1. A Broken Time Machine

In the six years since publication of the original paper, energy harvesting and intermittent computation have gone from an ivory tower pipe dream to the television remote sitting next to me as I type this article. In the years before I started working on Clank, it was clear from the down-scaling of compute and increased power efficiency that there was an opportunity to remove the dependence on batteries for the smallest computing devices. Batteries scaling has not kept pace with that of compute in terms of size, weight, and cost. Also consider the fact that batteries require maintenance and have a tendency to spontaneously combust, fulfilling the dream of smart dust required finding an alternative energy source.

Researchers realized that energy is everywhere, we are being bathed in energy: sunlight, wireless signals, thermal gradients, and motion are all some examples. This begs the question, “Can we harness this existing energy to perform computation?” Early attempts to answer this question came in the form of energy harvesting devices like the WISP [1], which pulls energy from the wireless spectrum.

Unfortunately, replacing batteries with harvested energy is non-trivial due to the inability to compute continuously with harvested energy. Even with today’s ultra-low-power microcontrollers (which require much less energy than the devices available at the time of Clank’s development), many sources of harvested energy provide only enough energy for short bursts of computation. Thus, harvested energy fundamentally breaks the abstraction of continuously-powered computation that we have all been implicitly trained in.

Given that asking programmers to reason about all possible effects on their software of common case and unpredictable power resets is a non-starter, the first work in the community attempted to provide automatic, software-level mechanisms for stretching computation across frequent power resets [2]. It turns out that reasoning about random and frequent power resets correctly is also error prone for these early energy harvesting support systems [2]–[4]. The crux of the problem is mixing volatile and non-volatile memory in a checkpointing system leads to durability differences that open the door to semantically-impossible program states [5]. This problem was coined as the “broken time machine” problem.

My goal with Clank (and its older sibling, software-only variant Ratchet [6]) was to create the first automatic intermittent computation support systems that maintain program correctness and performance. The driving observation behind Ratchet and Clank is that correct intermittent computation is a replayed-read-after-write-after-read problem: think of a control-flow decision based on variable \( x \), where \( x \) is later updated in the program. Executions where \( x \) is updated, but the program’s checkpointed forward progress is not consistent with that write must be prevented to ensure semantically consistent execution upon replay of the original read.

2. Read-Write-Replay Read Prevention

Clank and Ratchet leverage the notion of idempotency to decompose programs into a series of idempotent sections; Clank dynamically at the core’s memory interface and Ratchet statically at the compiler level. The idea of using idempotency came from a research paper reading group in 2013. I was a new postdoc at the University of Michigan, where I participated in both the computer architecture and computer security reading groups. The reading groups had graduate students present a research paper recently published at a top-tier venue (while everyone else ate free food—my favorite part). At one of these meetings, I heard about the high-level idea of idempotency. From that high-level description, I went to my office and reinvented the wheel for use in the energy harvesting space.

To be fair, Mahlke et al. represents the first use of idempotent code sections in their work on speculative processors [7]. They denote 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.” They use this notion to define how to handle a speculatively executing instruction that throws an exception and show that they can use the idempotency property to begin execution from the start of the section to move along the correct control flow graph.

Kim et al. follow on from Mahlke et al. to show that idempotency can be used to reduce the amount of data stored in speculative storage on speculative multi-threaded architectures [8]. They note that there are idempotent references that are independent of the memory dependencies that result in errors in non-parallizable code.

The big change in idempotency introduced by Clank and Ratchet is the focus on system memory. Specifically, protecting writes to non-volatile memory on microcontroller-based systems. Using idempotency allows for the tracking of only a small subset of writes (which are already less frequent than reads). Thus, idempotency provides for correctness and performance in intermittent computation systems.
The earliest works on idempotency focused on static, compiler-implemented analysis. For example, Clank’s older sibling paper Ratchet is a set of LLVM passes that decomposes a program statically into a series of checkpoint-connected idempotent sections of code [6]. As long as all code is idempotent, then a power failure can occur at any time without exposing replays to inconsistent execution. That is to say that (deterministic) variables will always read as the same value, no matter how many times the replay is re-executed.

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 [6]. Software systems also require software intelligence. My goal with Clank was to show that idempotency analysis is possible in a way that is abstracted away from software and more efficiently than possible with software (due to compiler-level analysis being static and pessimistic versus hardware’s ability to look at a single execution trace).

Clank is essentially hardware support for idempotence analysis. Idempotence analysis requires looking for write-after-read dependencies involving non-volatile memory locations. The base approach is to track whether memory locations are read-first or write-first. When a write occurs to a memory location that is marked as read-first, it signals an idempotency violation, which initiates a checkpoint. This base approach already works better than Ratchet, but I observed that by delaying the writes (i.e., buffering them in volatile memory, like a write back cache does), the checkpointed can also be delayed.

By adding this buffer of idempotency violating writes, Clank is able to avoid checkpoints so well that it creates a new problem. It turns out that eventually there are so few idempotency violations and idempotent sections so long that the run time overhead of replaying a section outpaces the overhead reduction of eliding checkpoints. For one benchmark Clank performed so well that the entire program became an idempotent section—thus becoming antithetical to the goal behind intermittent computation.

