Competition: Leveraging Hibernus to Manage Intermittent Power Supply During Hashcash Computation

Firdaus Ritom
mohd-firdaus.hirman-ritom@newcastle.ac.uk
Newcastle University
United Kingdom

Sergey Mileiko
serhii.mileiko@newcastle.ac.uk
Newcastle University
United Kingdom

Domenico Balsamo
domenico.balsamo@newcastle.ac.uk
Newcastle University
United Kingdom

Abstract

This is the abstract of Team 13's solution for the EWSN Sustainability Competition. Our solution utilizes Hibernus, an approach that detects imminent power losses and saves a snapshot of the system's state to non-volatile memory (FRAM) just before the power is lost. When power is restored, the system resumes from the saved state, ensuring seamless continuation of computations. In addition, we optimized the execution of the Hashcash algorithm by adjusting compiler settings and employing XOR shifts for more efficient random searching of the correct Hashcash solution. These optimizations slightly accelerated the process of searching for the correct Hashcash solution. The overall results showed that while Hibernus introduced some computational overhead due to internal voltage monitoring, it successfully saved and resumed Hashcash operations under a sinusoidal power profile.

CCS Concepts

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

Keywords

Energy harvesting, Intermittent computing, Embedded software

1 Introduction

There is increasing interest in promoting sustainability in the Internet of Things (IoT) by replacing battery-powered devices with low-power intermittent systems. Paper [5] introduces E-Cube, a remote benchmarking facility designed to evaluate the computational performance of battery-less intermittent devices.

E-Cube serves as the foundation for the sustainability competition at the EWSN 2024 conference, which aims to stress-test solutions using this facility [4]. Each team is tasked with developing software capable of performing Hashcash computations while managing various power profiles to simulate intermittent power supply conditions.

Hashcash is a proof-of-work algorithm designed to impose computational costs as a deterrent to email spamming [1]. The system must find a Hashcash solution such that the hashed output has a specific number of leading zero bits, based on the assigned difficulty level. In the competition, the format for the Hashcash solution or stamp is provided below:

$$ver : bits : date : resource : ext : rand : counter \quad (1)$$

where *ver* is always "1", *bits* is difficulty level (1 byte), *date* is fixed "24015" string, *resource* is a challenge string (15 bytes), *ext* is empty and *counter* is user generated number [4].

In addition to implementing the Hashcash algorithm, the competition requires teams to read from and write to external FRAM via I2C. The "*bits*" and "*resource*" data are read from the external FRAM at address 0x50, while the solved Hashcash solution must be written to the external FRAM at address 0x51. The main objective of the competition is to solve as many challenge strings as possible, provided by the external FRAM, within a fixed duration and under varying power profiles.

Due to intermittent nature, the system must save its Hashcash computation progress and resume when power returns. For our solution, we implemented Hibernus, an approach developed by Dr. Domenico Balsamo, one of our team members, as introduced in [2]. Hibernus saves the system's state to non-volatile memory (NVM) just before power loss, entering a low-power mode. When power is restored, Hibernus restores the system state and resumes computation seamlessly. Since Hibernus saves and restores system-level registers, it is application-agnostic and can be used across different programs. This approach is further discussed in Section 2.1.

We implemented optimizations to enhance the performance of Hashcash computation. These include migrating from the Energia IDE to Code Composer Studio (CCS), adjusting the compiler optimization level, and changing the "counter" values randomly instead of incrementing sequentially. These optimizations were made to increase the number of Hashcash solutions found. Detailed explanations of these methods can be found in Section 2.2.

2 System Design

The test platform was set up using an MSP430FR5994 microcontroller (MCU) connected to a power supply via a Schottky diode [4]. The competition provided an initial version of the Hashcash application, compiled for the Energia IDE. Section 2.1 covers the implementation of Hibernus, while Section 2.2 outlines the optimization strategies we applied.

2.1 Hibernus Implementation

Hibernus requires specific voltage threshold levels, V_S and V_R, to indicate when to trigger the saving and restoring of the system state. This necessitates constant monitoring of the input voltage, Vcc. However, during this competition, we were unable to create an external voltage divider connected to the Analog-to-Digital Converter (ADC) input pin, as described in paper [2]. Therefore, we needed an internal method to monitor Vcc.

The MSP430FR5994 features a built-in, configurable 12-bit ADC module (ADC12_B). To monitor Vcc internally, we configured Vcc as the reference voltage and set an internal reference voltage of $1.2V$ at the ADC input channel. This configuration allows for accurate

Vcc measurements between $1.8V$ and $2.2V$, enabling lower V_s and V_R values for a broader operating range during computations. We enabled the ADC12_B interrupts ADC12HIIE and ADC12LOIE to trigger saving and restoring, respectively. ADC measurements are taken every $100ms$ using the built-in timer module, Timer_A1.

To save the system state, since the operating voltage of the MCU is $1.8V$, we set the ADC12LOIE trigger to $1.9V$. The internal FRAM is utilized as non-volatile memory (NVM) for saving the system state. We save core registers (R1-R15), the stack pointer (SP), general-purpose registers (GP), and RAM into FRAM, creating a snapshot. Memory blocks of 4 bytes are compared with the previous snapshot to identify any differences, reducing the number of writes during the saving process. After saving, the MCU enters low power mode (LPM4).

