Competition: Fast Intermittent Computing via Mixed Memory Model

Saad Ahmed
Georgia Institute of Technology
sahmed@gatech.edu

Josiah Hester
Georgia Institute of Technology
josiah@gatech.edu

Abstract

Intermittently-powered embedded devices ensure forward progress of programs through state checkpointing in non-volatile memory (NVM). However, NVM writes are energy-expensive and add to the execution time of the application. We developed system support to maximize computational progress by minimizing the need to perform NVM writes in two ways. First, we focus on optimizing the reference implementation to allow faster execution which significantly reduces the application state to be checkpointed. Second, we employ a mixed memory model that reduces the update size at each checkpoint. These design decisions enable us to compute 3.5×-4.4× more challenges than a double buffering-based approach commonly used in existing literature.

CCS Concepts

• **Computer systems organization → Embedded software**.

Keywords

Intermittent Computing, Energy Harvesting, Batteryless Internet of Things, HashCash

1 Introduction

Miniaturized mechanical systems have enabled the design of tiny energy harvesters that help embedded devices eliminate their dependency on traditional batteries [2]. Harvested energy, however, is generally highly variable across space and time [2]; making periods of normal computation and periods of energy harvesting to be unpredictably interleaved. Checkpointing application state over non-volatile memory (NVM) is required to let the program cross periods of energy unavailability [1, 5, 7]. Checkpoints often represent a dominating factor in an application's energy profile and subtract from the energy budget that can be used for computations. Therefore, taming this overhead is crucial in order to maximize computational progress in a single energy cycle.

Challenges: Checkpointing overhead is defined by two factors: Volatile application state and frequency of checkpoints.

- **Volatile Application State** Larger the size of the volatile application state, the more the checkpointing energy, and the lesser the energy available for performing program execution. Therefore, system support is required to track changes in the application state from the previous checkpoints [1] in order to reduce the size of update.
- **Checkpointing Frequency** Applications running on battery-free devices must be instrumented to add system support that enables resumption of application execution across each power failure.

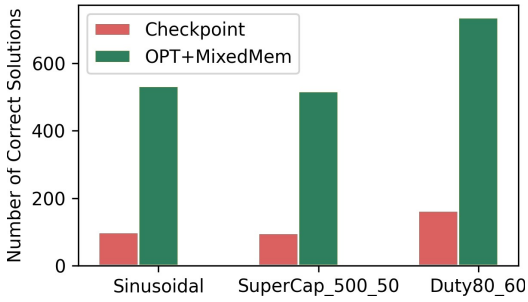

Figure 1: Figure shows the checkpointing overhead compared to the mixed memory model when running the optimized version of the application.

Figure 1 shows the overhead of a double-buffered approach (commonly used by checkpointing [1] and task-based systems [7] for state retention) compared to direct NVM writes. Double-buffering forces the system to write 2× the size of volatile state, which significantly increases the overhead [4]. The ideal configuration is to have a differential checkpointing mechanism [1] coupled with hardware support to trigger checkpointing when the voltage falls below a software-defined threshold [3]. However, we don't have any hardware support for this competition, so we focus only on software-based solutions.

Solution: We solve the above-mentioned challenges in two ways to maximize computational progress in each energy cycle. First, we optimize the reference implementation to employ msp430's intrinsic functions and inline assembly instructions that consume a lesser number of clock cycles compared to the ones generated by msp430gcc. Second, we employ a mixed-memory model to reduce the update size at each checkpoint. It allows the system to simply save the data onto NVM instead of tracking the changes, as required by some of the existing works [1, 6] thus saving energy.

Optimizing the application significantly reduces the need to perform a checkpoint, as the application is able to finish multiple iterations for most energy traces.

2 System Design

Our system profiled the entire application, especially the hot code paths and functions as most of time is spent executing the instructions within that path/function. We discuss each step in detail.

2.1 Optimizing SHA-1

Bit rotations, swapping endianess, and string concatenation functions form the core of SHA-1 algorithm. Therefore, we first targeted functions performing these operations to improve their implementation.

