Fast Embedded Software Hashing

Dag Arne Osvik*

EPFL IC IIF LACAL, Station 14, CH-1015 Lausanne, Switzerland
dagarne.osvik@epfl.ch

Abstract. We present new software speed records for several popular hash functions on low-end 8-bit AVR microcontrollers. Target algorithms include widely deployed hash functions like SHA-1 and SHA-256 as well as the SHA-3 (second round) candidates Blake-32 and Skein-256. A significant aspect of our implementations is that they reduce the overall resource requirements, improving not only execution time but also RAM footprint and sometimes ROM/Flash memory footprint at the same time, providing the best memory/performance trade-offs reported so far. We believe that our results will shed new light on the ongoing SHA-3 competition, and be helpful for the final stage of the competition.

1 Introduction

Cryptographic hash functions are one of the most widely used cryptographic primitives for security applications. They have been extensively deployed over the last few decades, especially after digital signatures became more and more popular. Moreover, many other "ancillary" applications, including hash-based message authentication codes, pseudo random number generators and key derivation functions, make use of cryptographic hash functions as their main underlying primitive.

To meet the requirements for such sensitive applications, a myriad of hash functions have been developed and (some of them) standardized, such as MD4 [18], MD5 [19], SHA-0, SHA-1 [20] and the SHA-2 [20] family, with various output lengths ranging from 128 to 512 bits. These hash functions all tend to be quite fast on modern processor architectures, in particular workstations (with 32 or 64-bit CPUs). Compared to their sister primitive blockciphers, these hash functions are much faster for the same input length. As a result, this has established the common belief that a hash function should be efficient, in particular fast; yet being fast is necessarily related to being lightweight in terms of underlying operations, which turned out to cause quite dramatic consequences for security.

That having been said, these widely deployed hash functions share another rather unexpected property as well. Namely, starting from 2004, collisions have been successfully found for MD4 [18], MD5 [19], and SHA-0; moreover, several algorithms for SHA-1 collisions, requiring less than the expected amount of work, have been published without yet being able to provide collisions. Nevertheless, it is widely *believed* that SHA-1 is broken and should not be used in sensitive applications requiring collision resistance (preimage attacks seem beyond practical reach with current technology); this leads to the question of whether the younger members of the SHA family will share the same destiny.

Despite the lack of any practical (or even theoretical) attack on any member of the newer SHA-2 family, this erosion of trust led NIST to decide to develop a new algorithm. For this purpose, a public competition [15] was announced by the NIST to develop a new cryptographic hash algorithm intended to replace the current SHA-2 standard [20]. The main reason they mentioned was that "*a successful collision attack on an algorithm in the SHA-2 family could have catastrophic effects for digital signatures.*" The new hash algorithm will be called SHA-3 and will be subject to a Federal Information Processing

* Supported by Microsoft Innovation Cluster for Embedded Software project

Standard (FIPS) as done for the Advanced Encryption Standard (AES) [4]. The competition officially started in late 2008 with submissions from all over the world including contributors from academia, industry and government institutions. As a result, 64 proposals were received, of which 51 met the minimum submission requirements and became the first round candidates.

In summer 2009 the number of candidates for the second round was cut down to a more manageable size of 14 by eliminating the ones having major security or performance flaws. Indeed, as reported in the competition announcement: "*NIST expects SHA-3 to have a security strength that is at least as good as the hash algorithms currently specified in FIPS 180-2, and that this security strength will be achieved with significantly improved efficiency.*" Note however that despite suffering from minor security issues, some of the high-performing candidates survived for the second round [14]; this clearly shows the importance of efficiency in the evaluation procedure. The total number of candidates was reduced to five finalists in December 2010 [16], and the new hash function standard will be announced in 2012.

Although standard workstations are stated as the *reference platforms* by NIST (to evaluate the software performance), they also stress that it is preferable if *"the algorithm can be implemented securely and efficiently on a wide variety of platforms."* For this purpose, several papers have been published (e.g. [10,11]) showing improved software performance results for SHA-3 candidates on architectures ranging from standard workstations to smartcards and exotic platforms like GPUs and Cell BE.

Microcontrollers are one of the key components of embedded systems with applications in all of the major electronics markets, including the automotive industry, home appliances, home entertainment, industrial automation, mobile electronics and wireless sensor networks. They are mainly utilized instead of general purpose CPUs for space, power and cost saving reasons. It is expected that the microcontroller market will continue to grow rapidly in the near future. Indeed, this market is estimated [8] to reach a value of USD 16 billion for 2011, with an expected 9% annual revenue growth over the next five years.

It is not difficult to see the need for cryptography in embedded processors and lightweight applications. AES has attracted a lot of attention from the community due to its high performance on such devices. Moreover, it is believed to be one of the reasons that Rijndael was selected as AES performing superior to the other AES contestants on similar ubiquitous platforms. Motivated by this fact, this work investigates the case for the currently used hash functions SHA-1/SHA-256 and some of the SHA-3 candidates on Atmel's 8-bit AVR microcontroller family.

The reasons for chosing this particular platform are straightforward: it is a popular architecture in real-life applications; the hardware is cheap and easily available; it is easy to develop programs for it using freely available development tools; and our target algorithms had already been implemented on the same architecture, making the contributions of this work easy to evaluate.

2 Target Algorithms

Ideally, one would aim to implement all relevant algorithms with all possible improvements on the same architecture to get a good overall benchmark. However, this is no easy feat. Indeed, just to make a single efficient implementation one needs to invest a significant amount of effort. Nevertheless, we have selected several target algorithms for which we can make substantial improvements over older performance results.

Our first target algorithms are two members of the SHA family: SHA-1 and SHA-256. Their inclusion is straightforward given their influence in the practical world, where the former is one of the most deployed hashing algorithms and the latter is the currently suggested standard. They are also the benchmark against which the SHA-3 candidates are measured.

