section 2.1) run-time overhead almost an order-of-magnitude compared to all previous approaches, while requiring no programmer intervention, involving only minor hardware modifications, and reducing memory footprint by 4x compared to software-based, continuous checkpointing [40] and data versioning [5, 27] approaches, and dynamically adapting to environmental conditions. Beyond this, Clank represents six high-level contributions:

- We create the first hardware support for idempotency tracking.
- We implement both an FPGA implementation using ARM source code and a cycle-accurate ARM Cortex M0+ simulator. We use these implementations to explore Clank's design space.
- We formally verify the correctness of our Verilog Clank implementation using bounded model checking. We also dynamically verify the correctness of every experimental trial on our Clank simulator using a reference monitor.
- We explore Clank's performance on both wholly non-volatile and mixed volatility systems and compare its performance to previous approaches.
- We address the problems of oversized idempotent sections: 1) due to power cycles too short to make a forward progress through a checkpoint (runt power cycles) or 2) due to increasing the cost of re-execution beyond the cost of taking a superfluous checkpoint (overhead inversion).
- We are the first work in the idempotency space to create idempotent sections long enough to show that there is a turning point where re-execution cost is greater than checkpointing cost.

2 CLANK OVERVIEW

Clank automatically and efficiently stretches program execution across frequent, random, power cycles. Figure 2 shows Clank's key components and how they fit into the development of software and run-time operation of a batteryless device. Clank consists of both hardware and compiler components. Clank hardware precisely tracks

memory accesses for idempotency violations, while Clank's compiler takes unmodified and unrestricted C programs and produces a binary that contains routines for checkpointing software state and restarting execution after a power cycle. The binary that Clank produces is similar to what an unmodified compiler produces, but with a small amount of reserved memory and some memory access instructions replaced with versions that carry semantic information to hardware; namely, that the specified memory access is *guaranteed not* to affect idempotency. The resulting binary is loaded (using traditional methods) to an energy harvesting device.

At run time, every time the device has enough power to turn on, it first executes the compiler-inserted restart routine. The restart routine loads the last checkpoint or starts execution from `main()` if no checkpoint exists. While software executes, Clank updates its buffers and checks for idempotency violations. When the detector determines a violation occurred, it signals the Write-back Buffer. The Write-back Buffer holds address/value tuples that, if written to non-volatile memory would violate idempotency of the currently executing section. This allows Clank to delay a checkpoint until enough idempotency violations occur to overflow the Write-back Buffer. When the Write-back Buffer does overflow, it sends an exception to the processor. The processor delivers the exception to Clank's compiler-inserted checkpoint routine, which saves volatile software state to non-volatile memory, resetting idempotency relationships, and passes control back to the program.

Clank includes a set of watchdog timers that help ensure that a program makes forward progress even with arbitrarily short power-on times and to minimize run-time overhead by balancing checkpoint and re-execution overhead.

2.1 Design Principles

The primary goal of Clank is to extend program execution across frequent and random power cycles correctly. As Section 8 details, there are many viable ways to meet this goal. To choose between the different approaches we provide a set of design principles that apply to all known approaches:

- Avoid slowing down the hardware: reducing the maximum frequency of the processor directly slows the program. This is one drawback of approaches that push non-volatility down to the flip-flops [28]: non-volatile flip-flops switch slower than their volatile counterparts [20]. Clank maintains the processor's original maximum frequency.
- Avoid executing additional instructions: every extra instruction that software has to execute to save its state is one less instruction that is able to move software state forward. Clank requires less than 1% extra instructions due to checkpointing, even for small buffer sizes.
- Avoid increasing hardware's power consumption: energy spent on added hardware is not available to move software state forward. This is a drawback of approaches that mandate significant hardware changes [20, 28, 30]. Similarly, approaches that use an analog-to-digital converter (ADC) to measure the voltage (to determine if they are about to lose power) [2, 18, 37] lose 40% of their energy to the ADC [6].
- Avoid lengthy re-execution: as checkpointing schemes advance, the overhead due to re-execution increases. In fact, for many configurations of Clank, re-execution overhead dominates and reducing checkpointing overhead only increases the total run-time overhead.

The key here is to minimize the *total* impact of the proposed system on the software. These four design principles are important because they all impact the run-time overhead of software running on energy harvesting devices. Section 7 contains experimental results that show how Clank performs in relation to each of these principles.

2.2 Why Hardware?

Given the large body of work that exists for the compiler community on compiler analysis passes for idempotency [7–10, 45], it is known how to decompose programs into a series of idempotent sections statically using only the compiler as opposed to the dynamic decomposition of Clank. Why modify the hardware? The answer lies in the difference between static (i.e., compile time) and dynamic alias analysis. Alias analysis [38], a core component of idempotency analysis, is a challenge even for today's most advanced compilers (e.g., LLVM). For example, the alias analysis passes that earlier idempotency works use are intraprocedural (i.e., within a single function). This results in a best case of a checkpoint every function call and return [40]. Moving to interprocedural analysis is a non-starter due to its lack of precision that results in many artificial aliases, each of which causes a checkpoint. Making matters worse, common programming features such as pointers result in large alias sets, which results in in-function checkpoints. Comparing the overhead of Clank to Ratchet [40], an idempotence-based compiler-only intermittent computation approach, it is clear that most aliases reported by even intraprocedural analysis are non-existent at run time. Clank thrives on this asymmetry.

3 CLANK HARDWARE

To stretch program execution across power cycles, Clank dynamically decomposes programs into a series of restartable instruction sequences. To connect two restartable instruction sequences, Clank uses a checkpoint that makes each idempotent sequence independent of the memory accesses in all previous sequences. The key to restartability is maintaining idempotence during execution. This is the role of Clank's hardware idempotency detectors. Clank's hardware is tasked with dynamically tracking idempotency and informing the checkpoint routine in software when a checkpoint needs to be created so that execution can move to the next restartable section. The challenge is doing this as efficiently as possible, i.e., with a few small checkpoints. Eventually, we introduce the idea of adding checkpoints to balance re-execution overhead and checkpoint overhead, but the ideal is to remove the need for any checkpoints and only insert them as needed to minimize total run-time overhead.

In this section, we describe the design and operation of Clank's hardware components. We follow this with a discussion of a series of optimizations that reduce Clank's hardware overhead and increase its effectiveness. Lastly, we cover our solution to a problem common to all transaction systems, the output commit problem.

3.1 Key Components

As Figure 3 shows, the primary components of Clank are buffers (Read-first and Write-first) and idempotency detection logic. To

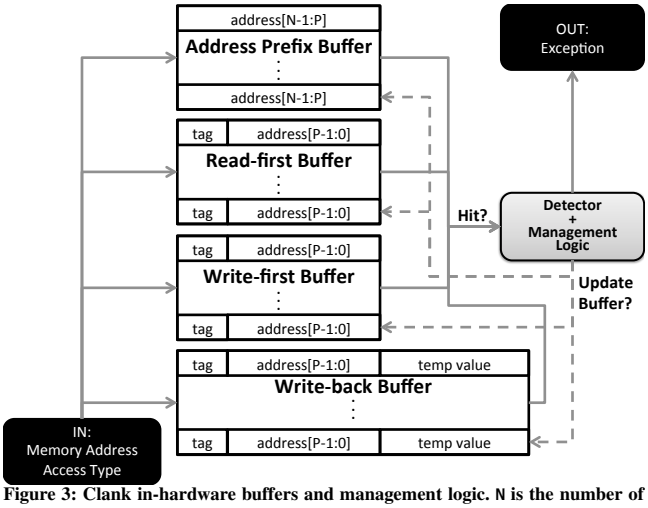

Figure 3: Clank in-hardware buffers and management logic. N is the number of bits in a memory address, e.g., a 128*K* memory requires 17 bits. tag is an index into the Address Prefix Buffer. Buffers are queried and updated in parallel.

this, we add a Write-back Buffer to reduce checkpoint frequency by delaying idempotency violating writes by storing updated values in a volatile buffer (as opposed to overwriting the original value in non-volatile memory). To reduce the hardware overhead of Clank, we leverage spatial locality of memory accesses by using an Address Prefix Buffer that stores upper bits of memory address, allowing the other buffers to store only a tag and the lower bits of each address—effectively acting as de-duplication of common address prefixes. Lastly, Clank adds two watchdog timers that help to extract more performance by balancing checkpoint and re-execution overhead and ensure that programs make forward progress during repeated runt power cycles. We describe each hardware component next.

