Competition: ShiSHA-1: Ligthning-Fast SHA-1 Implementation for Energy-Constrained Devices

Antonio Escobar-Molero
antonio.escobar@infineon.com
Infineon Technologies AG
Neubiberg, Germany
RedNodeLabs UG
Munich, Germany

Alberto Martín Martín
martmartalb@correo.ugr.es
Universidad de Granada
Granada, Spain
eesy-innovation GmbH
Unterhaching, Germany

Juan Cruz-Cozar
juan.cruzcozar@infineon.com
Infineon Technologies AG
Neubiberg, Germany

Jirka Klaue
jirka.klaue@airbus.com
Airbus SE
Hamburg, Germany

ABSTRACT

Intermittent computing requires the design of algorithms able to work with unpredictable power availability. The goal of ShiSHA-1 is to compute SHA-1 hashes as fast as possible in the MSP430, saving non-volatile checkpoints until finding the solution to a predefined set of Hashcash challenges. Our techniques to reduce the computation time of the algorithm also decrease its memory footprint and power consumption. Optimizations are done at the algorithm-, assembly- and system- levels to achieve a SHA-1 block execution of 155 µs in the 16-bit MSP430FR5994 operating at 24 MHz.

CCS CONCEPTS

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

KEYWORDS

Intermittent Computing, Energy Harvesting, Efficient Firmware Development, Embedded Benchmarking

1 INTRODUCTION

Intermittent computing techniques allow low-power devices to operate without batteries; typically using energy harvesters in environments with fluctuating power availability. The goal is to perform as much work as possible when energy is available, efficiently storing checkpoints in non-volatile memory to retain the progress when the power is cut off. As a toy benchmark, it is proposed to compete on solving Hashcash [1] proof-of-work challenges, given unpredictable time and energy budgets and using the 16-bit MSP430FR5994 MCU powered by a voltage source following an arbitrary waveform. Since there is no storage element and the voltage is independent of the current consumption, the optimal strategy is to perform as much work as possible when the voltage is above the minimum turn-on threshold of the MCU, e.g. by overclocking and optimizing the code for speed. The main goal of ShiSHA-1 is to minimize the execution time of each individual round of the SHA-1 algorithm to maximize the chances of solving the challenges within the available computing time. Presented techniques can be used as inspiration on how to maximize the energy efficiency in constrained devices when performing complex computations.

2 SPEEDING-UP SHA-1 FOR THE MSP430

Hashcash proofs are based on the SHA-1 cryptographic hash function. SHA-1 works by breaking the input into 512-bit blocks. The Hashcash string is normally short enough to fit into a single block. It then undergoes a series of operations, including bitwise logical functions, modular additions, and rotations, across 80 rounds of computation. SHA-1 is designed to create a unique 160-bit hash for a given input, making it computationally infeasible to reverse-engineer or find two different inputs that produce the same hash. MSP430 MCUs are ultra-low power devices, with limited memory and computing capabilities. Furthermore, working with a 16-bit architecture adds particular challenges when dealing with the 32-bit operations used in SHA-1. Multiple optimizations at different levels are required to achieve fast and energy-efficient operation.

2.1 Algorithm-Level Optimizations

The word length of the SHA-1 algorithm is 32-bit. It requires a 80-word buffer for storing the word sequence, labeled W[0], W[1],..., W[79]; and a 5-word buffer to save the state words, labeled A,B,C,D,E. The algorithm requires precomputing the whole words sequence and then running an 80-round scheme to sequentially update the state words. Alternatively, the computation of the word sequence and state functions can be interleaved, saving memory since only a 16-word circular array is required [2]. As a design choice, we sacrifice memory by not interleaving the computation, obtaining first the complete word sequence before executing the 80-round state functions. This is done in order to minimize the execution time for this particular architecture, in which an interleaved approach would complicate an efficient register allocation.

While it is impractical to speed-up independent hash calculations of non-related input messages, the fact that successive Hashcash attempts share most of the characters enables using some techniques to accelerate the computation of the algorithm. A number of intermediate results can be precomputed in the first round and reused in successive rounds, until the solution is found. Similar ideas have been previously exploited when cracking password-based key derivation functions based on SHA-1 [4]. By keeping the variable part of the solution in W[7] we can use the following properties: (1) W[0] to W[6] are constant, (2) W[9] to W[14] are

zero. We also use a fixed number of alphanumeric characters in the user random as the variable part, further allowing to keep the padding and length of the string constant, so (3) W[8] and W[15] are also constant. Given (1), (2) and (3), the following optimizations apply:

- `OPT-1`: The number of required XORs to compute the 80-word sequence is reduced from 192 to 147.
- `OPT-2`: The number of rounds for the update of the state words is reduced from 80 to 73. Furthermore, rounds 9 to 14 are faster since their respective word is zero and does not need to be loaded from memory and added.