One might argue that SHA-1 will soon retire and any performance improvement for SHA-1 is pointless. However, it is still preimage resistant, and many protocols for lightweight cryptography employing hash functions require only preimage resistance [3,5,6,21], making SHA-1 a sound candidate for such applications. We also show in this work that its resource requirements are quite competitive.

Regarding SHA-3 candidates, on the other hand, our selection process was more complex given the 14 candidates at that time, but in the end we decided to start this part of our work with BLAKE-32 and Skein-256. These two algorithms are among the five candidates that went on to become finalists.

3 Target Platform

For our work targeting low-end embedded systems we chose Atmel's AVR family of 8-bit microcontrollers. This family is widely used in embedded systems, is quite easy to program using GNU tools, and provides a relatively simple architecture in which we can explore tradeoffs between code size and execution speed.

Important features of the architecture include 32 registers of 8 bits each, mostly single-cycle execution of instructions, 16-bit pointer registers and addressing modes with pre-decrement or post-increment of the pointer value, separate program and data memories (Harvard architecture), hardware stack support, memory-mapped registers, and relative and indirect calls (`rcall`/`icall`) and branches (`rjmp`/`ijmp`).

The results from most instructions are written back to an input register, like on x86 processors. The stack pointer is a separate pair of registers and not part of the general-purpose register file, making it possible to use all 32 general-purpose registers for program data in fully reentrant (and interrupt safe) code.

AVRs with more memory than can be addressed with 16 address bits have support for extending the address, but we are not using this feature. Note that RAM can be very small, in some chips even not present apart from the register file, while program memory is comparatively large and useful for lookup tables with constant data.

The X, Y and Z registers consist of one pair of 8-bit registers each, overlapping the top 6 registers of the register file. One useful result of this is that for an aligned 256-byte lookup table, we can keep the upper half of the address in the high register and use the low register as index register without any further address computations.

There are optional instructions for loading and storing data in program memory; hence AVR processors that implement them have a *modified* Harvard architecture, but only the Z register may be used for addressing in this memory. Memory addressing with a 6-bit unsigned displacement is possible, but only with the Y and Z registers, and not for program memory. This turned out to prevent an optimization of SHA-1 for which we would need two pointers with displacement and a third to address program memory. The Z register is also the only one that may be used for indirect calls and branches.

Another interesting feature is the memory mapping of the 32 registers at the beginning of the address space, making it possible to write very simple and tiny loops for things like reading a block of data into registers.

Conditional branches are limited to a 7-bit signed displacement, which means a simple inner loop is limited to 64 instructions, including the branch instruction itself.

4 Approach

We first give an informal overview of our approach to developing our implementations. No strict development methodology has been followed. Instead we have tried to combine the necessary creative freedom with sound software engineering.

To achieve maximum performance we need access to low-level features of the processor architecture, which we only get at the assembler level. However, we can use macros and higher-level languages to create assembly code, and in our work we have used both M4, C macros and regular C programs for this purpose. The tools help us make the code more readable and easy to debug, but our particular choice of languages is simply a matter of availability and personal preferences.

When creating an efficient implementation, we first need to understand in detail the inner workings of the algorithm, and find out which computations are strictly necessary. Then we can combine this with what instructions, registers and addressing modes are available on a given processor architecture to find efficient ways of performing the computations. Already at this stage we take into account possibilities like free permutations through renaming of registers, reordering of loads and stores, or changing data layout in memory.

Note that in this work we are looking at how to make high-performance software implementations while avoiding excessive resource (RAM/ROM) requirements. That is, we aim for good speed/memory trade-offs, giving higher priority to running time and RAM footprint than ROM footprint (code size). Hence we do not take into account any specific hard limits or costs that could be constraining a real-world application, although our heavier weighting of RAM use than ROM use is due to RAM typically being about an order of magnitude smaller than ROM in embedded devices. Other interesting data points for performance trade-offs for these algorithms would include maximizing performance while keeping RAM and ROM use close to the minimum possible.

Given our priorities, we start by optimizing for straight-line code, without any loops or function calls. As always in performance optimization, we focus first and foremost our attention and efforts on the most time-consuming part, like the round function of a hash algorithm or block cipher.

Next we look at how to minimize the footprint of our code in RAM and ROM, while minimizing the resulting increase in run time. This involves e.g. identifying opportunities for sharing code between different iterations of a round function and moving their differences to short entry point stubs which load round constants into registers before jumping or falling through to the shared code of the function. Obviously, the more frequently a particular part of the code will be executed, the more memory we are willing to spend to keep that part of the code fast.

Once we have worked out how to implement the algorithm, it is time to write the actual code, compile it, and run it on the targeted hardware. For assembly programs in particular it is very useful to have code that shows us the entire (relevant part of the) state of the processor, and compare the actual state with the corresponding state of a reference implementation. This way we gradually build the new implementation while checking that the code really does what it is supposed to do.

5 Implementations

5.1 SHA-1

SHA-1 is considered broken[22,7], but remains a popular hash function. The SHA-1 algorithm uses a word size of 32 bits and produces a message digest of 160 bits. Just as with the older MD5 the input message m is processed in blocks of 512 bits after initial padding.