3.1.1 Idempotency Detection. An idempotency violation consists of a write to a memory address that was first accessed by a read instruction (during a given section of execution). In simpler terms, think of each memory location as being either write dominated or read dominated. Given a sequence of instructions, if the first access to a memory address is a write, then that location is write dominated. Conversely, if the first access to a memory address (in a given instruction sequence) is a read, then that location is read dominated. Expressing idempotency in these terms, an idempotency violation is a write to a memory location that is read dominated. Note that every memory address accessed during an instruction sequence must either be write or read dominated.

From this definition, it follows that the hardware must track, for each memory address accessed, if the address is write or read dominated. For this, Clank includes a buffer for each list;[1] the Read-first Buffer holds read dominated addresses, while the Write-first Buffer holds write dominated addresses. The last piece of the idempotency violation detector is the detection logic. The detection logic (which also serves as buffer management logic) checks the buffers on every memory access. If the address of the access is *not* in either buffer, then it gets added to the appropriate buffer, given the access type (read or write). On the other hand, if the access is a write and the address of the access is already in the Read-first Buffer (i.e., r->w in Figure 2), then the detection logic signals an idempotency violation. All other combinations of access types and Clank buffer states result in that memory access being ignored.

Given that these idempotency tracking buffers are hardware resources, they have limited capacity. When either buffer is full and an address needs to be added (i.e., overflow), the detection logic signals a full condition. Clank treats full conditions as an idempotency violation, thus handled by Clank's checkpointing routine. After Clank commits the checkpoint to non-volatile memory, it returns control to the instruction that caused the overflow-inducing memory access; this time, with buffers empty. Section 3.2 describes optimizations to this policy that dramatically reduce the number of buffer full exceptions—hence fewer checkpoints—while still maintaining correctness.

Buffer overflows create checkpoints, but whether idempotency violations do depends on the presence of other Clank hardware components, namely, the Write-back Buffer. If there is no Write-back Buffer, then Clank treats idempotency violations the same as a buffer overflow, i.e., it results in a checkpoint. But, if there is a Write-back Buffer, then the Write-back Buffer determines if a checkpoint is needed based on if it has room to store the idempotency-violating write.

3.1.2 Write-back Buffer. As shown in Section 7.1, creating a checkpoint for every idempotency violation is costly. To avoid this, we introduce a buffer that allows Clank to delay committing idempotency violating writes to non-volatile memory. This allows Clank to stretch the execution of an idempotent section past its natural limits. Our approach takes advantage of volatility in that idempotency-violating writes stashed in the Write-back Buffer automatically disappear during power-off periods—free rollback via redo logging [32].

When the detection logic signals an idempotency violation (which only happens for a write), the Write-back Buffer first checks if it already holds the address. If so, the value associated with that address is updated. If not and there is room, the Write-back Buffer adds the address and the new value—the value first read remains, unchanged, in non-volatile memory. If not and there is no room in the Write-back Buffer to store the address/value pair, the Write-back Buffer signals an exception. The exception is handled in much the same way as Read/Write-first Buffer overflows, but Clank must also update the non-volatile memory with the values stored in the Write-back Buffer. Doing so is complicated by the common case of power failures in that it requires double buffering to prevent power-cycle-induced corruption: first Clank copies the addresses and values from the Write-back Buffer (possibly volatile) to a scratchpad in non-volatile memory set aside by Clank's compiler. Second, Clank creates a new checkpoint so that a power cycle will restart post-copy to the scratchpad. Third, Clank overwrites the given addresses with their new values. Note that copying between non-volatile memory locations (the scratchpad and the program's memory) is naturally idempotent. Lastly, Clank creates a checkpoint so that the program will restart with the new values in place, otherwise, a restart while the Write-back Buffer is being copied to the scratchpad for the *next* section will result in an inconsistent state.

[1] Note that the only component required to track idempotency is a Read-first Buffer. All other components exist to improve performance.

Section 3.2 describes optimizations relating to the Write-back Buffer that relieve pressure on the Write-first and Read-first buffers—making them appear larger.

3.1.3 Address Prefix Buffer. Until now, the abstraction of Clank presented had each buffer tracking entire memory addresses. We observed that many of the entries had the same most significant bits due to the spatial locality of memory accesses during the short bouts of execution seen in intermittent execution. This is especially true with the small memories of batteryless devices which may have up to 256KB of system memory, but an address of 32-bits. This is an opportunity for de-duplication to free resources to store unique data.

We add the Address Prefix Buffer to take advantage of this observation. The Address Prefix buffer replaces duplicate address prefixes in Clank buffer's entries with a small tag that points to that prefix as stored in the Address Prefix Buffer. For example, in the version of Clank that we build (Section 6), each buffer stores the 6 least-significant bits of a memory address and a 2-bit tag that points to one of 4 Address Prefix Buffer entries, each storing the 24 most-significant bits of a memory address. Compare this 8-bit requirement to each buffer entry requiring 30 bits.[2] Thus, the Address Prefix Buffer makes each bit of added memory for Clank's in-hardware buffers more effective by storing more unique information. The downside of the Address Prefix Buffer is that it is another in-hardware buffer that can fill, resulting in added checkpoints.

3.1.4 Watchdog Timers. Another optional hardware component in Clank is a pair of watchdog timers. A watchdog timer is an in-hardware timer that counts down every clock tick, starting from a software-specified load value. When a watchdog timer reaches 0, it signals an exception. There are two watchdog timers in Clank, one that ensures that software makes forward progress and one that limits the number of cycles between checkpoints to balance checkpoint and re-execution overhead.

Progress. The most critical watchdog timer is tasked with ensuring that software makes forward progress within a power cycle. Harvested energy is unpredictable; therefore, it is possible that Clank will create idempotent sections larger than the average power-on time (i.e., a runt power cycle). Without a watchdog, a long idempotent section would continually restart, only to *never* finish before the next power cycle. Eventually, since power-on time is probabilistic, there may be a long enough power cycle, but waiting for only long power cycles wastes all of the other ones.

Clank reduces the number of wasted power cycles due to runt power cycles through the addition of a Progress Watchdog. The role of the Progress Watchdog is to break long idempotent sections by inserting superfluous checkpoints. This means that even with very short power-on times, software will make forward progress. The progress watchdog is dynamic and automatic—able to respond to changing power conditions *without* user or programmer intervention. Initially, the progress watchdog is disabled. This both saves power and avoids adding run-time overhead due to unneeded checkpoints. The progress watchdog is enabled by the restart routine run at the beginning of a power cycle. The restart routine checks the value of a variable that is `0` if there was a checkpoint last power cycle and `1` if not. In the `0` case, the restart routine sets the variable to `1` and leaves the Progress Watchdog disabled. In the `1` case, the restart routine sets the load value of the watchdog timer and enables it. The load value depends on if the existing load value in non-volatile memory is non-zero (a non-zero load value represents the case where the Progress Watchdog was enabled last power cycle, but still no forward progress was made). If non-zero, then the current load value is divided in half, otherwise, a default value is loaded. The last bit of bookkeeping comes during the first checkpoint of a power cycle. During the first checkpoint, Clank disables the progress watchdog, sets its load value to `0`, and then clears the variable (in non-volatile memory) that indicates whether a checkpoint happened this power cycle.

Performance. The second watchdog timer offered by Clank targets maximizing performance. As shown in Section 7.4, Clank is so successful at increasing the number of instructions between checkpoints that the overhead due to re-execution can dominate that of checkpointing (overhead inversion). Thus, minimizing total run-time overhead requires limiting the number of instructions between checkpoints such that re-execution and checkpoint overhead are balanced. That is the goal of Clank's Performance Watchdog.

