Parallelizing message schedules to accelerate the computations of hash functions

Shay Gueron[1,2], Vlad Krasnov[2]

[1] Department of Mathematics, University of Haifa, Israel
[2] Intel Architecture Group, Israel Development Center, Haifa, Israel

June 5, 2012

Abstract. This paper describes an algorithm for accelerating the computations of Davies-Meyer based hash functions. It is based on parallelizing the computation of several message schedules for several message blocks of a given message. This parallelization, together with the proper use of vector processor instructions (SIMD) improves the overall algorithm's performance. Using this method, we obtain a new software implementation of SHA-256 that performs at *12.11* Cycles/Byte on the 2nd and *10.84* Cycles/Byte on the 3rd Generation Intel® Core™ processors. We also show how to extend the method to the soon-to-come AVX2 architecture, which has wider registers. Since processors with AVX2 will be available only in 2013, exact performance reporting is not yet possible. Instead, we show that our resulting SHA-256 and SHA-512 implementations have a reduced number of instructions. Based on our findings, we make some observations on the SHA3 competition. We argue that if the prospective SHA3 standard is expected to be competitive against the performance of SHA-256 or SHA-512, on the high end platforms, then its performance should be well below *10* Cycles/Byte on the current, and certainly on the near future processors. Not all the SHA3 finalists have this performance. Furthermore, even the fastest finalists will probably offer only a small performance advantage over the current SHA-256 and SHA-512 implementations.

Keywords: SHA-256, SHA-512, SHA3 competition, SIMD architecture, Advanced Vector Extensions architectures, AVX, AVX2.

1 Introduction

The performance of hash functions is a significant workload of high end SSL/TLS and datacenters servers that perform data authentication (and encryption) at a large scale. In addition, SHA-256 [1] performance is the baseline for the SHA3 competition [11]. Furthermore, a truncation of SHA-512 (proposed in [3]) which has been recently standardized [12], implicitly extends the comparison baseline to SHA-512 as well. For this reason, we focus here on the performance of SHA-256 and SHA-512 on the ubiquitous x86_64 architectures, used in most server platforms.

SHA-256 and SHA-512 use the classical Davies-Meyer construction where the compression function is based on a block cipher, and message blocks are used as the

cipher's key. In this construction, the key expansion step of the compression function ("message scheduling" hereafter) depends only on the message, and is independent of the intermediate values of the computed digest (i.e., the ciphertext). This property allows for computing multiple message schedules in parallel.

In this paper, we propose a method called *n*-wise Simultaneous Message Scheduling (*n*-SMS for short), which parallelizes message scheduling of *n* message blocks, and uses Single-Instruction Multiple-Data (SIMD) instructions [5] to speed them up. The value of *n* depends on the hash algorithm and on the SIMD architecture that the algorithm runs on. For example, consider the new AVX architecture [5] where registers hold *128* bits of integer data, and the instructions can operate on both *32*-bit ("dword") and *64*-bit ("qword") elements. For SHA-256 whose message schedule operates on *32*-bit dwords, this architecture allows for parallelizing four message schedules, that is, using a *4*-SMS method. For SHA-512, whose message schedule operates on *64*-bit qwords, it allows for parallelizing two message schedules, using a *2*-SMS method.

Extended parallelism is facilitated by the new AVX2 architecture that Intel has recently announced, and will be first introduced in the next architecture (Codename "Haswell") in 2013 [8]. AVX2 (stands for Advanced Vector Extensions) architecture includes instructions that operate on integer elements stored in *256*-bit registers. It supports the use of *8*-SMS for SHA-256 (as well as for SHA-1) and *4*-SMS for SHA-512.

In this paper, we explain the *n*-SMS method and demonstrate how *2*-SMS, *4*-SMS and *8*-SMS can be used for SHA-256 and SHA-512 (as appropriate). We provide performance results on the 2nd and 3rd Generation Intel® Core™ processor, achieving *12.22* and *10.84* Cycles/Byte respectively for SHA-256, using *4*-SMS and the AVX instructions. For the future AVX2 architecture, no exact measurements can be made public. We therefore indicate the prospective performance benefit of *8*-SMS (for SHA-256) and *4*-SMS (for SHA-512) by counting the number of instructions in the compression function and comparing it to other implementations.

2 Preliminaries and notations

Figures 1 and 2 briefly describe SHA-256 and SHA-512 (the detailed definition can be found in FIPS180-2 publication [1]).