All the individual message blocks go through two parts; first each 512-bit block is expanded to 2560 bits (80 words of 32 bits). This message expansion is done by computing the recurrence relation

$$w_i = \begin{cases} m_i & \text{if } 0 \le i \le 15, \\ (w_{i-3} \oplus w_{i-8} \oplus w_{i-14} \oplus w_{i-16}) \lll 1 & \text{if } 16 \le i \le 79 \end{cases}$$

for each block, where w_i and m_i denote the i^{th} word of the expansion and the message respectively. Next, this expanded block goes through 80 rounds, one round per word, of the following recurrence relations:

$$\begin{aligned}
e_{i+1} &= d_i \\
d_{i+1} &= c_i \\
c_{i+1} &= b_i \lll 30 \\
b_{i+1} &= a_i \\
a_{i+1} &= (a_i \lll 5) + f_i(b_i, c_i, d_i) + e_i + k_i + w_i
\end{aligned}$$

where the k_i are a set of fixed values (one for every 20 rounds), and the f_i are defined as follows:

$$f_i(X,Y,Z) = \begin{cases} (X \wedge Y) \oplus (\bar{X} \wedge Z), & 0 \le i \le 19, \\ X \oplus Y \oplus Z, & 20 \le i \le 39, \\ (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z), & 40 \le i \le 59, \\ X \oplus Y \oplus Z, & 60 \le i \le 79. \end{cases}$$

Note that for rounds 0 through 19 the f_i perform a select operation (bitwise "if X then Y else Z"), and for rounds 40 through 59 they compute the majority function.

Observe that the words of the expanded message (w_i) are used in order of increasing index. Therefore, once w_{i-16} has been used by the compression function, we can always replace w_{i-16} by w_i, eliminating the need for extra storage for the expanded message. We can freely choose how many used message words to update in one go, and for our AVR implementation we have chosen to update all 16 at once. This allows the message expansion to be implemented as a very straightforward function, without any runtime computation of where to find the input words for the recurrence relation.

With the expansion performing 16 updates in one go, it only needs to be called four times, making function call overhead relatively insignificant. The expansion starts by saving part of the round function state to the stack, freeing up registers so that we can reduce the number of loads per iteration of the recurrence relation as follows. We enter the loop with w_{i-16} and w_{i-15} already in registers, then load and xor each of w_{i-3}, w_{i-8} and w_{i-14} into w_{i-16}. Finally we rotate the result, store it as the new w_i, and prepare for the next iteration by copying (in registers) w_{i-15} over w_{i-16} and w_{i-14} (the one we loaded last) over w_{i-15}.

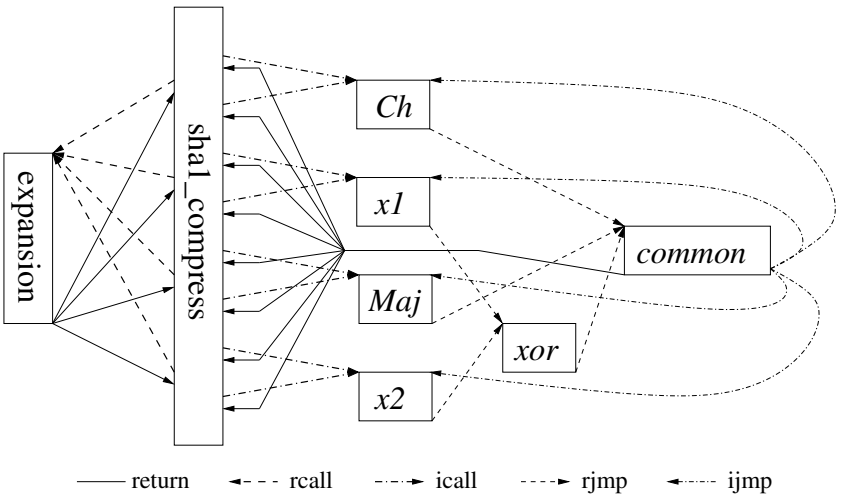

Fig. 1. Control flow of our SHA-1 implementation.

Most of the computations for the round function are the same for all rounds, and for half of them (20-39 and 60-79) everything apart from the round constant is the same. To account for the differences we split the computations into 4 functions (*Ch*, *x1*, *Maj* and *x2*), one for each group of 20 consecutive rounds. The functions *x1* and *x2* only differ in the round constant used, so their only task is to load the appropriate round constant before jumping to *xor*, which computes f_i before jumping to the common code (*common*). Functions *Ch* and *Maj* also load round constants, compute f_i and then jump to *common*. Finally, *common* completes the round function computation and then either continues by jumping back to the starting point of the current round function with an indirect jump, or returns to the core compression function. The address for the jump is put in the Z register by the compression core and used for an indirect call to the round function, then kept and reused for the indirect jump during round function iterations. The entire internal control flow of the core of our SHA-1 implementation is illustrated in Figure 1.

5.2 SHA-256

The SHA-256 algorithm is one of the four hash functions of the SHA family published after SHA-1, collectively known as SHA-2. Cryptanalysis over the last years [9,13] has not revealed any real weakness. However, researchers are wary due to SHA-2 having the same overall structure as the other members of its family.

Like SHA-1, SHA-256 is designed for a word size of 32 bits, and processes its input message in blocks of 512 bits, with padding at the end to ensure the total input length is a multiple of 512 bits. Each of the blocks is then expanded to 2048 bits (64 words of 32 bits) before it is processed by the compression function. First the 512 bits of the message block are divided into words named $w_0 \dots w_{15}$. Then $w_{16} \dots w_{63}$ are computed as follows:

$$\begin{aligned}
\sigma_{0,i} &= (w_{i-15} \ggg 7) \oplus (w_{i-15} \ggg 18) \oplus (w_{i-15} \ggg 3) \\
\sigma_{1,i} &= (w_{i-2} \ggg 17) \oplus (w_{i-2} \ggg 19) \oplus (w_{i-2} \ggg 10) \\
w_i &= w_{i-16} + \sigma_{0,i} + w_{i-7} + \sigma_{1,i}
\end{aligned}$$