Unlike the Progress Watchdog, the Performance Watchdog's load value is fixed. To determine the load value for a given program we assume the ideal scenario where there are no program-induced checkpoints. Given this ideal, it is possible to calculate the optimal watchdog value given the average on time, restart overhead, and the average number of cycles required to save a checkpoint. The results in Section 7.4 indicate that the optimal watchdog value balances checkpoint and re-execution overhead. Any run-time deviations from the ideal result in slight overhead imbalances that yield near-optimal total overhead.

Clank's compiler-inserted start-up routine loads the predefined timer value into the Performance Watchdog and enables it. Then, the checkpoint routine reloads the load value to the Performance Watchdog (i.e., resets the watchdog) every checkpoint—it is always enabled. If the Performance Watchdog ever reaches zero, it signals an exception, which results in a checkpoint.

3.2 Optimizations

So far, the descriptions of Clank's detection and buffer management logic has been kept simple to aid understanding. But there are several observations that we make that lead to performance optimizations. The optimizations result in fewer checkpoints by reducing pressure on Clank's hardware buffers—making the structures seem larger than their hardware cost—and delaying, to the last moment, when Clank throws the checkpoint-inducing exception. This increases performance relative to a given total buffer capacity while maintaining correctness.

3.2.1 Ignore False Writes. One optimization opportunity is ignoring writes to non-volatile memory addresses that do *not* actually change the value of that memory location. For example, if memory location `0x00001234` contains `5` and, for whatever reason, software attempts to write `5` to that location, Clank ignores that write for purposes of idempotency violation detection. Note that the write still

[2] Clank tracks memory accesses at the word level, hence 30-bit addresses. This is still safe as a byte read/write marks the entire word that that byte belongs to, but in cases where byte-level accesses dominate, there is a risk of adding unneeded checkpoints.

causes updates to the write buffer if, in fact, the location is write dominated. This optimization requires saving values from addresses as they are read which we do by co-opting the Write-back Buffer. The effect of this optimization is reduced pressure on the Write-back Buffer, resulting in fewer checkpoints.

3.2.2 Remove Duplicate Entries. What happens when an idempotency violation occurs but is buffered by the Write-back Buffer? In such cases, it is possible to clear the address from the Read-first Buffer (because addresses involved in idempotency violations must be in that buffer) and ignore all future accesses to that address. In general, memory addresses should only reside in a single buffer. This optimization improves performance by relieving pressure on the Read-first Buffer—Clank's most critical resource.

3.2.3 No Write-first Overflows. In the description of the Write-first and Read-first buffers, we stated that Clank signals an exception when either buffer overflows. There is an opportunity to reduce checkpoints by ignoring overflows to the Write-first Buffer. Realize that the Write-first Buffer stores addresses of write-dominated memory locations. Thus, Write-first entries only serve to avoid false detections of idempotency violations. So ignoring new entries to a full buffer just opens the door to false detections—Clank operates correctly, just pessimistically. Since both a full buffer and a false detection result in a checkpoint, it is better for performance reasons to delay the checkpoint by ignoring write buffer overflows and, instead, wait for a full read buffer or an idempotency violation to cause the checkpoint.

3.2.4 Ignore TEXT Segment Accesses. Program binaries consist of several sections, e.g., `text`, `heap`, and `stack`. We observe that, for many programs, the `text` section is rarely the cause of idempotency violations. Thus, there is an opportunity to reduce pressure on the Read-first Buffer by ignoring all reads to addresses within the `text` section.

To maintain correctness and support esoteric behaviors like self-modifying code, every write to a memory address within the `text` section causes Clank to create a checkpoint. The asymmetry between the reads and writes to memory addresses in this region makes this optimization effective, but only for small Read-first Buffers (see Section 7.2).

3.2.5 Latest Checkpoint. The last optimization increases the number of instructions between checkpoints by delaying checkpoints for as long as possible while maintaining correctness. As opposed to checkpointing when we can no longer track memory accesses, we stop tracking memory accesses, allow any reads to go through, and checkpoint before the first write. The observation driving this is that only writes break idempotency. For example, in the case where the Read-first Buffer fills, we can allow arbitrary/untracked reads without a checkpoint and still maintain correctness. As long as Clank checkpoints *before* the first write that follows a fill of any of Clank's buffers, Clank maintains correctness. This increases the effective size of the Address Prefix and Read-first buffers.

3.3 Output Commit Problem

All replay and transaction systems suffer from the output commit problem [12]. The output commit problem occurs when a system creates an output that it cannot take back or does not want to be replayed. Clank follows previous approaches and commits outputs, at the word level, as soon as they occur by surrounding on output with checkpoints. This minimizes the likelihood that a power cycle occurs between writing an output and the ensuing checkpoint. To know what constitutes an output, Clank relies on the device's memory map. Clank treats any writes outside of the physical memory range of the device as an output.

A more robust solution to the output commit problem requires measuring power to ensure enough energy remains to commit the output or side-stepping the problem by making protocols and devices robust to disruptions and replays. Our benchmarks indicate that most programs do not require such complex solutions, but in the rare cases where programs do require larger-granularity guarantees, Clank provides a foundation for programmers to implement a more advanced solution.

4 CLANK COMPILER

Like many embedded systems, energy harvesting devices execute from a single binary. This means that it is reasonable to expect any software loaded onto an energy harvesting device to be recompiled. This opens the door for a compiler component to Clank that works synergistically with Clank's hardware components to increase their effectiveness—without burdening programmers. To this end, we provide a compiler that not only adds the required support routines to respond to Clank's hardware events but actually reduces pressure on Clank's hardware buffers by limiting the memory accesses that the idempotency detector needs to react to. We describe the individual compiler components below and Section 7.1 shows the effect on run-time overhead that Clank's relatively simple compiler affords.

4.1 Checkpoint

The primary compiler-inserted software routine required by Clank is the checkpoint routine. The basic checkpoint routine is simple: back-up to non-volatile memory all program state residing in volatile memory that was modified since the last checkpoint. In the case of wholly non-volatile microcontrollers, this consists of general-purpose register values (including the stack pointer, link register, and program counter) and device configuration/status registers. The tricky bit is determining which of the two checkpoint slots to write to in non-volatile memory. Two slots are required for cases where the power cycles in the middle of writing a checkpoint. Given that it takes many cycles (e.g., 40 for our implementation) to write an entire checkpoint to non-volatile memory, but the act of committing a checkpoint must appear atomic, we rely on double buffering: a checkpoint only commits once a variable we call `checkpoint pointer` points to it. The checkpoint routine updates `checkpoint pointer` as the last thing it does. Thus, any power cycles in the middle of the checkpoint routine cause the software to restart execution back at the beginning of the section that was just being checkpointed—which still has its checkpoint stored safely in the other checkpoint storage slot.

After saving volatile state to non-volatile memory, the checkpoint routine sets the `checkpoint pointer` to the start of the phase 2 checkpoint routine. The second phase of checkpointing resets all of Clank's in-hardware buffers. The checkpoint routine then resets

the Performance Watchdog and disables the Progress Watchdog. Finally, the checkpoint routine updates `checkpoint pointer` with a program address and returns control back to the program, at the point it was interrupted.

In cases where there is a Write-back Buffer, checkpoints get more complicated. In this case, the checkpoint routine must start by copying the list of address, value tuples from the Write-back Buffer to a compiler-reserved location in non-volatile memory. If a power-cycle occurs during or directly after this process, those values are safely ignored and the section that was just finishing execution is restarted. Once all the tuples are double-buffered in non-volatile memory, the previously described checkpointing process may start. Having a Write-back Buffer also preempts the second phase of checkpointing; before the second phase starts, the program values in the non-volatile version of the Write-back Buffer are updated.