To restore the system state, we configured ADC12HIIE to trigger at $2.1V$. When this voltage is reached, the MCU exits LPM4 and begins reading from the internal FRAM to restore the system state from the saved snapshot. Since the SP, core registers, and RAM have been restored and the GP registers have been reconfigured, the MCU can resume executing the previous cycle's instructions. This makes the Hibernus method application-agnostic, as no changes to the application are necessary.

2.2 Hashcash optimization

As mentioned in Sec. 1, several optimizations were made to increase Hashcash solutions. Due to Energia IDE's limited configuration capabilities on the MSP430FR5994 platform, we ported the code to CCS, which offers more control over memory and compiler settings. The CCS linker command file allowed us to define a fixed FRAM memory region for system state saving. Additionally, CCS compiler settings enable us to balance size and speed so we set the speed optimization level to maximum $'5'$ as application size is small.

During Hashcash computation, the stamp changes each time a solution is not found. In initial Hashcash application, the "$counter$" portion of the stamp was incremented by 1 until a solution was discovered. To hasten the search, we modified the approach to randomly change the "$counter$" using the XOR shift method, as introduced in [3]. This method generates a random number quickly by performing three shifts: first, it left shifts the number by 13 bits, then right shifts by 17 bits, and finally right shifts by 5 bits. This combination was determined to be optimal for producing randomness [3]. Utilizing the XOR shift method helps to evenly distribute the search for the correct "$counter$" value increasing the probability of quickly finding the Hashcash solution.

3 Current Results

This section discusses the measurements taken during development and compares the initial and latest code versions. Measuring the saving time of Hibernus was crucial, as the system needs to save its state before the power supply drops below operational levels. It was found that saving the states to FRAM took approximately $7ms$, while it took about $11.45ms$ for the voltage to drop from $3V$ to $1.8V$ when the supply was abruptly turned off. The saving time for Hibernus is adequate to handle square waveform power profiles.

Table 1 shows the comparison between the initial code and the latest version regarding the number of Hashcash solutions solved under various power profiles and testing durations. With a constant power supply of $3V$ for 120 seconds, the latest code produced a lower number of Hashcash solutions. This decrease is attributed to the additional initialization processes for the ADC and Timer modules, as well as the interrupts triggered every $100ms$ for V_{cc} monitoring. Additionally, using the XOR shift method instead of incrementing the "$counter$" by 1 resulted in an increase in Hashcash solutions from 627 to 633 during tests with a constant power supply over 120 seconds.

Table 1: Hashcash performance for each power profiles

Power Profile	Duration (s)	No. of Hashcash Solved	
		Initial code	Latest code
Constant	120	734	633
Sinusoidal	120	199	347
Duty50_60	120	442	341
Sinusoidal	300	199	538
Duty50_60	300	442	341

The sinusoidal power profile demonstrated the system's capability to handle intermittency. In the initial code, the system yielded the same number of solutions for 120 seconds and 300 seconds durations. However, the Hibernus yielded more Hashcash solutions when increasing from 120 seconds to 300 seconds under the sinusoidal power profile confirming that Hibernus could successfully save and restore the system state.

Hibernus unable to handle Duty50_60 power profile (50% duty cycle square waveform), as there was no increase in Hashcash solutions when duration was extended. This issue stemmed from the slow sampling time of the ADC module, which monitors Vcc every $100ms$. MCU was unable to quickly save the system state, especially since the voltage drops from $3V$ to $1.8V$ in just $11.45ms$.

4 Conclusions

The current implementation of Hibernus is capable of handing intermittent power supply. However, it has limitation for worst case scenario such as abrupt power loss. Further enhancements can be made by changing the V_S and V_R as well as the sampling rate to monitor V_{cc}.

5 Acknowledgment

This work is supported by the UK EPSRC NIA EP/W022877/1 and UK EPSRC IAA EP/X525601/1, the British Academy's Researchers at Risk Fellowship.

References

[1] Adam Back. 2003. Hashcash. http://www.hashcash.org/.
[2] Domenico Balsamo et al. 2015. Hibernus: sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7, 1, 15–18. DOI: 10.1109/LES.2014.2371494.
[3] George Marsaglia. 2003. Xorshift rngs. *Journal of Statistical Software*, 8, 14, 1–6. DOI: 10.18637/jss.v008.i14.
[4] Markus Schuß. 2024. Ewsn'24 sustainability competition: competition format & how to use the e-cube testbed. https://iti-ecube.tugraz.at/wiki/images/f/f5/Ecube_tutorial.pdf.
[5] Markus Schuß and Carlo Alberto Boano. 2024. E-cube: towards a first benchmarking facility for battery-free systems. In *Proceedings of the 2024 International Conference on Information Technology for Social Good*. Association for Computing Machinery, New York, NY, USA, 399–403. DOI: 10.1145/3677525.3678688.