Next the extended message block $(w_0 \dots w_{63})$ goes through 64 iterations of the following round function:

$$\begin{aligned}
\tau_{0,i} &= (a_i \ggg 2) \oplus (a_i \ggg 13) \oplus (a_i \ggg 22) \\
Maj_i &= (a_i \wedge b_i) \oplus (a_i \wedge c_i) \oplus (b_i \wedge c_i) \\
t_{2,i} &= \tau_{0,i} + Maj_i \\
\tau_{1,i} &= (e_i \ggg 6) \oplus (e_i \ggg 11) \oplus (e_i \ggg 25) \\
Ch_i &= (a_i \wedge b_i) \oplus (\bar{a}_i \wedge c_i) \\
t_{1,i} &= h_i + \tau_{1,i} + Ch_i + k_i + w_i \\
(h_{i+1}, g_{i+1}, f_{i+1}, e_i) &= (g_i, f_i, e_i, d_i + t_{1,i}) \\
(d_{i+1}, c_{i+1}, b_{i+1}, a_i) &= (c_i, b_i, a_i, t_{1,i} + t_{2,i})
\end{aligned}$$

where *Maj* and *Ch* represent the (results from) majority and choice operations.

Unlike SHA-1, SHA-256 uses a different constant value (k_i) for each round. A natural approach to handling this is to put the constants in a table, and load them one by one as needed, and this part of the round function code can always be the same. To support this approach we would need memory to store the 64 constants of four bytes each, as well as a pointer to the current element in the table of constants. If this table were to reside in the Flash memory of the AVR, we would be forced to keep the pointer in the Z register, while to be able to keep the pointer in the X or Y registers we would need to spend valuable RAM to store the table.

The big state of the round function, as large as the entire register file of AVR processors, means we cannot keep it all in registers while leaving room for computations and any necessary pointers. Instead we keep only the a and e values (eight bytes total) in registers and the rest in temporary storage on the stack, loading and storing them as needed.

In the end the combinations of constraints led us to a design with a fully unrolled main loop performing a sequence of calls to functions, with the Y and Z registers being used for the message block and temporary state. The round constants are embedded in the code of the unrolled main loop, removing the need for a separate table and pointer to them. In effect we embed the constant table in the code and use the instruction pointer to load its elements.

Round function computations are split into the following functions:

f0 : Load b, c, f, g. Compute t_2 and $\tau_1 + Ch$.
f1 : Load $w_{i-16}, w_{i-15}, w_{i-7}, w_{i-2}$. Compute w_i.
f2l : Load w_i, fall through to *f2*.
f2 : Load d and h. Compute a and e.

In the first 16 rounds, which use $w_0, \ldots, w_{15}$ directly, we call *f2l* to load w_i before proceeding with *f2*. For the remaining 64 rounds we use *f1* to compute w_i before calling *f2*. When w_i has been computed it is written to the position of w_{i-16}, as w_{i-16} is not needed for the computation of any later values. The computation of w_i differs depending on the relative positions of w_{i-16}, w_{i-15}, w_{i-7} and w_{i-2}, as these wrap around to the beginning of the message block whenever they reach the end.

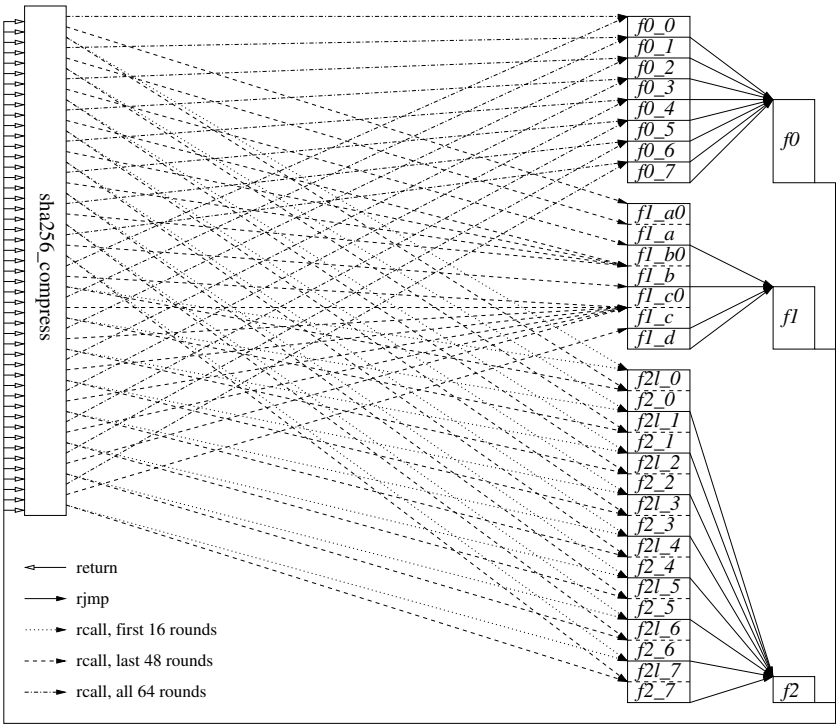

Fig. 2. Control flow for each group of sixteen rounds of our SHA-256 implementation.

Control flow of the core of our SHA-256 implementation is illustrated in Figure 2. Sixteen rounds of the core compression function are shown, each round making a sequence of function calls (using rcall) to the various entry points of *f0*, *f1* and *f2*. From each entry point we jump (rjmp) or fall through to the shared code, which ends with a return instruction transferring control back to the core. For brevity we only show sixteen of the 64 unrolled rounds; differences are indicated with dotted and dashed lines.

Our implementation leaves one pointer register (without displacement support) that can be used as a constant pointer, allowing us to copy constants to the stack e.g. every 1, 2, 4, 8 or 16 rounds and removes the need for full unrolling of sha256_compress. Since we are using a pointer to the state we can also further share code for *f0* and *f2* in much the same way we have done for the key expansion code. We estimate that these improvements (which we have not implemented) can reduce the total code size (including the table of constants) to about 2 kilobytes, with very little slowdown but up to 64 bytes extra RAM needed, and more complex control flow.