4.2 Start-up

The first thing a device does is start up. Normally, embedded software incorporates a start-up routine that initializes the devices attached to the processor, fixes the stack, and then passes control to `main()`. Stretching program execution across power cycles does *not* require a more complex start-up routine, just a different one. The start-up routine for energy harvesting devices must be able to restore state from a checkpoint in non-volatile memory and use data in that checkpoint to initialize the processor and device state. To reduce the complexity of the start-up routine, the compiler creates the first checkpoint in memory. This checkpoint essentially initializes the state and runs `main()`—a traditional boot. Using this scheme, the start-up routine does not have to worry about whether this is boot 1 or boot N.

The start-up routine proceeds as follows:

(1) read the last valid checkpoint variable to know which checkpoint to load
(2) read the `checkpointed last cycle` variable to determine if we need to enable the Progress Watchdog
(3) if the Progress Watchdog is enabled, read its load value and divide by 2, then update the load value both in the watchdog and in memory
(4) enable the Progress Watchdog
(5) load the checkpoint into the processor's registers

4.3 Bridging the Semantic Gap

While the heart of this paper is architectural support for automatically enabling existing programs to execute intermittently, there is an opportunity for compiler analysis to make Clank hardware more effective. Observe that Clank hardware examines all memory accesses, but from software's perspective, many memory accesses have no impact on idempotency. This presents an opportunity for a compiler to bridge the semantic gap between software and hardware: the compiler can inform the hardware, via special memory access instructions that a memory access is guaranteed *not* to impact the idempotency of the currently executing sequence of instructions. By doing this, Clank's in-hardware buffers appear larger as they no longer need to track the addresses of many memory accesses.

The challenge is that the dynamic idempotency analysis of Clank makes it impossible to apply directly existing static idempotency analysis passes during compilation. The issue is that current static idempotency analysis passes are intraprocedural, but Clank's dynamic analysis is able to span multiple functions. Thus, while memory accesses may not affect idempotency inside a given function, it is possible that the same access will affect the idempotency of past or future functions.

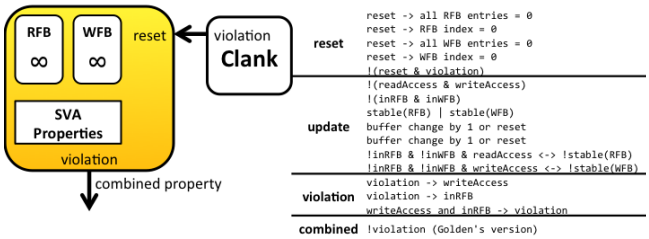

Figure 4: Formal verification setup. We formally verify an infinite-resource idempotence reference monitor using the listed properties. We then verify a property on the reference monitor combined with a high-performance implementation of Clank: there is no memory access pattern that can cause a violation in the reference monitor before Clank signals a violation.

Our goal here is only to show that there is benefit to combining compiler analysis with hardware support; we see this as an area of future exploration. Clank takes the first step in this direction by identifying memory accesses that are Program Idempotent. A Program Idempotent memory access is one where it is impossible, under any possible re-execution or program control flow to cause an idempotency violation. More simply, Clank identifies memory accesses where there is never a write that follows a read. If such a pattern exists, it is possible for a power cycle to occur after the write, but before a checkpoint. This causes the read to get a different value. Two access patterns that satisfy this constraint are read-only locations and initial write(s) followed by only reads. Therefore, Program Idempotence can be summed-up by the access pattern `W*->R*`.

Clank's analysis of Program Idempotent memory accesses is easy to implement by profiling execution, but very limited in the amount of memory accesses that it can identify as ignorable. We see a range of future work on more sophisticated analyses that provide better results; our goal is to provide a proof-of-concept. One can imagine a compiler that inserts checkpoints to make analysis easier or to break the relationship between memory accesses before and after the checkpoint to make it possible to ignore more accesses.

5 VERIFICATION

Our goal here is to formally verify that our implementations of Clank (both hardware and software) preserve idempotency. Proving that our implementation preserves idempotency, for all possible memory access patterns, allows us to be sure that it is *impossible* for a power cycle—coming at any time—to result in a re-execution that is inconsistent with a single, continuous execution. So, while our goal in the previous sections was to relax constraints as much as possible to improve performance, our goal here is to validate that our high-performance implementations are correct (i.e., there are no implementation bugs and all optimizations preserve program semantics no matter the power cycle or memory access pattern).

The challenge is that our implementations are optimized for performance, not verification. Adding the complexity required to make

verification tractable results in large and slow hardware. Another challenge is that the property that we want to verify requires keeping a complete history of memory accesses made during the execution of an idempotent section, but our implementation has the ability to ignore certain accesses.

To address these challenges, we rely on an easy-to-verify, infinite resource reference monitor as the centerpiece for verification. Figure 4 shows our verification setup. We start by proving that this reference monitor is correct using the 15 properties shown in Figure 4. To prove these properties correct, we implement them as SystemVerilog assertions and use a bounded model checking tool to verify that they hold for any possible memory access sequence of length 32 or less [22]. When proven correct, we know that under any possible power cycle and program memory access pattern, the reference monitor tracks idempotence correctly. Then we connect the reference monitor to each of our high-performance implementations and verify that for all possible power cycle and memory access patterns up to 32 cycles, the high-performance implementation will always signal an idempotency violation (i.e., initiate a checkpointing operation and causing the reference monitor to reset) before the reference monitor signals an idempotency violation. Doing this proves that the high-performance implementation is correct.

32 is our bound for the model checker. Increasing the bound results in an exponential increase in the model checker state space. At a bound of 32, we are able to size Clank's buffers so that the generated model is complete, i.e., there is no unique sequence of events greater than 32 cycles not already represented and verified in the first 32 cycles. If we assume that increasing Clank's buffer sizes does not interfere with correctness, then Clank is correct.

6 IMPLEMENTATION

To demonstrate that Clank works on today's energy harvesting devices, we implement Clank for the ARM Cortex-M0+ processor. We select this processor because its ultra-low-power draw makes it a popular choice for energy harvesting devices. The processor also has its source code available, which bolsters our implementation. ARM Cortex processors are currently used in the lowest power System-on-Chips [17, 23, 24]. The Cortex-M0+ has a two-stage pipeline, executes 16-bit instructions[3] with a 32-bit data word. The processor we implement has a 32-cycle hardware multiply unit. Being a low complexity processor, there is no memory management unit (i.e., all addresses are physical) and there is only a single privilege level.

Our implementation consists of three artifacts: 1) an FPGA prototype built on Xilinx's VC709 development board [42] using ARM-provided source code for the Cortex-M0+ [1]; 2) a cycle-accurate instruction-set simulator for ARMv6-M enhanced with a timing model that matches the Cortex-M0+ we implement on the FPGA; and 3) a Clank policy simulator that takes in a memory access log from a benchmark program (an output of our instruction set simulator) and outputs a detailed run-time overhead analysis along with a dynamic verification of correctness. We use the hardware implementation to evaluate the area, frequency, and power overheads of Clank (see Section 7.3) and to validate our simulators. We use the simulators to evaluate software run-time overheads due to Clank and for a design space exploration of that examines the tradeoff of hardware and run-time overheads at almost 1,000,000 configurations for each of our 23 benchmarks.

[3] A limited number of special-purpose instructions require 32-bits.

Benchmark	**Running Time** (ms)	**Size** (bytes)	**Size Increase**
adpcm_decode	2,062	382,869	0.04%
adpcm_encode	2,237	1,408,021	0.01%
aes	12	44,269	0.34%
basicmath	64,586	51,850	0.29%
bitcount	24,894	42,249	0.36%
blowfish	11,976	358,609	0.04%
crc	2	38,881	0.39%
dijkstra	26,320	79,601	0.19%
fft	16,336	46,245	0.33%
limits	<1	1,360	11.18%
lzfx	36	112,101	0.14%
overflow	<1	1,296	11.73%
patricia	9,565	392,513	0.04%
picojpeg	1,553	104,169	0.15%
qsort	17,601	313,846	0.05%
randmath	<1	612	28.84%
rc4	11	7,484	2.03%
regress	28	864	17.59%
rsa	1	43,533	0.35%
sha	6,222	3,284,830	0.00%
stringsearch	4,859	55,321	0.27%
susan	6,417	75,038	0.20%
vcflags	<1	1,800	8.44%
average	8,466	297,712	0.05%

Table 1: Cycle count and code size (in bytes) of MiBench2 [14] benchmarks. Also shown is the percent increase in code size for a Clank configuration representative of those listed in Table 2, including both watchdog timers.