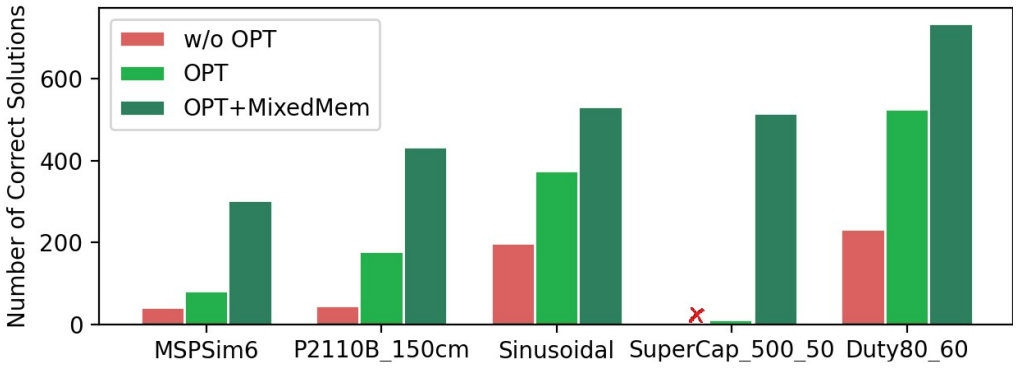

Figure 2: Figure shows the number of correct solutions provided by the application after implementing the optimizations and mixed memory model for five energy traces, when running for a duration of 60 seconds.

- **Bit Rotations:** As MSP430 is 16-bit MCU, rotating a long integer (32-bits) requires data movement that causes significant degradation in the application execution. Therefore, we used inline assembly code for performing bit rotation of long integers by 1, 5, and 30.
- **Swapping Endianess:** Reversing the endianness of a 32-bit value not only involves swapping the lower 16 bits with the upper 16 bits but also requires swapping each byte in each half. For this purpose, we employed MSP430's intrinsic `SWAP_BYTES` function which is a macro for `SWPB` instruction.
- **String Concatenation:** Reference implementation used `snprintf` function, which concatenates all strings, even the ones that did not change from the previous iteration, thus consuming a lot of clock cycles. We replaced it with a custom helper function that only concatenates the counter with the unchanged stamp string, thereby improving performance.

Additionally, we performed loop unrolling for `sha_transform` function and avoided `memcpy` as much as possible.

2.2 Mixed Memory Model

MSP430FR5994 is equipped with a mixed memory model that allows non-volatile (FRAM) and volatile memory (SRAM) to be accessed as main memory. Using this feature, we designed the system in a way that maximizes allocation of `.data` as well as `.text` sections onto SRAM so that accesses remain faster and energy-efficient [4]. For this purpose, we profiled application's execution to find the best configuration for both data and code placement. We allocated hot code paths in SRAM as it is faster to execute and placing such functions in FRAM incurs access latency, especially beyond 8MHz as it requires a delay cycle. Faster execution significantly reduces the volatile application state to be retained across reboots which helps improve performance.

2.3 Results

Figure 2 shows the number of correct solutions for the reference code, i.e., without optimizations (`w/o OPT`) and after implementing optimizations (`OPT`) and energy-aware mapping of code and data (`MixedMem`). The application ran for a duration of 60 seconds on each of the traces. We can clearly see that the number of correct solutions provided by the application is at least 2.7× more than that of the reference code on all of the traces. We can also observe that the reference code could not finish one iteration for `SuperCap_500_50`. However, after optimizations, it was able to finish one iteration, and with state-retention, it was able to provide 517 correct solutions.

Figure 1 shows that, with all optimizations and mixed memory, our design decisions enable us to compute 3.5×-4.4× more challenges than a double buffering-based approach commonly used in the existing literature [1, 7].

2.4 Conclusion

We designed system support to enable speedup computations to provide a hashcash solution for different difficulty levels. We showed that we can compute at least 2.7× more number of solutions compared to the reference implementation. This improvement stems from system's ability to perform faster execution and avoid application state tracking by employing a mixed memory model.

References

[1] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 70–81.

[2] Saad Ahmed, Bashima Islam, Kasim Sinan Yildirim, Marco Zimmerling, Przemysław Pawełczak, Muhammad Hamad Alizai, Brandon Lucia, Luca Mottola, Jacob Sorber, and Josiah Hester. 2024. The Internet of Batteryless Things. *Commun. ACM* 67, 3 (2024), 64–73.

[3] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2014. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* 7, 1 (2014), 15–18.

[4] Hrishikesh Jayakumar, Arnab Raha, Jacob R Stevens, and Vijay Raghunathan. 2017. Energy-aware memory mapping for hybrid FRAM-SRAM MCUs in intermittently-powered IoT devices. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 3 (2017), 1–23.

[5] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[6] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. ASPLOS* (March 5–11). ACM, Newport Beach, CA, USA.

[7] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. 41–53.