5.3 BLAKE

BLAKE[1] is one of the five finalist candidates for the new SHA-3 standard. Like SHA-1 and SHA-256, BLAKE processes its input as a sequence of 512-bit message blocks after adding padding. However, unlike SHA-1 and SHA-256, the BLAKE compression function has no message block expansion, just permuted reuse of 32-bit words of the message block. This is somewhat similar to MD5, but with ten instead of four permutations. All operations work on 32-bit values throughout the algorithm.

The BLAKE-32 compression takes four input values; the 256-bit chaining value h, the 512-bit message block m, the 128-bit salt s and the 128-bit counter t. From these values and sixteen 32-bit constants $c_0, \ldots, c_{15}$ a new 256-bit chaining value h' is computed.

The internal state ν is initialized as follows:

$$\begin{pmatrix} \nu_0 & \nu_1 & \nu_2 & \nu_3 \\ \nu_4 & \nu_5 & \nu_6 & \nu_7 \\ \nu_8 & \nu_9 & \nu_{10} & \nu_{11} \\ \nu_{12} & \nu_{13} & \nu_{14} & \nu_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_1 \oplus c_5 & t_0 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}$$

Next the BLAKE round function is applied ten times, each round consisting of

$$G_0(\nu_0, \nu_4, \nu_8, \nu_{12}),\ G_1(\nu_1, \nu_5, \nu_9, \nu_{13}),\ G_2(\nu_2, \nu_6, \nu_{10}, \nu_{14}), G_3(\nu_3, \nu_7, \nu_{11}, \nu_{15}),$$
$$G_4(\nu_0, \nu_5, \nu_{10}, \nu_{15}), G_5(\nu_1, \nu_6, \nu_{11}, \nu_{12}),\ G_6(\nu_2, \nu_7, \nu_8, \nu_{13}),\ G_7(\nu_3, \nu_4, \nu_9, \nu_{14}),$$

where each $G_i(a, b, c, d)$ performs this sequence of computations for round r:

$$\begin{aligned} a &\leftarrow a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\ d &\leftarrow (d \oplus a) \ggg 16 \\ c &\leftarrow c + d \\ b &\leftarrow (b \oplus c) \ggg 12 \\ a &\leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\ d &\leftarrow (d \oplus a) \ggg 8 \\ c &\leftarrow c + d \\ b &\leftarrow (b \oplus c) \ggg 7 \end{aligned}$$

The functions σ_r are ten permutations of the sequence $\{0, \ldots, 15\}$. Apart from σ_0 being the identity permutation, these are all random-looking, preventing us from optimizing our implementation based on patterns inside them. Note that $G_0, \ldots, G_3$ are independent and can be performed in parallel (and hence in any order we may wish), and the same holds for $G_4, \ldots, G_7$.

Finally the new chaining value h' is computed from the initial state h, the salt s and the internal state ν:

$$\begin{aligned} h'_0 &\leftarrow h_0 \oplus s_0 \oplus \nu_0 \oplus \nu_8 \\ h'_1 &\leftarrow h_1 \oplus s_1 \oplus \nu_1 \oplus \nu_9 \\ h'_2 &\leftarrow h_2 \oplus s_2 \oplus \nu_2 \oplus \nu_{10} \\ h'_3 &\leftarrow h_3 \oplus s_3 \oplus \nu_3 \oplus \nu_{11} \\ h'_4 &\leftarrow h_4 \oplus s_0 \oplus \nu_4 \oplus \nu_{12} \\ h'_5 &\leftarrow h_5 \oplus s_1 \oplus \nu_5 \oplus \nu_{13} \\ h'_6 &\leftarrow h_6 \oplus s_2 \oplus \nu_6 \oplus \nu_{14} \\ h'_7 &\leftarrow h_7 \oplus s_3 \oplus \nu_7 \oplus \nu_{15} \end{aligned}$$

BLAKE has the nice (but not so obvious) property that we can reorder its computations in a way that allows us to get performance close to that of a straight-line implementation while reducing code size significantly. We decided to split the computations of the BLAKE round in three major parts:

f0 : Load $c_{\sigma_r(2i)}$. Shared: xor with $m_{\sigma_r(2i+1)}$.
f1 : Load $c_{\sigma_r(2i+1)}$. Shared: xor with $m_{\sigma_r(2i)}$.
g : Store output from g_{i-1}, load input for g_i. Shared: perform the rest of G_i.

In our implementation the g_i share most (656 bytes) of their code, and we use eight function entry points to execute the small amount of code that differs, followed by a jump (or fall-through) to the shared code. We also use sixteen function entry points for each of *f0* and *f1*, one for each of the constants $c_0, \ldots, c_{15}$. Total code size for *g*, *f0* and *f1* is 1008 bytes.

