Competition: Hashcash on the MSP430 – Intermittent Computing and Code Optimization

Andreas Könsgen
Jens Dede
Saurabh Band
Shadi Attarha
Anna Förster
{ajk|jd|saurabh|sattarha|afoerster}@comnets.uni-bremen.de
Sustainable Communication Networks
University of Bremen, Germany

Abstract

This work presents our optimizations for a hashing implementation running an MSP430 microcontroller that operates in minimal and interrupted power conditions. Depending on the energy source, we can significantly increase the performance and, thus, the number of correct solutions. Most performance enhancement results from storing the program state in non-volatile FRAM. Optimizing the code yields further acceleration, whereas tuning the compiler toolchain only gives a minor performance contribution.

CCS Concepts

• **Hardware → Chip-level power issues**; • **Security and privacy → Hash functions and message authentication codes**.

Keywords

Hashcash, MSP430, Intermittent Computing, Code Optimization, Software Development Toolchains

1 Introduction

The growing interest in sustainable computing has led to a rise in the development of energy-harvesting devices, which operate using intermittent power sources such as solar, thermal, or radio frequency energy. These devices, while eco-friendly, pose significant challenges due to their unpredictable power availability. The EWSN 2024 Sustainability Competition aims to benchmark the performance of such intermittently powered devices, specifically focusing on their ability to handle computationally intensive tasks under intermittent power conditions. In the challenge, those tasks are performed by a Texas Instruments MSP430FR5994 microcontroller, using HashCash – a proof-of-work hashing algorithm – on a remotely-controlled testbed with intermittent power. The objective is to find valid hashes for challenges provided on an external FRAM as follows:

(1) **Read the challenge**: Retrieve a challenge from the external FRAM. This challenge serves as the input for the hash computation process and also provides a given difficulty.
(2) **Calculate a valid SHA1 hash**: Compute a SHA1 hash based on the challenge. The goal is to find a hash that meets the specified difficulty level, defined by the number of leading zeros in the hash. This step requires multiple iterations, as each attempt involves slightly updating the input and recalculating the hash until a valid result is obtained.
(3) **Store the valid result**: Once a SHA1 hash meeting the required difficulty is found, store the result in the external FRAM memory for later retrieval and validation.

This process is repeated till all given challenges are solved or the maximum execution time is exceeded. Afterwards, the results are evaluated and rated. The more correct hashes are calculated, the better.

Starting from a provided reference implementation, this work describes how we optimize the calculation of hashes using different approaches.

2 Our Approach

Ensuring the proper operation in spite of power failures can be achieved using several methods. We identified three main points: Firstly, one has to make sure the results are properly stored during power failure and recovered later. Secondly, optimized computing functions for the given challenge can give an essential gain and thirdly, also the compiler can be tuned by optimizing the parameters. Our investigations reveal that the first step significantly impacts the outcome under intermittent power. The second step shows a twofold acceleration with constant power, while the third step has a minor effect. The following subsections detail our approach.

2.1 Data Storage

Our approach of handling the challenge differs from our initial expectations. Initially, we believed the main task would involve implementing checkpoints to save the execution state in case of power failure, focusing on optimizing their placement and frequency due to the assumed slow speed of flash ROM compared to RAM. However, the built-in FRAM of the MSP430FR5994 microcontroller operates at 8 MHz and is nearly as fast as SRAM, as confirmed by our tests and also by the user manual. This eliminated the need for checkpoint optimization; it is sufficient to place persistent variables into the internal FRAM instead of using SRAM.

Since this FRAM operates at a lower voltage than the microcontroller's processing unit, a slow voltage drop will always cause the processor to fail before the FRAM. Thus, there is no need for read-after-write validation to ensure data integrity in the FRAM.

Selecting the right variables for storage in the internal FRAM is critical. We avoided placing variables within the SHA functions due to their short runtime and the large number of variables. Instead, we stored the challenge counter, solution counter, and the address for the next correct solution as persistent variables in the internal FRAM.

To handle potential inconsistencies if power fails during variable updates, we introduced a status variable to log which variables were successfully updated. This status variable is also stored in FRAM, enabling proper reinitialization of the program after a power failure.

2.2 Code Optimization

To maximize overall speed, we focused not only on making the code resilient to power outages but also on optimizing hash computations.