The SHA-256 and SHA-512 flows can be viewed as "Init" (setting the initial values), a sequence of "Update" steps (invocation of the compression function), and a "Finalize" step, which takes care of the padding of the message. Depending on the message's length, either one or two Update function calls may be required. Thus, the performance of SHA-256 / SHA-512 can be closely approximated by the number of Update function calls (N in Figures 1 and 2), which is given as a function of the message length ("length") in bytes as follows:

SHA-256:

$$N = \begin{cases} \left\lfloor \frac{length}{64} \right\rfloor + 2 & length \bmod 64 \geq 56 \\ \left\lfloor \frac{length}{64} \right\rfloor + 1 & else \end{cases}$$

SHA-512:

$$N = \begin{cases} \left\lfloor \frac{length}{128} \right\rfloor + 2 & length \bmod 128 \geq 112 \\ \left\lfloor \frac{length}{128} \right\rfloor + 1 & else \end{cases}$$

3 Parallelizing message schedules

3.1 The main observation: *n*-Simultaneous Message Scheduling

One of the classical approaches to construct a hash function (h) defines a compression function (c), operating on fixed length strings, and applying the Merkle-Damgård cascade. In many constructions, the compression function is based on a block cipher (E_K). We focus on the Davies-Meyer construction that has $c\ (H, M) = E_M\ (H) \oplus H$, where the message scheduling step is the key expansion of the underlying block cipher. In other words, Davies-Meyer construction requires frequent key expansions, each one being used for a single encryption. Consequently, this step can consume a significant portion of the compression function computations.

SHA-256	SHA-512
SHA-256 uses sixty four 32-bit words, $K_0^{256}, K_1^{256}, \dots, K_{63}^{256}$	SHA-512 uses eighty 64-bit words, $K_0^{512}, K_1^{512}, \dots, K_{79}^{512}$
+ denotes addition mod 2^{32} $SHR^n(x) = x \gg n$ $ROTR^n(x) = (x \gg n) \vee (x \ll (32 - n))$ $Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$ $Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$ $\Sigma_0^{256}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$ $\Sigma_1^{256}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$ $\sigma_0^{256}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$ $\sigma_1^{256}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$	+ denotes addition mod 2^{64} $SHR^n(x) = x \gg n$ $ROTR^n(x) = (x \gg n) \vee (x \ll (64 - n))$ $Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$ $Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$ $\Sigma_0^{512}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x)$ $\Sigma_1^{512}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$ $\sigma_0^{512}(x) = ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x)$ $\sigma_1^{512}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x)$
Preprocessing: • Append the bit '1' to the message. • Append k bits '0', where k≥0 is the smallest integer satisfying Len+k = 448 (mod 512), and Len is message's bit-length. • Append the vale Len, represented as a 64-bit integer in Big-Endian notation. • Set the initial hash value to $H_0^0 = 0x6a09e667; H_1^0 = 0xbb67ae85;$ $H_2^0 = 0x3c6ef372; H_3^0 = 0xa54ff53a;$ $H_4^0 = 0x510e527f; H_5^0 = 0x9b05688c;$ $H_6^0 = 0x1f83d9ab; H_7^0 = 0x5be0cd19$ • Parse the padded message as N 512-bit message blocks $M^1, M^2, \dots, M^N$.	Preprocessing: • Append the bit '1' to the message. • Append k bits '0', where k≥0 is the smallest integer satisfying Len+k = 896 (mod 1024), and Len is message's bit-length. • Append the vale Len, represented as a 128-bit integer in Big-Endian notation. • Set the initial hash value to $H_0^0 = 0x6a09e667f3bcc908; H_1^0 = 0xbb67ae8584caa73b;$ $H_2^0 = 0x3c6ef372fe94f82b; H_3^0 = 0xa54ff53a5f1d36f1;$ $H_4^0 = 0x510e527fade682d1; H_5^0 = 0x9b05688c2b3e6c1f;$ $H_6^0 = 0x1f83d9abfb41bd6b; H_7^0 = 0x5be0cd19137e2179.$ • Parse the padded message as N 1024-bit message blocks $M^1, M^2, \dots, M^N$.

Fig. 1. SHA-256 and SHA-512 constants, functions and preprocessing/padding.

i (=1, 2, …, N) is the serial number of the processed block.
nr = 63 for SHA-256 and 79 for SHA-512

Message scheduling (computing $\{W_t\}$):

$$W_t = \begin{cases} big\_endian(M_t^i) & 0 \le t \le 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_o(W_{t-15}) + W_{t-16} & 16 \le t \le nr \end{cases}$$

Set the eight working variables, a, b, c, d, e, f, g, h, to:

$a = H_0^{i-1}; b = H_1^{i-1}; c = H_2^{i-1}; d = H_3^{i-1},$
$e = H_4^{i-1}; f = H_5^{i-1}; g = H_6^{i-1}; h = H_7^{i-1}$

Compress:

For t = 0 to nr:

$T_1 = h + \Sigma_1(e) + Ch(e,f,g) + \mathrm{K_t} + W_t$
$T_2 = \Sigma_0(a) + Maj(a,b,c)$
$h = g; g = f; f = e; e = d + T_1; d = c; c = b; b = a; a = T_1 + T_2$
End for

Compute the i^{th}-step intermediate hash value H^i:

$H_o^i = a + H_0^{i-1}; H_1^i = b + H_1^{i-1}; H_2^i = c + H_2^{i-1}; H_3^i = d + H_3^{i-1};$
$H_4^i = e + H_4^{i-1}; H_5^i = f + H_5^{i-1}; H_6^i = g + H_6^{i-1}; H_7^i = h + H_7^{i-1}$

Fig. 2. SHA-256 and SHA-512 compression functions ("Update").