A second problem was that of runt power cycles. Runt power cycles are those too short to even make a single checkpoint, preventing software from making forward progress. I solved these problems by inserting artificial idempotency violations to induce a checkpoint. This ensures forward progress while also balancing the run time overhead of checkpointing and replay.

With the theory of operation worked out, I had to go about implementing a Clank prototype. Being a postdoc, implementing a custom ASIC was out of scope; but I needed to get realistic performance numbers. To do so I built the world’s fastest ARMv6-m instruction set simulator. The simulator produced memory traces for each benchmark. To determine the effects of various Clank configurations, I had to simulate the effects on performance of that configuration given the memory trace. For this, I constructed a multithreaded Clank simulator in Java. I ran simulations for several weeks on my one work computer...all the resources that I had at the time.

3. Follow-on work
An interesting tidbit of—at the time surprising—insight that came out of the evaluation was that Clank has the lowest overhead on mixed volatility systems, i.e., Flash-based microcontrollers. Unfortunately, Flash-based microcontrollers dominate deployed systems. Even worse, the frequent checkpoints endemic to continuous checkpointing systems like Clank and Ratchet doom Flash-based devices to an early demise due to their limited write/erase endurance (often as little as 100,000 write/erase cycles).

In follow-on work, I sought to enable intermittent computation on Flash-based microcontrollers. The key insight is that SRAM acts as a non-volatile memory when off-times are short—which is common in many energy harvesting deployments. By keeping checkpoints in SRAM as opposed to writing them to Flash, performance is improved and system lifetime’s are dramatically extended [9]. Follow on work to this addressed the overhead of measuring the energy available in the buffer capacitor [10], the lowest voltage experienced during off periods, and a reconfigurable buffer capacitor fabric.

References

[3] H. Jayakumar, A. Raha, and V. Raghunathan, “Quickrecall: A low computation on Flash-based microcontrollers. The key insight is that SRAM acts as a non-volatile memory when off-times are short—which is common in many energy harvesting deployments. By keeping checkpoints in SRAM as opposed to writing them to Flash, performance is improved and system lifetime’s are dramatically extended [9]. Follow on work to this addressed the overhead of measuring the energy available in the buffer capacitor [10], the lowest voltage experienced during off periods, and a reconfigurable buffer capacitor fabric.

References

[3] H. Jayakumar, A. Raha, and V. Raghunathan, “Quickrecall: A low computation on Flash-based microcontrollers. The key insight is that SRAM acts as a non-volatile memory when off-times are short—which is common in many energy harvesting deployments. By keeping checkpoints in SRAM as opposed to writing them to Flash, performance is improved and system lifetime’s are dramatically extended [9]. Follow on work to this addressed the overhead of measuring the energy available in the buffer capacitor [10], the lowest voltage experienced during off periods, and a reconfigurable buffer capacitor fabric.

References

[3] H. Jayakumar, A. Raha, and V. Raghunathan, “Quickrecall: A low computation on Flash-based microcontrollers. The key insight is that SRAM acts as a non-volatile memory when off-times are short—which is common in many energy harvesting deployments. By keeping checkpoints in SRAM as opposed to writing them to Flash, performance is improved and system lifetime’s are dramatically extended [9]. Follow on work to this addressed the overhead of measuring the energy available in the buffer capacitor [10], the lowest voltage experienced during off periods, and a reconfigurable buffer capacitor fabric.

References

[3] H. Jayakumar, A. Raha, and V. Raghunathan, “Quickrecall: A low computation on Flash-based microcontrollers. The key insight is that SRAM acts as a non-volatile memory when off-times are short—which is common in many energy harvesting deployments. By keeping checkpoints in SRAM as opposed to writing them to Flash, performance is improved and system lifetime’s are dramatically extended [9]. Follow on work to this addressed the overhead of measuring the energy available in the buffer capacitor [10], the lowest voltage experienced during off periods, and a reconfigurable buffer capacitor fabric.

References

[3] H. Jayakumar, A. Raha, and V. Raghunathan, “Quickrecall: A low computation on Flash-based microcontrollers. The key insight is that SRAM acts as a non-volatile memory when off-times are short—which is common in many energy harvesting deployments. By keeping checkpoints in SRAM as opposed to writing them to Flash, performance is improved and system lifetime’s are dramatically extended [9]. Follow on work to this addressed the overhead of measuring the energy available in the buffer capacitor [10], the lowest voltage experienced during off periods, and a reconfigurable buffer capacitor fabric.

References

[3] H. Jayakumar, A. Raha, and V. Raghunathan, “Quickrecall: A low computation on Flash-based microcontrollers. The key insight is that SRAM acts as a non-volatile memory when off-times are short—which is common in many energy harvesting deployments. By keeping checkpoints in SRAM as opposed to writing them to Flash, performance is improved and system lifetime’s are dramatically extended [9]. Follow on work to this addressed the overhead of measuring the energy available in the buffer capacitor [10], the lowest voltage experienced during off periods, and a reconfigurable buffer capacitor fabric.

References

[2] B. Ransford, J. Sorber, and K. Fu, “Memen...