Since the input messages for the SHA calculations are small, under 56 bytes, we removed code handling large messages, specifically `sha_init()` and `sha_update()`. The messages for the SHA computations vary only by the solution counter, placed at the end of the string. To avoid repeatedly generating the entire message string inside the `generate()` function's loop, we moved the string preparation outside the loop. This approach requires either recomputing the string or storing it in FRAM and recover it after a power failure.

Decimal formatting of the solution counter, which is also part of the SHA input string, was replaced with the hexadecimal format, avoiding time-consuming radix conversion. We also removed random number generation for each challenge string to reduce overhead.

For `sha_transform()`, instead of shifting the values between variables, we implemented variable rotation directly in each of the 80 SHA rounds. However, this approach resulted in slower execution, likely due to the C compiler's optimizer.

We enabled the `UNROLL_LOOP` flag in `sha_transform()` to replace the loop which implements the compression function with linear code.

When expanding the chunk passed to `sha_transform()` from 16 to 80 words, it might seem beneficial to do so in parallel to the compression function as suggested by [1]. However, this requires additional CPU registers, which we avoided since the compression function itself already allocates a large amount of registers.

The `sha_transform()` function requires bitwise rotation at different occasions. Since there is no such operator in the C language, the rotation is implemented with two bit shifts and a bitwise `OR` operation. We implemented assembly code using native CPU instructions for a direct execution of the rotation, but this also led to slower execution. Complete implementation of the SHA function in assembly code may provide more consistent performance by giving control over machine instructions, circumventing unpredictable compiler optimizations.

In future work, running code from SRAM instead of FRAM could significantly accelerate execution, as SRAM eliminates the wait states required for FRAM access. The FRAM is clocked at 8 MHz whereas the SRAM runs at the CPU speed of 16 MHz. Small code fragments which run locally, e.g. loops, can benefit from the microcontroller's the FRAM cache which however has a size of only 32 bytes. For larger code, the program can copy a function to SRAM, however given SRAM's limited size of 4 kilobytes, only critical parts, like the compression function, should be placed there, with provisions for reloading after power failures.

Furthermore, parts of the hash function could be precomputed since most of the input message remains constant between solution attempts. Inside the search for a solution of a challenge, it is only the solution counter which changes between two consecutive input messages for the SHA function, and this counter is placed at the end of the input string. Additionally, preloading the challenge strings from external to internal FRAM could allow immediate processing after power failures.

Assembly implementation is necessary for these optimizations and also by itself yields further acceleration. However, the compexity of working with assembly language requires local debugging, which was not possible due to the lack of an available MSP430FR5994. Hence, the current C implementation in our case represents the best achievable solution under these constraints.

2.3 Compiler Optimization

The used GNU C compiler offers also some potential optimizations. One are the compiler flags. The flags are usually set for minimum size and best compatibility. Here, we optimize those for speed and the given code.

We achieved enhancements by replacing the compiler flag for size optimization `-Os`, which is Energia's default setting, by `-O2` (speed). This enables some additional optimizations which are otherwise left out because they require additional memory footprint. Since the FRAM size of the MSP430 is, however, sufficiently large, this does not result in problems. Setting this flag to `-O3` or `-Ofast` decreases the performance, which shows the sometimes unpredictable behaviour of the compiler which we already discussed in the context of code optimization.

Also, we considered that the version of the MSP430 GNU C compiler used in the Energia IDE (4.6.3) is outdated, since Energia is no longer maintained since some years. Therefore, we evaluated the most recent compiler provided by Texas Instruments (9.3.1.11 and 8.3.1.25) and adapted the code accordingly. However, using the updated toolchain did not yield any enhancement, in contrary the execution speed was slightly slowed down. Therefore, we decided to stick to the version 4.6.3.

3 Conclusion

In this paper, we outlined our strategies for implementing and optimizing HashCash on the MSP430FR5994 in environments with intermittent power, highlighting the capabilities of energy-harvesting devices for handling computationally intensive tasks. Through various techniques, including storage optimization, code refinement, and strategic compiler flag selection, we achieved performance improvements of up to the factor of two. However, further optimizations would necessitate direct access to the hardware, which was not feasible in our case.

References

[1] C. Franck and J. Großschädl: Optimized Implementation of SHA-512 for 16-bit MSP430 Microcontrollers. In Proc. International Conference on Information Technology and Communications Security, pp. 86-99. Springer, 2021.