7 EVALUATION

Here we evaluate Clank's impact on existing programs and a commercial low-power processor, showing that it cheaply stretches existing programs across power cycles correctly and efficiently. To evaluate Clank we use the 23 programs listed in Table 1 that comprise the MiBench2 IoT benchmark suite [14]. Our goal in selecting benchmarks it to build a diverse set of long-running programs that one would expect to run on a resource constrained system. Our set of benchmarks is the most diverse and extensive in the literature. When looking at overheads, keep in mind that most of the benchmarks fail to complete execution within a single power cycle—i.e., they are impossible to run intermittently without Clank.

7.1 Clank Design Space

Hardware designers looking to add Clank to their processor have a range of buffer sizes and buffer configurations to choose from. The only component of Clank required is a Read-first Buffer; everything else just boosts performance at the cost of increased hardware complexity. Complicating matters is the fact that each buffer's entries are different sizes and perform different functions. Thus, the goal of this section is to explore the software/hardware overhead tradeoff space of Clank to help hardware designers select an optimal configuration.

Because implementing and running millions of configurations on our FPGA implementation is impractical, we employ our cycle-accurate instruction set simulator [15] in combination with our Clank policy simulator [16] for this experiment. We use the instruction set simulator to generate a memory access log for each benchmark, then pass that log and the desired Clank buffer configuration to our

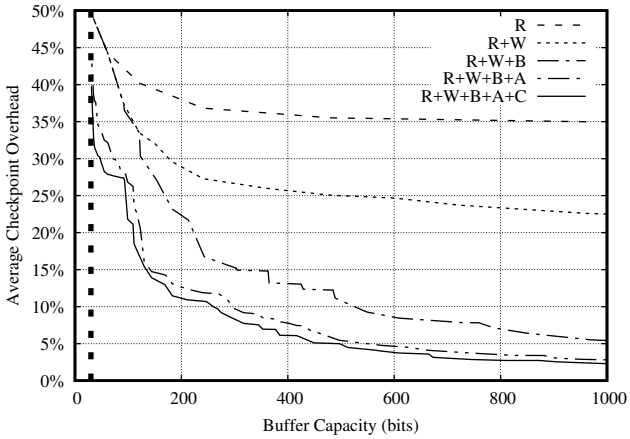

Figure 5: Pareto frontiers of Clank hardware size vs. average (across all 23 benchmarks) run-time overhead for five different versions of Clank. `R` represents only having a Read-first Buffer; `R+W` adds the ability to have a Write-first Buffer; `R+W+B` adds the ability to have a Write-back Buffer; `R+W+B+A` adds the ability to have an Address Prefix Buffer; and `R+W+B+A+C` shows the impact of ignoring Program Idempotent memory accesses. The dashed vertical line represents the hardware requirement of a single Read-first Buffer entry.

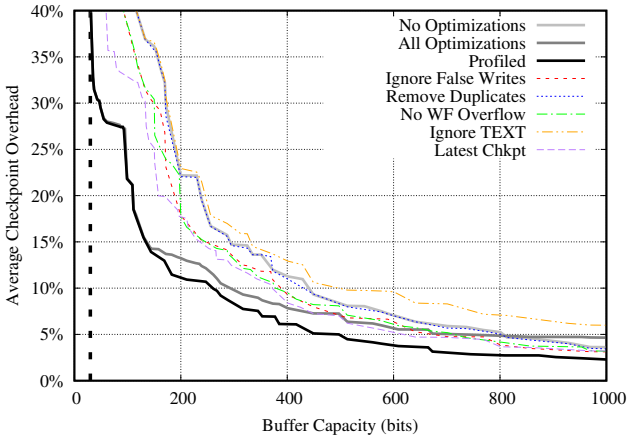

Figure 6: Pareto frontiers of checkpointing overhead vs. buffer size tradeoff space for the Clank policy optimizations discussed in Section 3.2. Except for `All Optimizations` and `Profiled`, at most one optimization is enabled at a time. `Profiled` represents selecting the best performing policy optimization setting of the 32 possible for each benchmark when computing the average. The dashed vertical line represents the hardware requirement of a single Read-first Buffer entry.

Clank policy simulator. The policy simulator executes the memory trace adding checkpoints and re-executions as needed given the buffer composition and size(s) and the power cycle distribution (100*ms* average power-on time for our experiments[4]). The policy simulator also dynamically verifies that it preserves idempotence. The policy simulator outputs a detailed breakdown of checkpointing and re-execution overheads. We explore over 32 policy optimization settings, for 26,000 Clank hardware configurations, for two versions (with and without Program Idempotent memory accesses) of each of the 23 benchmarks (requiring over 8 CPU-months).

Figure 5 shows five Pareto frontiers of average checkpointing overhead for varying levels of complexity of Clank hardware.[5] Looking at the frontiers, the value of the additional Clank buffer types and the compiler is clear. Results show that adding a Write-first Buffer adds value once there are at least two Read-first entries. Interestingly, the results show that spending bits on Write-back Buffer entries is much preferred over spending them on the Write-first Buffer. This makes sense as the Write-back Buffer performs a superset of functions that the Write-first Buffer performs. Results also show that adding the Address Prefix buffer fundamentally changes the performance of Clank by allowing more buffer entries for the same total number of buffer bits. This enables the worst-case average checkpointing overhead to go down from 50% with just 30 bits of buffer space to 40%. Lastly, having the compiler remove Program Idempotent memory accesses, while not a huge improvement, does improve Clank's performance, especially at small buffer sizes.

[4]Note that the key environmental factor that determines run-time overhead is average power-on time. Outside of runt power cycles, Clank's overhead is invariant of the timing of power cycles.

[5]Even though our policy simulator also provides the total and re-execution overheads, we focus on checkpointing overhead as re-execution overhead is controlled through the Performance Watchdog, not buffer size. In general, processor designers should aim to approach zero added checkpoints due to program behavior, instead of having all checkpoints result from the Performance Watchdog. Doing so results in a program-independent minimal total overhead.

7.2 Effects of Clank Policy Optimizations

Section 3.2 details five optimizations to Clank's policy on when to take checkpoints. The design space experiment from the previous section included looking at benchmark performance at all 32 possible Clank policy optimizations settings. In this section, we look at the same data differently to assess the value of each policy optimization. Figure 6 shows the Pareto frontiers for eight policy optimization settings, from all optimizations off, to all on, to only enabling one optimization at a time, to picking the best performing set of optimizations for each benchmark (`profiled`). The results show that the optimal policy depends on the benchmark and Clank buffer size—e.g., `All Optimizations` is sometimes the worst decision. Fortunately, energy harvesting devices use a single, static binary, so it is reasonable to expect profiling to choose the optimal policy setting for a program.

7.3 Hardware Overhead

As mentioned in Section 2.1, hardware overheads are just as important to the total run-time overhead of long-running programs on harvested energy as are software overheads. To measure hardware overheads, we employ our FPGA implementation based on the ARM Cortex-M0+. We construct four Pareto-optimal hardware configurations that have a similar number of total buffer bits: Read-first Buffer (`R`); Read-first and Write-first buffers (`R+W`); the detector buffers plus the Write-back Buffer (`R+W+B`); and all buffer types (`R+W+B+A`). We implement each hardware configuration using Xilinx Vivado [43]. We run Vivado with default settings except for synthesis, where we instruct Vivado to use BlockRAM for all memory (as opposed to LUT ram). This provides overhead numbers comparable to full-custom chip design—the target of Clank. For delay, we set the clock frequency to 50 MHz and validate that all designs still meet timing. To determine power overhead, we first attempted to use Xilinx's Power Analyzer tool with default settings, but all hardware configurations were within the noise of the tool. So, instead, we rely on the area overhead in determining realistic power overhead figures.