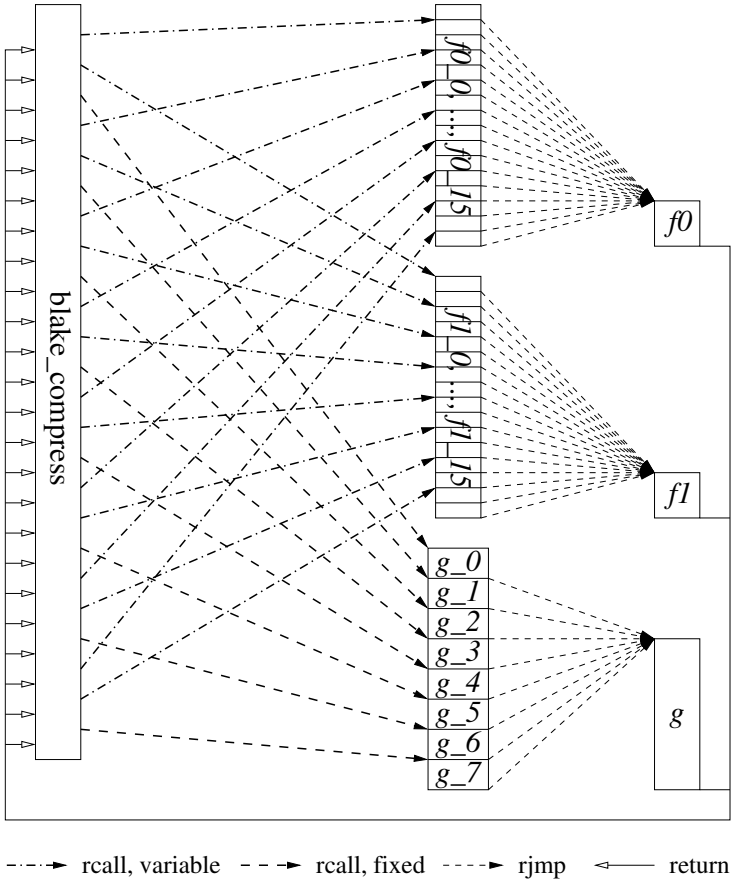

Fig. 3. Control flow of a single round of our BLAKE implementation. The variable rcall arrows change their destination in every round.

Given these 40 function entry points, we can program each application of G_i as a series of three function calls – one to $f0_{\sigma_r(2i+1)}$, one to $f1_{\sigma_r(2i)}$, and one to g_i – with two pointer adjustments as needed to allow *f0* and *f1* to read the correct message word. Due to the complexity of the σ permutations we have unwound the entire top level ("blake_compress") of the round function code. Still, with at most five instructions (ten bytes) per application

of G_i, the code size of blake_compress for the ten rounds in BLAKE-32 is relatively modest at 756 bytes.

The control flow of one round of our BLAKE implementation is illustrated in Figure 3. The variable rcall arrows are permuted according to the σ_r permutations; only the first round is performed exactly as depicted.

Again, like for SHA-256, we have discovered further enhancements to our implementation of BLAKE. By adjusting the state pointer between applications of the various G_i, we can share more of their code, while keeping the runtime cost very low. The reason we can do this is easily seen from the very simple patterns in Table 1. The skewed indices simply

i	state elements modified by G_i	
	standard	skewed
0	0 4 8 12	0 4 8 12
1	1 5 9 13	0 4 8 12
2	2 6 10 14	0 4 8 12
3	3 7 11 15	0 4 8 12
4	0 5 10 15	0 5 10 15
5	1 6 11 12	0 5 10 11
6	2 7 8 13	0 5 6 11
7	3 4 9 14	0 1 6 11

Table 1. Standard and skewed indices of BLAKE state

have $i \mod 4$ subtracted from them. The first four lines are identical in the column with skewed indices, meaning that their code for loading and storing state words can also be identical. The last four also have a lot in common, although they are all different; they can utilize two sequences of code, one handling elements 15, 10, 5 and 0, and one handling elements 1, 6 and 11, in that order, with one entry point for each element and each falling through to the next in its sequence. The pseudo-assembler code in Table 2 illustrates how to write this. Together with the same setup for writing elements back to the state, and the

```
lc:    load element 0
       load element 4
       load element 8
       load element 12
       return
ld15: load element 15
ld10: load element 10
ld5:   load element 5
ld0:   load element 0
       return
ld1:   load element 1
ld6:   load element 6
ld11: load element 11
       return
```

Table 2. Compact code for loading BLAKE state using a moving state pointer

necessary adjustments of the state pointer, we can then perform the necessary loading and storing with only the above 11 loads and stores, compared to the 32 in our current code. According to our calculations, this would let us save 298 bytes of code, need 2 extra bytes

of RAM (one more level of function calls) and 98 more cycles per block (about 3 cycles per byte).

5.4 Skein

Skein[2] is another SHA-3 finalist candidate, and uses a tweakable block cipher called Threefish as its mixing function. The round function design of Threefish is even simpler than those of the old SHA algorithms, and its main part, the MIX function, consists only of 64-bit addition, rotation and exclusive-or operations. All operations work on 64-bit values throughout the algorithm. As we have only targeted 256-bit Skein in this work, we only describe Threefish-256 and can simplify our notation accordingly (refer to the Skein specification document for the complete version).

Threefish-256 is a block cipher taking as inputs a 256-bit plaintext, a 256-bit key and a 128-bit tweak value, producing a 256-bit ciphertext as output. A round consists of two MIX functions, each applied to one half of the state, and then a swapping of words between the two halves. Every four rounds a round key is added to the full state. Rotation constants applied in the MIX function repeat every 8 rounds. In total 72 rounds are performed, or 9 times the 8 rounds of computation depicted in Figure 4, plus an addition of the round key after the last round.

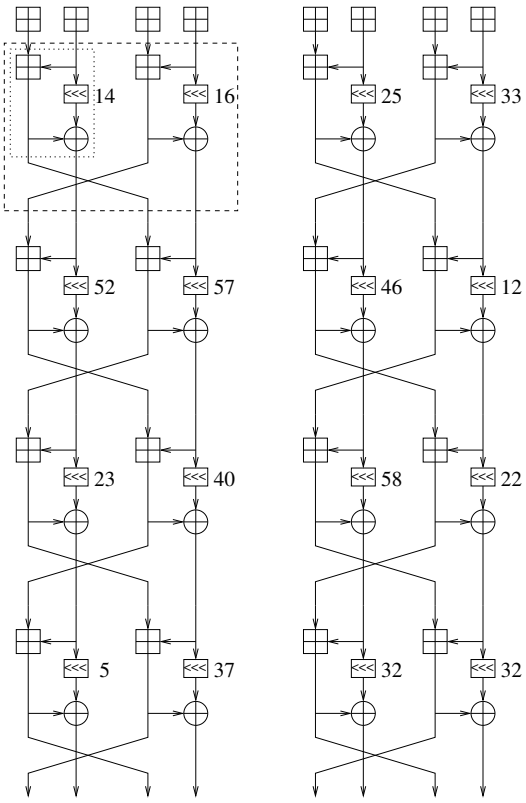

Fig. 4. Threefish-256 quad-rounds; even (*qr0*) on the left, odd (*qr1*) on the right. The dotted-line box shows the contents of one MIX function, and the dashed-line box contains one round. Before each quad-round a round key ($k_{s,0}, \ldots, k_{s,3}$; not shown) is added to the intermediate state.