For example, we measured the OpenSSL (version 1.0.0e) implementation on the latest 2nd Generation Intel® Core™ processor. We found that the message scheduling step consumes ~*26*% of the computation time for SHA-1, ~*27*% for SHA-256, and ~*29*% for SHA-512. Obviously, optimizing the message scheduling in these hash algorithms can have a significant effect on the overall performance. We propose to optimize it via parallelization.

The idea behind our proposed *n*-Simultaneous Message Scheduling (*n*-SMS) method is to utilize the registers' width of a given SIMD architecture in order to compute, in parallel (and independently of the compression step), as many message schedules as possible. In general, we choose the parameter n by

$$n = \frac{supported\ SIMD\ registers\ width}{hash\ function\ word\ size}$$

For SHA-256, the compression function operates on *k=32*-bit elements. When running on the SSE architecture that supports *r=128*-bit registers, we use *n=r/k=4*. On the AVX2 architecture that supports *r=256*-bit registers (and integer SIMD operations), we use *n=r/k=8*. Similarly, for SHA-512 we used *n=2* on the SSE architecture, and *n=4* on the AVX2 architecture.

Finally, we mention that additional other steps of the algorithms can also be parallelized. One example is the step $K_t + W_t$ of both SHA-256 and SHA-512 (see Fig. 2).

3.2 Applying *4*-SMS to SHA-256: a detailed example

We show here the application of the *4*-SMS to SHA-256, allowing the computation of four message schedules, used for four *64*-byte blocks of the same message. When

hashing a buffer, *4*-SMS can be applied as long as the remaining portion of the buffer contains at least *256* bytes to be processed (when the remaining portion is shorter, we revert to the standard serial computations).

The constants, used for *4*-SMS SHA-256:

The *4*-SMS SHA-256 uses sixty-four *128*-bit constants, $K`^{256}_0, K`^{256}_1, \ldots, K`^{256}_{63}$

$K`^{256}_i = K^{256}_i \parallel K^{256}_i \parallel K^{256}_i \parallel K^{256}_i; 0 \le i \le 63$

(i.e., the SHA-256 constants concatenated four times).

The promoted SHA-256 functions:

The functions σ_0, σ_1, ROTR and SHR are "promoted" to a *128*-bit vectorized version as follows: Let X` = $x_3||x_2||x_1||x_0$, and Y` = $y_3||y_2||y_1||y_0$, be *128*-bit values, consisting of four *32*-bit dwords (x_0, x_1, x_2, x_3), (y_0, y_1, y_2, y_3.). We define:

$$SHR`^n(X`) = SHR^n(x_3) \parallel SHR^n(x_2) \parallel SHR^n(x_1) \parallel SHR^n(x_0)$$
$$ROTR`^n(X`) = ROTR^n(x_3) \parallel ROTR^n(x_2) \parallel RORT^n(x_1) \parallel ROTR^n(x_0)$$
$$\sigma`^{256}_0(X`) = ROTR`^7(X`) \oplus ROTR`^{18}(X`) \oplus SHR`^3(X`)$$
$$\sigma`^{256}_1(X`) = ROTR`^{17}(X`) \oplus ROTR`^{19}(X`) \oplus SHR`^{10}(X`)$$
$$X` + Y` = x_3 + y_3 \parallel x_2 + y_2 \parallel x_1 + y_1 || x_o + y_o$$

With these notations, we modify the SHA-256 computations, as follows.

The *4*-SMS Update function:

Prepare the "quadruped" message schedule, {W`$_t$}:

$$W`_t = \begin{cases} M^{(i-1)\times 4+4}_t \parallel M^{(i-1)\times 4+3}_t \parallel M^{(i-1)\times 4+2}_t \parallel M^{(i-1)\times 4+1}_t & 0 \le t \le 15 \\ \sigma`^{256}_1(W`_{t-2}) + W`_{t-7} + \sigma`^{256}_0(W`_{t-15}) + W`_{t-16} & 16 \le t \le 63 \end{cases}$$

Add the constants:

$W`_t = W`_t + `K`^{256}_t; 0 \le t \le 63$

Compress:

For j = 0 to 3:

Set the eight working variables, a, b, c, d, e, f, g, h, to:

$$a = H^{(i-1)\times 4+j}_0; b = H^{(i-1)\times 4+j}_1;$$
$$c = H^{(i-1)\times 4+j}_2; d = H^{(i-1)\times 4+j}_3;$$
$$e = H^{(i-1)\times 4+j}_4; f = H^{(i-1)\times 4+j}_5;$$
$$g = H^{(i-1)\times 4+j}_6; h = H^{(i-1)\times 4+j}_7$$