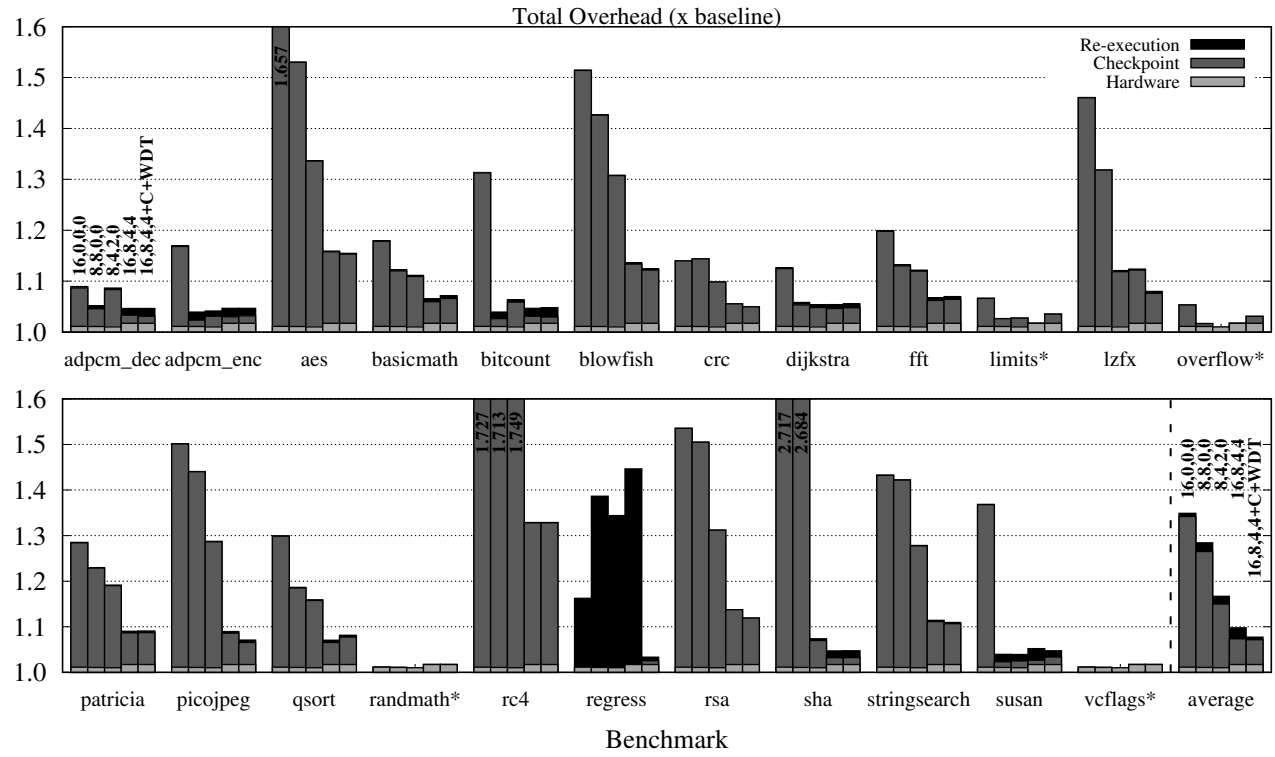

Figure 7: Total run-time overhead (i.e., increased energy usage, plus cycles spent checkpointing, plus cycles wasted on re-executing previously executed instructions) for each benchmark and each configuration of Clank shown in Table 2 at 100*ms* average power-on time. `C+WDT` represents using the compiler and Performance Watchdog timer. Benchmarks marked with an asterisk reliably complete execution within a single power cycle—all other benchmarks fail to execute on intermittent power without Clank.

R, W, WB, AP	LUT	FF	Memory	Avg	SW
16, 0, 0, 0	2.46%	0.74%	0.18%	1.13%	33.75%
8, 8, 0, 0	2.35%	0.74%	0.18%	1.09%	27.32%
8, 4, 2, 0	2.14%	0.70%	0.21%	1.01%	15.66%
16, 8, 4, 4	3.40%	1.52%	0.26%	1.73%	8.03%
16, 8, 4, 4 (+C+WDT)	3.40%	1.52%	0.26%	1.73%	5.98%

Table 2: Hardware overheads and associated software run time overhead (average across all benchmarks) of four Clank buffer compositions. The comma-separated numbers represent the number of Read-first, Write-first, Write-back, and Address Prefix buffer entries, respectively. All buffer compositions are globally Pareto-optimal and have a similar number of total buffer bits.

This is likely pessimistic given Clank's buffers and logic switch less often than the processor's logic.

Table 2 shows the hardware overheads associated with the baseline and four Clank configurations (we include a fifth row is to give context of best achievable performance with the most complex hardware configuration). These results show that Clank performs well (less than 6% software overhead) with a modest amount of extra hardware—less than 2%. Another important consideration is that Clank's buffers are fully-associative. While this is tolerable for the small buffer sizes used by Clank, this will greatly increase hardware cost for buffers with many entries.

Clank has minimal effect on circuit timing because the buffers are accessed in parallel (as shown in Figure 2), with feedback for buffer updates occurring in parallel to exception generation. When considering timing it is important to consider that most energy harvesting microcontrollers are under-clocked compared to what the silicon can support to match memory access timing.

Figure 7 combines both the hardware and software overhead data together to provide the total run-time overhead for the Clank configurations listed in Table 2 for each benchmark. Note that while the Clank buffer configurations selected in Table 2 are globally Pareto optimal, they may not be Pareto optimal for a given benchmark. For example, `bitcount` and `adpcm_decode` have worse performance when adding the Write-back Buffer, while the `average` result shows that adding the Write-back Buffer is a huge boon to performance.

7.4 Checkpoints vs. Re-execution

In most previous work on idempotency, the few cycles between idempotency violations meant that re-execution overhead approached zero. Clank, through pushing idempotency to non-volatile memory and with its Write-back Buffer, opens the door for re-execution cost to dominate the cost of checkpointing. In this experiment, we explore the tradeoff between checkpoint overhead and re-execution overhead with respect to power cycle frequency.

For this experiment we use the same infrastructure and benchmarks as the design space exploration experiment, except we simulate a near infinitely large Clank buffer configuration. The goal of this experiment is to find the point where re-execution time dominates total run-time overhead and find the value of the Performance Watchdog timer that achieves the minimal total overhead. Figure 8 shows the results of this experiment. The results show that the optimal overhead is one that is split evenly between checkpointing and re-execution. Not shown is that the minimum possible run-time

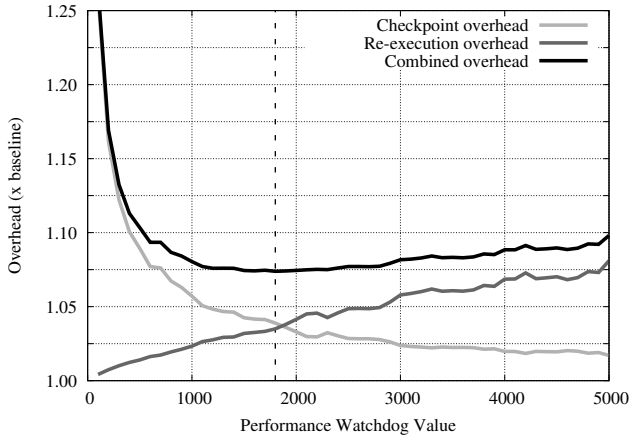

Figure 8: Assuming infinite-sized Clank buffers, this figures shows the general tradeoff between checkpoint overhead and re-execution overhead and how the Performance Watchdog timer imposes a balance of those overheads to minimize total overhead. The dashed vertical line indicates the minimum total run-time overhead. The tradeoff between checkpoint and re-execution overhead holds in-general, regardless of average power-on time.

overhead for Clank, regardless of buffer size, is directly related to the average power-on time of the energy harvesting device. This makes sense as re-execution overhead is dictated by the number of power cycles and the optimal total overhead occurs when checkpointing overhead matches re-execution overhead. Given this, it is possible to calculate the optimal seed value in the ideal case of checkpoints only being added due to the Performance Watchdog (as opposed to program behavior).