The size of the internal chaining variables of Threefish exactly match the size of the general-purpose register file, and the round function computations do not need any temporary variables (apart from the carry flag). This allows us to keep the entire state in registers during the quad-round functions, removing all need for loading or storing data between individual rounds. The rest of the state of the function is stored on the stack and in 5 fixed RAM locations during those quad-round computations. Using fixed memory locations allows faster instructions to be used for loading and storing data, but also makes the code non-reentrant. This means that a call to the compression function can not be stopped in the middle of execution to make way for another thread of execution that would then also call the compression function. However, there is nothing preventing the use of multiple simultaneous hash contexts. Also, as long as no interrupt code uses the same memory locations, our implementation remains interrupt safe. It is possible to avoid non-reentrancy at moderate cost by storing to the stack instead, but we have not investigated the exact cost of doing this.

The Threefish-256 key schedule is based on the 256-bit key $(k_0, \ldots, k_3)$ and the 128-bit tweak (t_0, t_1). From these values two more are computed: $t_2 = t_0 \oplus t_1$ and $k_4 = C_{240} \oplus k_0 \oplus k_1 \oplus k_2 \oplus k_3$, where C_{240} is a constant (the decimal number 240 encrypted with AES-128 under the all-zero key) used to ensure that the extended key $(k_0, \ldots, k_4)$ cannot be all zero. Finally, the round keys are computed as follows:

$$
\begin{aligned}
k_{s,0} &= k_{s(mod\ 5)} \\
k_{s,1} &= k_{s(mod\ 5)} + t_{s(mod\ 3)} \\
k_{s,2} &= k_{s(mod\ 5)} + t_{(s+1)(mod\ 3)} \\
k_{s,3} &= k_{s(mod\ 5)} + s
\end{aligned}
$$

where s is the round number divided by 4, as Threefish only applies a round key for every 4 rounds of mixing.

As we can see from this, all the $k_0, \ldots, k_4$ and t_0, t_1, t_2 are used multiple times. Computing all the $k_{s,i} | s \in \{0, \ldots, 9\}, i \in \{0, \ldots, 3\}$ and storing them in a lookup table would be relatively simple and fast, but would also require 320 bytes of RAM. On the other extreme, computing k_4 and t_2 on the fly every time we need them would not require any extra storage apart from register spills, but would waste computing time. The solution we chose is somewhere inbetween, extending the storage for the Skein context structure to also include k_4 and t_2. Keeping all of the k, t and plaintext values in the context structure means we can use a single pointer to access them all, without having to first copy them to the stack. This keeps register pressure down (no need for multiple pointers), saves stack space, and saves the time required for copying data to the stack. The only downside is that the context structure itself is expanded from 80 to 96 bytes, which could be a problem in an application needing to keep many simultaneous contexts.

The resulting code for Skein has the very simple control flow shown in Figure 5; this flow is the simplest among all implementations presented in this paper. The complexity of Skein is mostly in details like memory and register allocation and how precisely to perform the word rotations. One little optimization detail is that it is not necessary to perform the addition of the round number as part of the round key addition for round 0. This is illustrated by the first call to *qr0* skipping its first part.

Like for SHA-256 and BLAKE, our Skein implementation also has room for further improvements, and in this case it is relatively trivial. Note that each of the quad-round functions gets called only once, and that they all end with a jump to the corresponding common code. Each of those calls can be replaced with the code that is now being called, followed by a call (not jump) to the common code, and the jump instruction is then no longer needed, saving two bytes and two cycles for each of the 18 jumps in the current code. The resulting very simple control flow is shown in Figure 6.