For t = 0 to 63:

$$T_1 = h + \Sigma^{256}_1(e) + Ch(e,f,g) + W`^j_t$$
$$T_2 = \Sigma^{256}_0(a) + Maj(a,b,c)$$
$$h = g; g = f; f = e; e = d + T_1;$$
$$d = c; c = b; b = a; a = T_1 + T_2;$$

End for

Compute the $[(i-1)\times 4+j+1]^{th}$ intermediate hash value H^i:

$$H_0^{[(i-1)\times 4+j+1]} = a + H_0^{(i-1)\times 4+j};$$
$$H_1^{[(i-1)\times 4+j+1]} = b + H_1^{(i-1)\times 4+j};$$
$$H_2^{[(i-1)\times 4+j+1]} = c + H_2^{(i-1)\times 4+j};$$
$$H_3^{[(i-1)\times 4+j+1]} = d + H_3^{(i-1)\times 4+j};$$
$$H_4^{[(i-1)\times 4+j+1]} = e + H_4^{(i-1)\times 4+j};$$
$$H_5^{[(i-1)\times 4+j+1]} = f + H_5^{(i-1)\times 4+j};$$
$$H_6^{[(i-1)\times 4+j+1]} = g + H_6^{(i-1)\times 4+j};$$
$$H_7^{[(i-1)\times 4+j+1]} = h + H_7^{(i-1)\times 4+j}$$

End for

This arrangement can readily use SIMD vector instructions. If N is the number of message blocks in the padded message, and $N = 4 \times floor(N/4) + r$, with $r = 0, 1, 2, 3$, we perform the *4*-SMS "Update" on the first *floor*(N/4) quadrupled blocks, and (if r≠0) process the remaining r blocks in the standard (serial) way, using the regular "Update" function. Alternatively it is possible to use *3*-SMS or *2*-SMS functions. After all the blocks are processed, the resulting *256*-bit message digest is $H_0 \parallel H_1 \parallel H_2 \parallel H_3 \parallel H_4 \parallel H_5 \parallel H_6 \parallel H_7$.

4 Software implementations

This section describes the software implementation of our proposed method.

To apply the *4*-SMS method to SHA-256, we used the following SSE4 instructions (see [5]) instructions:

- *PSRLD* (in order to compute the *SHR* function)
- *PSLLD*, *PSRLD* and *PXOR* (in order to compute *ROTR*)
- *PADDD* (to compute "+") and *PXOR* (to compute "⊕")
- *PINSRD* (to prepare the first *16* values of the message schedule)

When using the AVX architecture [5], it is also possible to use the non-destructive destination variants of these instructions, namely *VPSRLD*, *VPRLLD*, *VPXOR*, *VPADDD*, *VPINSRD* (which saves many "move" operations).

Fig. 8 of the Appendix provides a code snippet example for writing *4*-SMS SHA-256 message scheduling using SSE/AVX C intrinsic (to compile, note that the Intel Compiler (icc) uses AVX instructions (instead of regular SSE), if the compilation flag '-xAVX' is used).

For performance code, we also point out that the discussed code implementations were contributed to the open source community (as an OpenSSL 1.0.1 patch), and are fully available to the readers from [4]. This patch includes assembly language AVX and AVX2 versions of both SHA-256 and SHA-512 (written in the OpenSSL style as a perl-asm script).

4.1 SHA-256 and SHA-512 using AVX2 instructions

AVX2 architecture [8] extends the above instructions, straightforwardly, to operate on twice as many elements (stored in *256*-bit registers) in a single instruction. For example, AVX architecture has the instruction:

```
vpaddd xmm1, xmm2, xmm3
```

for adding four *32*-bit elements in xmm2 and xmm3, storing the results in (the four elements of) xmm1. The AVX2 architecture straightforwardly promotes the instruction to operate on eight elements, as follows:

```
vpaddd ymm1, ymm2, ymm3
```

Fig. 9 of the Appendix shows and example of *4*-SMS SHA-512 message scheduling implementation using AVX2 C intrinsics.

5 Dealing with short messages

The optimization that underlies our *n*-SMS method requires that the length of the hashed message is at least two blocks: *128* bytes for SHA-256 and *256* bytes for SHA-512 (i.e., *32* "words" of *32* bits for SHA-256 and *32* "words" of *64* bits for SHA-512).

Obviously, the *n*-SMS method achieves the maximal gain the number of blocks in the message is a multiple of the number of words that fit in a SIMD register (of the particular architecture we want to optimize on). For example, a message of *256* bytes has *4* SHA-256 blocks (or *2* SHA-512 blocks) that can fit into xmm. For the "remainder blocks", it may be preferable to hash using the standard ALU based computations. For example, a message of *320* bytes has *5* blocks and may be hashed by a call to *4*-SMS followed by a call to serial Update.