7.5 Clank Compared to Previous Approaches

There are several previous approaches that allow programs to tolerate intermittent computation. To put Clank's performance into perspective with earlier work, we find a common benchmark `fft` and report the total run-time overhead of several approaches. Given that re-execution overhead is a critical factor in total run-time overhead, we ensure the average power-on time is the same for each result. Table 3 shows the drastic performance improvement of Clank.

7.6 Clank on Mixed-volatility Systems

The previous section framed Clank with respect to earlier work that also targeted wholly non-volatile systems. Clank also applies to mixed-volatility systems as targeted by DINO [27]. DINO targets systems similar to the FLASH-based systems of today that have large amounts volatile SRAM where commonly accessed memory, like the stack, is located. Non-volatile Flash, due to its speed and power disadvantages, is used to store infrequently accessed values.

To explore Clank's performance on mixed-volatility systems and to compare to DINO, we make several small modifications to Clank to allow it to adapt to large amounts of volatile RAM. First, we restrict it to tracking accesses to the range of memory that is non-volatile. Second, we add a register that tracks how deeply the stack was written to during an idempotent section. This allows Clank to efficiently backup the stack by only writing what has changed to the checkpoint—since the stack is now volatile.

Approach	Total Overhead	Burden
DINO	not ported	programmer
Mementos on FRAM	117%–145%	V measurement
Hibernus	38%	V measurement
Hibernus++	36%	V measurement
Ratchet	32%	compiler
Clank	6%	architecture

Table 3: Comparison of total run-time overhead of several intermittent computation approaches for `fft` with 100*ms* average power-on times.

	Composition	Buffer Bits	Overhead
DINO	mixed	N/A	170%
Clank	mixed	30	3%*
		<100	3%*
		<400	3%*
Clank	wholly NV	30	24%
		<100	5%
		<400	3%*

Table 4: Clank's run-time overhead on the `DS` benchmark used in DINO's evaluation [27] at 100*ms* average power-on time. Memory composition is either mixed (i.e., both non-volatile and volatile) or wholly non-volatile. Results marked with an asterisk are dominated by re-execution overhead.

To compare run-time overheads, we download the `DS` benchmark from DINO's public GIT repository. Table 4 shows the overhead of running `DS` on both a wholly non-volatile version and a mixed-volatility version of Clank at three different buffer sizes (denoted by number of bits). `30` represents having a sole Read-first Buffer entry. The results indicate that Clank actually performs better with some volatility. This result is due to the reduction in checkpoints outweighing the increase in checkpoint size. Experiments with other programs produce similar results.

8 ADDITIONAL RELATED WORK

Clank is related most closely to the areas of checkpointing for energy harvesting/energy-harvesting devices. But, because Clank leverages the notion of idempotence, we cover those related works as well. Finally, we expand our coverage to research on general-purpose recovery techniques.

8.1 Energy Harvesting Devices

As this research has taken place, a new class of devices has emerged in tandem with the shrinking size and power requirements of today's microcontrollers. These new ultra-low power devices are challenging traditional power requirements and seem to be ideal candidates for energy harvesting techniques. Unfortunately, these techniques provide unreliable power availability which can result in computation prematurely being cut off.

Checkpointing techniques have been proposed to better mitigate this unreliable power availability, but often impose overheads in the 100's of percent [37]. Recent work has greatly improved on these overheads by asking the programmer to manually decompose programs into a set of tasks [27] with well-defined interfaces [5] that turn checkpointing into lighter-weight selective data versioning. While this tradeoff of programmer effort for lower overhead works well for small programs operating in predictable and regular environments, it is not a general solution to intermittent computation. Another constraint these techniques have is the low speed and high power draw of Flash-based non-volatile memories. As

emerging non-volatile memories, such as FRAM, have moved closer and closer to the CPU [39], new checkpointing techniques have shown the improvements that can be seen with quicker non-volatile memory [2, 18].

Unfortunately, correctly mixing volatile and non-volatile memory in a checkpointing system is non-trivial and error prone [36]. Early systems like Mementos [37] (and later approaches based on Mementos [2, 18]) were later revealed to be incorrect, leading to semantically impossible states [27].

8.2 Idempotency

We leverage the notion of idempotency when creating restartable execution traces. Mahlke et al. represents the first use of idempotent code sections, targeted at speculative processors [29]. Mahlke et al. denotes these restartable code sections being 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." The paper uses this notion to define how to handle a speculatively executing instruction that throws an exception and shows how to leverage the idempotence property to begin execution from the start of a section of code and still move along the control flow graph correctly.

Kim et al. follows on from Mahlke et al. to show that idempotency can be used to reduce the amount of data stored in speculative storage on speculatively multi-threaded architectures [21]. They note that there are idempotent references that are independent of the memory dependencies that result in errors in non-parallelizable 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 a platform to provide 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 kills correctness, Encore is able to drastically improve performance compared to previous work, while preventing recovery just 3% of the time.

De Kruijf et al. 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 [10]. In this initial work, the authors focus on soft faults that do not mangle the register state and note that registers are usually protected by other means.

As a follow-on, de Kruijf et al. presents algorithms to consume the idempotence information generated by the initial compiler pass [10] and use it to better inform the register allocation step, allowing the new compiler to extend the live range of registers [9]. Extending the live range of register values that are live-in to the idempotent section all the way to the end of the section creates a free checkpoint that enables recovery from side-effect-free faults. In contrast, faults on energy harvesting devices have side effects (e.g., they wipe the registers), complicating recovery.

These earlier works on idempotency focus on static, compiler-implemented analysis. Unfortunately, compiler alias/idempotency analysis passes are limited to analyzing at the function level. This severely limits the number of instructions between checkpoints and results in significant run-time overheads [26, 40]. Much like concurrent efforts for desktop-class systems [33], we expose hardware's ability to cheaply and effectively perform dynamic alias/idempotency analysis. Our results show the power of hardware to fit thousands of instructions between checkpoints; so much that, for the first time, re-execution overhead becomes the limiting factor.

8.3 Other Recovery Techniques

Early research into fault tolerance for computer systems presents the idea of Backward Error Recovery (BER), "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" [34]. Systems have often chosen checkpointing as a technique to enable the recovery points needed for BER [12]. CATCH is an example of a checkpointing system that leveraged the compiler to provide transparent roll-back recovery [25].

The problem with checkpointing is it requires an understanding of what state the process might overwrite in the future [34], often forcing systems to pessimistically checkpoint all state [37]. Research into deterministic replay noticed that by combining concepts of checkpointing and logging, the high overhead of taking a checkpoint could be reduced by logging loads [31]. This reduction in state necessary to enable correct restart improves overhead.

Future work noted that in the presence of reliable memory, undo logging could be used to record stores instead of loads [32]. Work in architecture has revealed that about $\frac{1}{3}$ of instructions tend to be memory operations, $\frac{2}{3}$ of which are loads and the rest stores. By logging stores instead of loads a further reduction in overhead was enabled.

The notion of memory idempotency leveraged by Clank and Ratchet [40] builds upon this trend. Noting that not all of the stores must be included in the undo log reduces the set of information that must be stored. By logging the stores that alias with the loads that occurred since the last checkpoint, during recovery the system can undo the side-effects of the stores and cause the program to begin from conceptually the same state as it did before the interrupted execution.

9 CONCLUSION

We build Clank, a hardware-based system that automatically and dynamically adapts programs to the intermittent model of execution of harvested energy. Clank does this by leveraging the idea of idempotency, i.e., that a sequence of instructions is re-executable as long as the values read during that sequence are preserved. Experimental results show that Clank supports a wide range of programs with no programmer intervention or major hardware modification. Lastly, experiments show that Clank has *total* run-time overheads of as little as 2.5%—near one tenth of the previous best approach, exposing re-execution time as a dominant component of total run-time overhead.