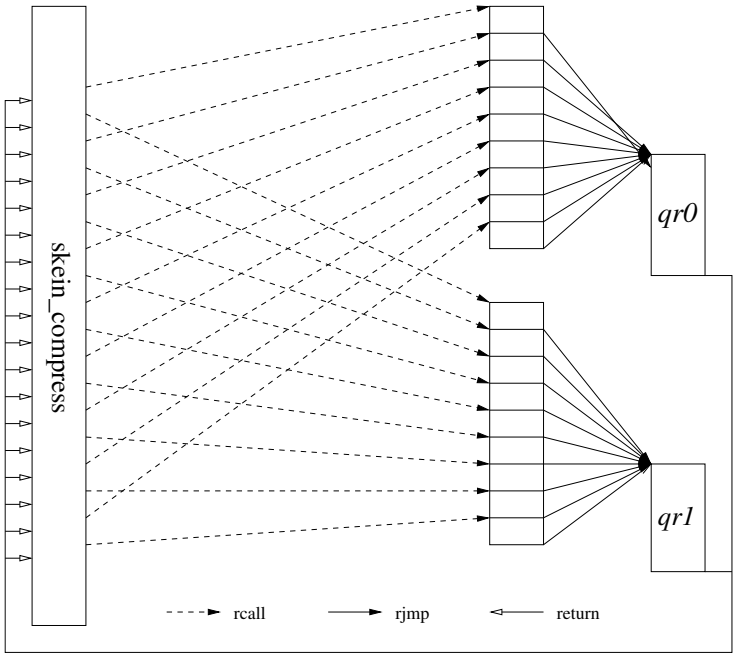

Fig. 5. Control flow of our Skein implementation

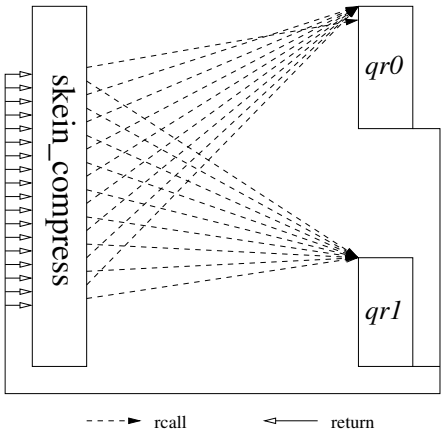

Fig. 6. Control flow of further optimized Skein implementation

6 Results

Table 3 lists our results (for actual running code) together with the best results we have found for other implementations, while Table 4 provides a list of relative improvements. In the cases where we only implemented the compression functions, only the speed and RAM footprint numbers are directly comparable, and we only give an upper bound on the relative improvement in code size.

Algorithm	**Reference**	**Time (cycles/byte)**	**RAM (bytes)**	**ROM (bytes)**	**Notes**
SHA-1	[17]	579	198	1022	
SHA-1	New	177	122	1352	
SHA-256	[17]	783	416	1598	
SHA-256	New	335	158	2720	
Blake-32	[17]	1115	245	6684	
Blake-32	[10]	324	251	1804	
Blake-32	New	263	206	2076	cf only
Skein-256	[12]	300	201	3200	
Skein-256	New	287	123	2464	v1.2, cf only

Table 3. Speed of compression functions (cf)

Algorithm	**Time**	**RAM**	**ROM**	**Notes**
SHA-1	3.27	1.62	0.76	
SHA-256	2.34	2.63	0.59	
Blake-32	1.23	1.22	< 0.87	cf only
Skein-256	1.045	1.63	< 1.30	cf only

Table 4. Relative improvements (> 1 is improvement)

We have shown that it is possible to increase the speed not only of the latest SHA-3 candidates, which in some cases have been the subject of significant optimization efforts, but also the much older SHA-1 and SHA-256 functions. These have been around since 1995 and 2001, respectively, and the AVR architecture was originally developed in 1996. Still, after these have been around for a decade or more, we were surprised to find that it was possible to get the large speedup factors in Table 4 for SHA-1 and SHA-256 compared to the best previous assembler implementations for which performance numbers are publically available.

The techniques we have employed are as far as we know not new themselves, but it seems like they have not previously been applied to this extent in implementations of cryptographic hash functions.

Our improved implementations of the older SHA-1 and SHA-256 standards mean that on the AVR architecture it is unlikely that SHA-3 will achieve NIST's stated goal of at least as good security as the older SHA functions while delivering significantly improved efficiency.

On the other hand, the graphs of control flows in our implementations are much simpler for the SHA-3 candidates, and we also achieved much lower relative speedups in these cases, despite the much shorter time people have had to study them. This combination suggests that programmers will have an easier time optimizing their implementations of these new algorithms, a useful feature for the future SHA-3 standard.

References

1. BLAKE website. http://131002.net/blake/.
2. Skein website. http://www.skein-hash.info.
3. The PHOTON Family of Lightweight Hash Functions. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference*, volume 6841, page 219, 2011.
4. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
5. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A Lightweight Hash. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.
6. Andrey Bogdanov, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, and Yannick Seurin. Hash Functions and RFID Tags: Mind the Gap. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 283–299. Springer, 2008.
7. Christophe De Cannière and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.
8. Electronics.ca Publications. http://www.electronics.ca/presscenter/articles/1364/1/Microcontroller-Market-Forecasted-to-Reach-Over-16-billion-worldwide-In-2011-/Page1.html.
9. Henri Gilbert and Helena Handschuh. Security Analysis of SHA-256 and Sisters. In Mitsuru Matsui and Robert J. Zuccherato, editors, *Selected Areas in Cryptography*, volume 3006 of *Lecture Notes in Computer Science*, pages 175–193. Springer, 2003.
10. Stefan Heyse, Ingo von Maurich, Alexander Wild, Cornel Reuber, Johannes Rave, Thomas Poeppelmann, and Christof Paar. "Evaluation of SHA-3 Candidates for 8-bit Embedded Processors". The Second SHA3 Candidate Conference, 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/HEYSE_EvaluationSHA-3Candidatesfor8-bitProcessors.pdf.
11. Joppe W. Bos and Deian Stefan. Performance Analysis of the SHA-3 Candidates on Exotic Multi-core Architectures, 2010. http://www.ee.cooper.edu/~stefan/pubs/conference/ches2010.pdf.
12. Jörg Walter. Fhreefish (Skein implementation) website. http://www.syntax-k.de/projekte/fhreefish/.
13. Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Analysis of Step-Reduced SHA-256. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2006.
14. NIST. Announcement of Second Round SHA-3 Candidates. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Email_Announcing_Round2_Candidates.pdf.
15. NIST. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.
16. NIST. The SHA-3 Finalists. http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Email_Announcing_Finalists.pdf.
17. Daniel Otte. AVR-Crypto-Lib, 2009. http://www.das-labor.org/wiki/Crypto-avr-lib/en.
18. R. Rivest. The MD4 Message-Digest Algorithm. RFC 1320 (Informational), April 1992.
19. R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.
20. "Secure Hash Standard". National Institute of Standards and Technology, NIST FIPS PUB 180-3, U.S. Department of Commerce, October 2008.
21. Adi Shamir. SQUASH - A New MAC with Provable Security Properties for Highly Constrained Devices Such as RFID Tags. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 2008.
22. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.