2.2 Assembly-Level Optimizations

While high-level programming languages boost productivity by offering abstractions and easier syntax, they often sacrifice efficiency, especially in resource-constrained environments. This overhead can be detrimental to performance and energy usage. We have developed the time-critical SHA-1 algorithm completely in MSP430 assembly, obtaining dramatic gains. It showcases how low-level programming is still relevant and compilers are not to be blindly trusted, even at the highest optimization levels. Many small optimizations have been introduced, like minimizing memory load and store operations or trying to avoid instructions with expensive addressing modes. Additionally, we exploit as much as possible the limited instruction set. As an example, in each round left rotations of 5- and 30-bit are required. In MSP430, the most straightforward implementation is to perform the rotations bit by bit using three assembly instructions per bit (RLA, RLC, ADC). To optimise this, the 30-bit left rotation is replaced with an equivalent 2-bit right rotation, while the 5-bit left rotation is done with the 32-bit hardware multiplier, saving a few cycles. In particular, we can highlight some optimizations related to the most time-consuming section of the algorithm, the 80-round update of the state variables:

- `OPT-3`: The strategy during the 80 rounds is to always keep the state variables (A, B, C, D, E) in the CPU registers, requiring two 16-bit registers for each (R4-R13). R14 and R15 are used for temporal calculations, which gives just enough registers to minimize memory accesses and maximize operations involving only registers (with only 1 clock cycle cost). Memory accesses in round i are only needed to load the respective W[i] and the constant required for that round. As a trick, the stack pointer register, SP, is also used as a general-purpose register, since an additional register for storing memory pointers to retrieve W[i] enables faster operation in indirect autoincrement mode (2 clock cycles) than using the absolute addressing mode (3 cycles).

2.3 System-Level Optimizations

We implemented several key strategies aimed at improving the performance for the MSP430FR5994. First, we overclocked the system to 24 MHz, which is out-of-specs, but the system remained stable at this frequency. However, code execution from FRAM causes a bottleneck, since it cannot be accessed at frequencies above 8 MHz and a small instruction cache is used, which suffers from frequents misses in the large SHA-1 routine. To address this, we moved the SHA-1 function into SRAM, ensuring that it can run without being constrained by the speed of code fetching. Nevertheless, with only 8KB of SRAM available, we faced significant space limitations. The fully-unrolled implementation of SHA-1 only fitted in SRAM after applying the assembly-level optimizations mentioned earlier, which significantly reduced the memory footprint. FRAM is still briefly used during the program as an efficient mechanism for storing non-volatile checkpoints after each hash calculation.

- `OPT-4`: The main routine is relocated to SRAM to avoid a critical bottleneck caused by the slower access speed of FRAM. This has also the advantage of a reduction in the supply current, since misses from the FRAM instruction cache significantly increase power consumption [3].
- `OPT-Startup`: Moving the function to SRAM notably increases the startup time, since the whole function needs to be copied from the FRAM during the initialization. Since we practically fill the whole SRAM this may take a few tenths of milliseconds, what would prevent the program to reach the next checkpoint in a worst-case scenario in which only very short bursts of energy are available. To prevent this, we increase the clock frequency to 24 MHz before the C-startup code, decreasing the startup time to a few milliseconds.

3 CONCLUSIONS

Tested compilers (GCC, IAR, TI) for MSP430 are not able to compete in terms of speed with a properly designed assembly implementation, since for a complex function they introduce inefficiencies in register allocation, increased memory store-and-load with expensive addressing modes and do not fully take advantage of potential gains provided by special CPU instructions. Overclocking the CPU to 24 MHz proved effective and energy-efficient only when paired with SRAM execution. With all the optimizations, including overclocking before the C-startup routine, the system is able to finish the first SHA-1 block and reach the initial checkpoint in only a few milliseconds, while subsequent checkpoints are reached every 155 µs, enabling operation even in the harshest environments.

Table 1: Time to compute a SHA-1 block at 24 MHz after successively applying the proposed optimizations. Architecture-specific OPT-3 and OPT-4 provide the highest gains.

OPTs	Time	Delta from previous row
Naive	640 µs	C-compiled with speed optimizations
1	600 µs	Precomputation of constant XORs
1, 2	560 µs	Optimizing rounds in the state words
1, 2, 3	220 µs	Assembly implementation (~ 2.5× faster)
1, 2, 3, 4	155 µs	Execution from SRAM (~ 1.4× faster)

REFERENCES

[1] Adam Back et al. 2002. Hashcash-a denial of service counter-measure. http://www.hashcash.org/papers/hashcash.pdf Accessed: 2024-09-25.

[2] D. Eastlake and P. Jones. 2001. RFC3174: US Secure Hash Algorithm 1 (SHA1).

[3] Texas Instruments. 2016. *SLAA728: Optimizing Active Mode Current Consumption on MSP Devices.* Technical Report.

[4] Andrea Francesco Iuorio and Andrea Visconti. 2019. Understanding optimizations and measuring performances of PBKDF2. In *2nd International Conference on Wireless Intelligent and Distributed Environment for Communication: WIDECOM 2019.* Springer, 101–114.