We show that Clank applies to two different memory compositions. The key to this is idempotency—programs implicitly tell the processor the state that they depend on to re-execute correctly. Idempotency extends beyond energy harvesting devices; what is possible by applying our approach to desktop/server-class systems?

REFERENCES

[1] ARM. 2016. DesignStart for Processor IP. (2016). http://www.arm.com/products/processors/designstart-processor-ip/

[2] D. Balsamo, A.S. Weddell, G.V. Merrett, B.M. Al-Hashimi, D. Brunelli, and L. Benini. 2014. Hibernus: Sustaining Computation during Intermittent Supply for Energy-Harvesting Systems. *Embedded Systems Letters, IEEE* 7 (2014), 15–18. Issue 1.

[3] S. C. Bartling, S. Khanna, M. P. Clinton, S. R. Summerfelt, J. A. Rodriguez, and H. P. McAdams. 2013. An 8MHz 75 uA/MHz zero-leakage non-volatile logic-based Cortex-M0 MCU SoC exhibiting 100% digital state retention at VDD=0V with <400ns wakeup and sleep transitions. In *International Solid-State Circuits Conference*. 432–433.

[4] Michael Buettner, Richa Prasad, Alanson Sample, Daniel Yeager, Ben Greenstein, Joshua R. Smith, and David Wetherall. 2008. RFID Sensor Networks with the Intel WISP. In *Conference on Embedded Network Sensor Systems (SenSys)*. 393–394.

[5] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 514–530.

[6] John H. Davies. 2008. *MSP430 Microcontroller Basics*. Newnes, Newton, MA, USA.

[7] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *International Symposium on Computer Architecture (ISCA)*. 497–508.

[8] Marc de Kruijf and Karthikeyan Sankaralingam. 2011. Idempotent Processor Architecture. In *Symposium on Microarchitecture (MICRO)*. 140–151.

[9] M. de Kruijf and K. Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*. 1–12.

[10] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *Conference on Programming Language Design and Implementation (PLDI)*. 475–486.

[11] Samuel DeBruin, Bradford Campbell, and Prabal Dutta. 2013. Monjolo: An Energy-harvesting Energy Meter Architecture. In *Conference on Embedded Networked Sensor Systems (SenSys)*. 18:1–18:14.

[12] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. 1992. Manetho: Transparent Roll Back-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computing* 41, 5 (May 1992), 526–531.

[13] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A. Mahlke, and David I. August. 2011. Encore: Low-cost, Fine-grained Transient Fault Recovery. In *International Symposium on Microarchitecture (MICRO)*. 398–409.

[14] Matthew Hicks. 2016. MiBench port targeted at IoT devices. https://github.com/impedimentToProgress/MiBench2. (2016).

[15] Matthew Hicks. 2016. Thumbulator: Cycle accurate ARMv6-m instruction set simulator. https://github.com/impedimentToProgress/thumbulator. (2016).

[16] Matthew Hicks. 2017. Clank research artifact code repository. https://github.com/impedimentToProgress/ClankRepo. (2017).

[17] Texas Instruments. 2015. MSP432P401R. (March 2015).

[18] H. Jayakumar, A Raha, and V. Raghunathan. 2014. QUICKRECALL: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers. In *Conferences on Embedded Systems and VLSI Design*. 330–335.

[19] J. M. Kahn, R. H. Katz, and K. S. J. Pister. 1999. Next Century Challenges: Mobile Networking for Smart Dust. In *Conference on Mobile Computing and Networking (MobiCom)*. 271–278.

[20] S. Khanna, S.C. Bartling, M. Clinton, S. Summerfelt, J.A. Rodriguez, and H.P. McAdams. 2014. An FRAM-Based Nonvolatile Logic MCU SoC Exhibiting 100% Digital State Retention at VDD= 0 V Achieving Zero Leakage With < 400-ns Wakeup Time for ULP Applications. *Solid-State Circuits, IEEE Journal of* 49, 1 (Jan 2014), 95–106.

[21] Seon Wook Kim, Chong liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. 2006. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Transactions on Programming Languages and Systems* 28, 5 (2006), 942–965.

[22] Daniel Kroening and Mitra Purandare. 2016. EBMC: The Enhanced Bounded Model Checker. (2016). http://www.cprover.org/ebmc/

[23] Silicon Labs. 2016. EFM32 Zero Gecko 32-bit Microcontroller. (2016). http://www.silabs.com/products/mcu/32-bit/efm32-zero-gecko/Pages/efm32-zero-gecko.aspx

[24] Yoonmyung Lee, Gyouho Kim, Suyoung Bang, Yejoong Kim, Inhee Lee, P. Dutta, D. Sylvester, and D. Blaauw. 2012. 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*. 402–404.

[25] C.-C.J. Li and W.K. Fuchs. 1990. CATCH-compiler-assisted techniques for checkpointing. In *Symposium on Fault-Tolerant Computing (FTCS)*. 74–81.

[26] Q. Liu, C. Jung, D. Lee, and D. Tiwari. 2016. Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recovery. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 228–239.

[27] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Conference on Programming Language Design and Implementation (PLDI)*.

[28] Kaisheng Ma, Yang Zheng, Shuangchen Li, K. Swaminathan, Xueqing Li, Yongpan Liu, J. Sampson, Yuan Xie, and V. Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA)*. 526–537.

[29] Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. 1993. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems* 11 (1993), 376–408.

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

[31] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously recording program execution for deterministic replay debugging. In *International Symposium on Computer Architecture (ISCA)*. 284–295.

[32] M. Prvulovic, Zheng Zhang, and J. Torrellas. 2002. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture (ISCA)*. 111–122.

[33] Qingrui Liu and Changhee Jung. 2016. Lightweight Hardware Support for Transparent Consistency-Aware Checkpointing in Intermittent Energy-Harvesting systems. In *Symposium on Non-Volatile Memory Systems and Applications (NVMSA)*, Vol. 5.

[34] B. Randell, P. Lee, and P. C. Treleaven. 1978. Reliability Issues in Computing System Design. *Comput. Surveys* 10, 2 (June 1978), 123–165.

[35] Benjamin Ransford, Shane Clark, Mastooreh Salajegheh, and Kevin Fu. 2008. Getting Things Done on Computational RFIDs with Energy-aware Checkpointing and Voltage-aware Scheduling. In *Conference on Power Aware Computing and Systems (HotPower)*. 5–5.

[36] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Workshop on Memory Systems Performance and Correctness (MSPC)*. Article 5, 3 pages.

[37] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 159–170.

[38] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

[39] Texas Instruments. *MSP430FR59xx Datasheet*. Texas Instruments. http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf

[40] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *Symposium on Operating System Design & Implementation (OSDI)*. 17–32.

[41] Yu-Chi Wu, Pei-Fan Chen, Zhi-Huang Hu, Chao-Hsu Chang, Gwo-Chuan Lee, and Wen-Ching Yu. 2009. A Mobile Health Monitoring System Using RFID Ring-Type Pulse Sensor. In *Conference on Dependable, Autonomic and Secure Computing (DASC)*. 317–322.

[42] Xilinx. 2016. Xilinx Virtex-7 FPGA VC709 Connectivity Kit. (2016). http://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html

[43] Xilinx. 2016. Xilinx Vivado Design Suite. (2016). https://www.xilinx.com/products/design-tools/vivado.html

[44] Hong Zhang, Jeremy Gummeson, Benjamin Ransford, and Kevin Fu. 2011. *Moo: A Batteryless Computational RFID and Sensing Platform*. Technical Report UM-CS-2011-020. Department of Computer Science, University of Massachusetts Amherst, Amherst, MA. http://www.cs.umass.edu/publication/details.php?id=2114

[45] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. 2013. ConAir: Featherweight Concurrency Bug Recovery via Single-threaded Idempotent Execution. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 113–126.