For long messages the cost of finishing a single block is relatively small. However, this is not the case for short messages. For example, a message of *64* bytes has a single block but requires two Updates – one for the message block, and another one for the padding block. Therefore, we suggest an optimization for specific short sizes, which saves the expansion of the last padding block altogether. Our optimization works for the following cases:

Case 1: the length of the hashed message is an integer multiple of the block size. In this case, the padding block is the bit *1*, followed by *447* zero bits, concatenated with the *64*-bit number that represents the length of the message in bits (in Big Endian notation).

Case 2: the last block message is longer than 55 bytes for SHA-256, or longer than 111 bytes for SHA-512. In this case, the padding spans across two blocks, and the last of them can be pre-expanded.

We give one example of SHA-256 padding block for a *64*-byte message:

Example 1: A pre-expanded padding block for SHA-256 when the message length is *64*-byte is the following sequence of bits: $1||0^{447+54}||1000000000$ (where 0^{447+54} indicates a string of *447+54=501* zero bits, and || denotes concatenation).

This optimization is useful for applications that hash many short buffers, because the padding blocks for the relevant sizes can be pre-expanded.

6 Results

To asses our proposed algorithm, we took the OpenSSL 1.0.1 [13] optimized (assembly) implementation as a reference point, where, at the first step, we further optimized it by replacing all the *ror* instructions with *shrd* instructions (this optimizes the code for the 2nd Generation Intel® Core™ Processors architectures; see [2] for more details). We applied the *4*-SMS method (for SHA-256) and the *2*-SMS method (for SHA-512) to this code (leaving the remaining pieces of the code untouched, except for the replacement mentioned above). This helps isolating the contribution of the *n*-SMS method from the impact of other possible optimizations (of the "encryption" portion of the Update function). The code was written in x64 assembly language (using OpenSSL perl-asm style) and is available as an OpenSSL 1.0.1 patch [4].

The experiments were carried out on three processors: Core™ 2 Duo T9400, Core™ i7-2600k (codename "Sandy Bridge"), Core™ i7-3770K (codename "Ivy Bridge"). The performance was measured in two ways:

1. Each measured function was isolated, run 25,000 times (warm-up), followed by 100,000 iterations that were clocked (using the RDTSC instruction) and averaged. To minimize the effect of background tasks running on the system, each experiment was repeated five times, and the minimum result was recorded. We used an 8KB buffer for these measurements. The reported performance numbers in Figures 3, 6 and 7 were obtained with the same measurement methodology.
2. We used OpenSSL's built-in utility "openssl speed sha256". This utility computes SHA-256 on buffers of various sizes, for *3* seconds, and reports the performance in *1000*s bytes per second hashed. Such measurements appear in Fig. 4.

All the runs were carried out on a system where the Intel® Turbo Boost Technology, the Intel® Hyper-Threading Technology, and the Enhanced Intel Speedstep® Technology, were *disabled*.

6.1 SSE/AVX results on Core 2 Duo, 2nd and 3rd Generation Intel® Core™ Processors

Fig. 3 shows SHA-256 performance using the *4*-SMS method, for multiple buffer lengths, compared to the OpenSSL 1.0.1 implementation [13]. The results are reported in CPU Cycles/Byte.

The performance advantage of the *4*-SMS method is apparent: it achieves significant improvement across all three architectures. The most significant

improvement is achieved on 2nd Generation Core: up to *31*% improvement compared to the original OpenSSL 1.0.1 (note that some of this improvement is due to the usage of the *shrd* instruction for rotation). On Core 2, we gain up to *8*% and on the new 3rd Generation Core the improvement is up to *17*%.

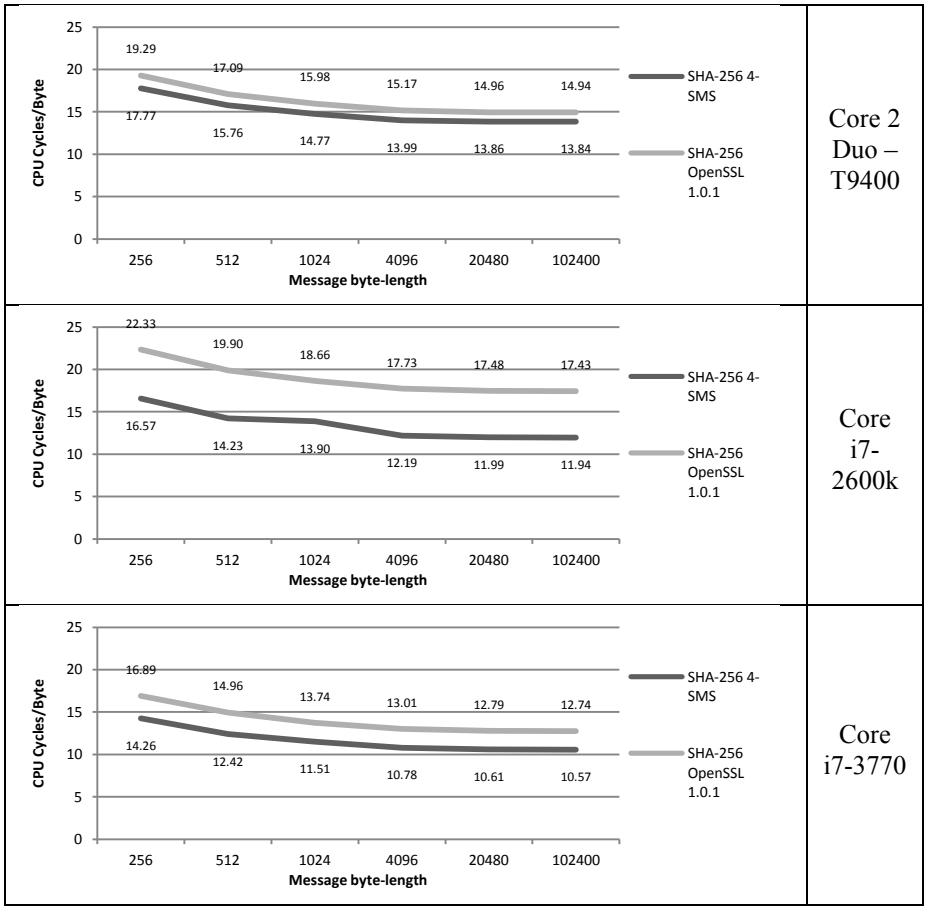

Fig. 3. Performance of SHA-256 on different processors and various message lengths.

Fig. 4 shows the performance of SHA-256 as reported by the 'openssl speed sha256' utility. As can be seen from the graphs the *4*-SMS method provides significant speedup, consistent with the results of Fig. 3. Note that rotation via the *shrd* instruction, which is significant in the 2nd Generation Core™ Processors, is not necessary (though does not slow down) on the 3rd Generation Core™ Processors, due to the micro-architectural improvements of the latter processor.

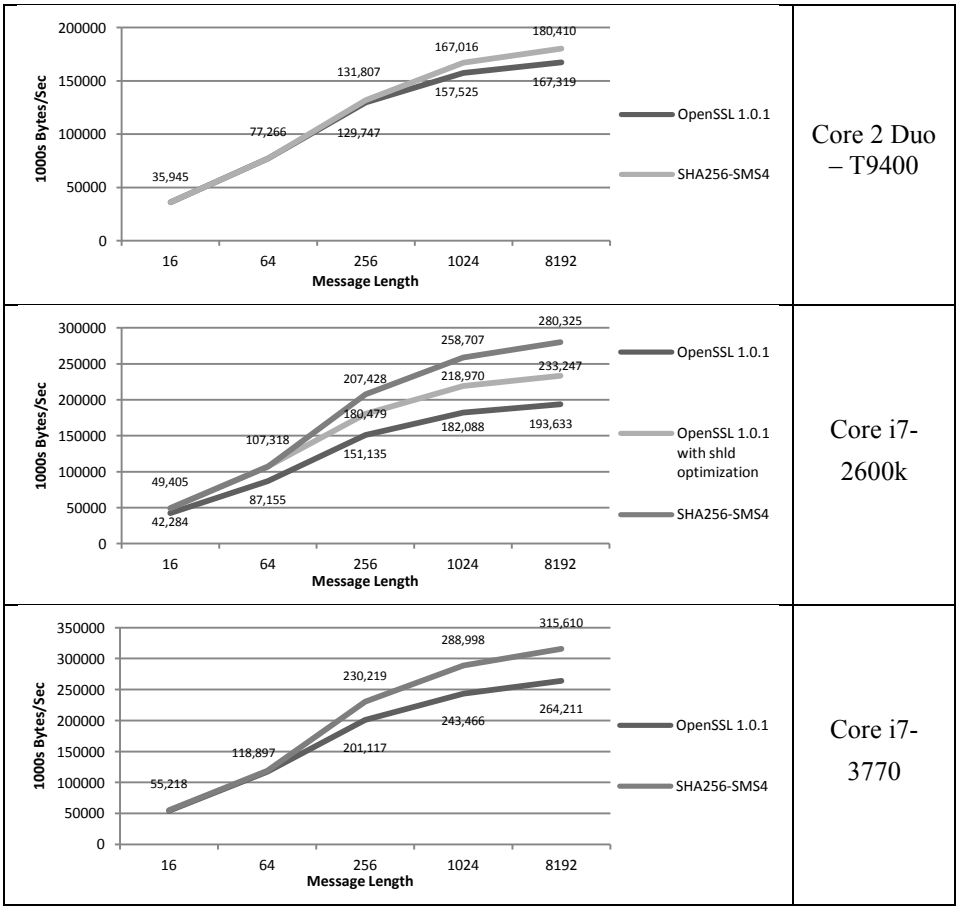

Fig. 4. Performance of SHA-256 as reported by the "openssl speed sha256" utility. Three variations where measured: 1) the original OpenSSL 1.0.1 implementation; 2) OpenSSL 1.0.1 with 4-SMS optimization [4]; 3) an optimized OpenSSL 1.0.1 implementation, obtained from using 'shrd' for rotation (instead of 'ror'; see [2] for details)–relevant only for the 2nd Generation Core™. The performance is reported in 1000s bytes hashed per second (higher is better)

6.2 AVX2 results

For AVX2, code can already be written, compiled (see [15] and tested for correctness (see [6] and [7]). However, since a processor that runs the AVX2 instructions is not yet available, we use a different method for demonstrating the performance benefit of our method. We compare the number of instructions required by the current and by the proposed implementations of an Update on *512* bytes (8 calls for OpenSSL; *2* for *4*-SMS and *1* for *8*-SMS). The number of instructions was counted using the histogram option of the SDE tool [6], generated with the "–mix" flag.

Fig. 5 shows the results, indicating AVX2 would potentially provide even greater speedup, by reducing the instruction count by a considerable amount. For SHA-512, the *4*-SMS saves *~21*% of the total number of instructions. For SHA-256, the *8*-SMS saves *~27*% of the total number of instructions. Of course, this cannot be linearly correlated to cycles count (e.g., because modern processors are "out-of-order" machines), and are only an indication for the expected speedup.

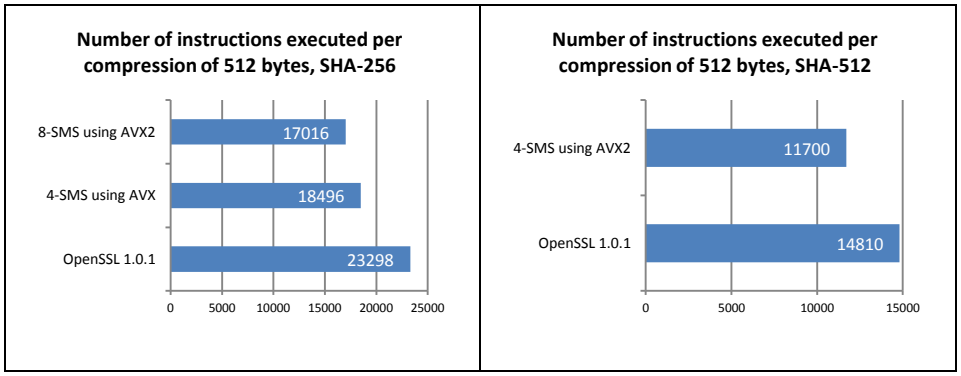

Fig. 5. Instruction count per compression of a 512 byte message block for different implementations

6.3 Results for small buffers using pre-expanded message schedule

Fig. 6 demonstrates the benefit of pre-expanding a message schedule for the final block, for short messages. In fact, even for messages a relatively large *2*KB message, this optimization gain *~2*% improvement. Note that for a short message of *64* bytes, there are only *2* calls to the Update function (i.e., only two blocks are processed), and the *~30*% saving obtained from pre-expanding is significant.

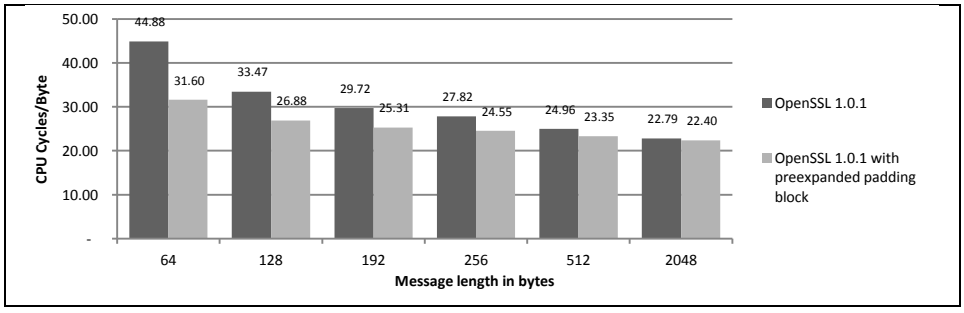

Fig. 6. Performance of OpenSSL 1.0.1 on a Core i7-3770 for short buffers, with and without the pre-expanded message schedule optimization, for the final block.

Such optimization is very useful for applications that hash many short messages of fixed lengths.

7 Conclusion

We applied the *n*-SMS method (with appropriate value of *n*) to SHA-1, SHA-256 and SHA-512 functions. For SHA-1, we could not gain performance by using the *4*-SMS method (apparently, the SHA-1 message scheduling is too simple). The *2*-SMS method improved the results by only *3.2*% on a Core i5-3770 for SHA-512, and causes degradation on Core 2 T9400. The reason is that the associated overhead outweighs the gain from parallelizing only two schedules. This means that the *n*-SMS method does not achieve significant gain for SHA-1 and for SHA-512 on the current AVX/SSE architectures.

For SHA-256, we showed that our *4*-SMS improves up to *31*% compared to the best existing public implementation (OpenSSL 1.0.1), and indicated why it is expected to improve even more, by taking advantage of the future AVX2 architecture, and applying the *8*-SMS method.

In addition we showed that in some instances the message scheduling for the last block of the padded message can be calculated in advance, therefore eliminating the need to perform any scheduling at all for that block. We applied this technique to OpenSSL's 1.0.1 implementation of SHA-256, and got significant speedup. We decided against including this optimization in [4] due to the complex branching involved to support many message lengths. However a system that needs to support only few fixed message lengths is surly to benefit greatly from such implementation.

It is interesting to see that the results of our proposed optimization are quite close to the theoretically achievable ones. Our initial measurements showed that the message scheduling in SHA-256 consumes *~25*% of the computations. Therefore, roughly speaking, the potential performance benefit of parallelizing four message schedules is at most *18.75*% of the total computations. Note that the fastest SHA-256 reference code on the Core i7-3770 performs at *12.74* Cycles/Byte, so we can expect our *4*-SMS method to reduce it to *10.35* Cycles/Byte at best. Indeed, the performance we report above is rather close, namely *10.57* Cycles/Byte. We estimate that the remaining gap is mainly due to two factors a) Rotation with the AVX instructions has to be implemented by two shifts and a xor instructions (compared to a single "rotate" ALU instruction); b) There is some overhead associated with "gathering" separate blocks of the message into the AVX registers (using the instruction *VPINSRD*).

Obviously, additional optimizations to the "encryption" part of the Update (for the discussed algorithms and for any other Davies-Meyer hash), can be incorporated together with the *n*-SMS method, and add further performance improvements.

7.1 Reflections on the SHA3 competition

To conclude, we comment on some implication of our results, to the SHA3 competition, which, at the time that the paper was written, is in its final stage.

Fig. 7 shows the performance of SHA-256 and SHA-512 – the OpenSSL 1.0.1 implementation implementations, as well as our *n*-SMS method. On the same graph, we plot the performance of the five SHA3 finalists. The implementations of the SHA3 finalists where obtained from [14], and re-measured under the same conditions and with the same methodology that was explained above, for complete consistency (see the Appendix for further details). These are the best known implementations to date

(that we aware of). All the numbers in Fig. 7 are measured for *8*KB buffers, and reported in Cycles/Byte.

Originally, the comparison baseline for the SHA competition was set (by NIST) to *~18* Cycles/Byte, compatible with the performance of SHA-256 on Core 2 , which was the high end processor at the announcement time. The rationale was that a significant performance gain is required from a successful SHA3, so that this advantage would encourage adaptation of the new standard. Indeed, all of the five SHA3 finalists satisfy this requirement.

However, the situation changes with our improved SHA-256 performance results, moving the comparison baseline to *13.84* Cycles/Byte on Core 2 Duo processors, and to an even lower baseline of *10.77* Cycles/Byte on the latest generation processors.

Moreover, we showed here that additional significant performance improvement is expected with the coming processor generation (performance numbers on real silicon will be available in 2013), and this will aggravate the situation.

In addition, with the standardized SHA-512 truncation [12] (and suggested in [3]), the performance of SHA-512 is also a viable comparison baseline – using exactly the same rationale. The performance of SHA-512 is already standing at *8.04* Cycles/Byte on today's processors (see Fig. 7), and the results we presented here, indicate that with the emergence of the new processors generation (in 2013), this performance will be further significantly improved.

This information implies that three of the five candidates, namely Grøstl, JH, and Keccak are already below the minimum requirements bar!

Based on our findings, we conclude that if the prospective SHA3 is expected to be competitive against the performance of SHA-256 and/or SHA-512, on the high end platforms, as the competition's definition mandates, its performance should be well below *8* Cycles/Byte on the current, and most certainly on the near future processors. Only two SHA3 candidates - Blake and Skein512 - meet these requirements, and it now remains to review their performance. With the old comparison baseline (*~18* Cycles/Byte), these algorithms could offer a speedup factor of *~2.5x* (and even more). However, with the baseline being changed by the *n*-SMS method, they offer a marginal performance boost (on the high end platforms). We point out that the coming AVX2 architecture (2013) will reduce this already small margin even further.

This is quite an unfortunate situation for NIST's SHA3 competition (especially because several fast proposed algorithms were eliminated in the first rounds of the competition). It suggests that no matter what algorithm is eventually selected to be the SHA3, its adoption should be motivated by considerations that are *not* based on performance on high end platforms. It also suggests that the SHA3 candidates could highly benefit from improvements in their software implementations.

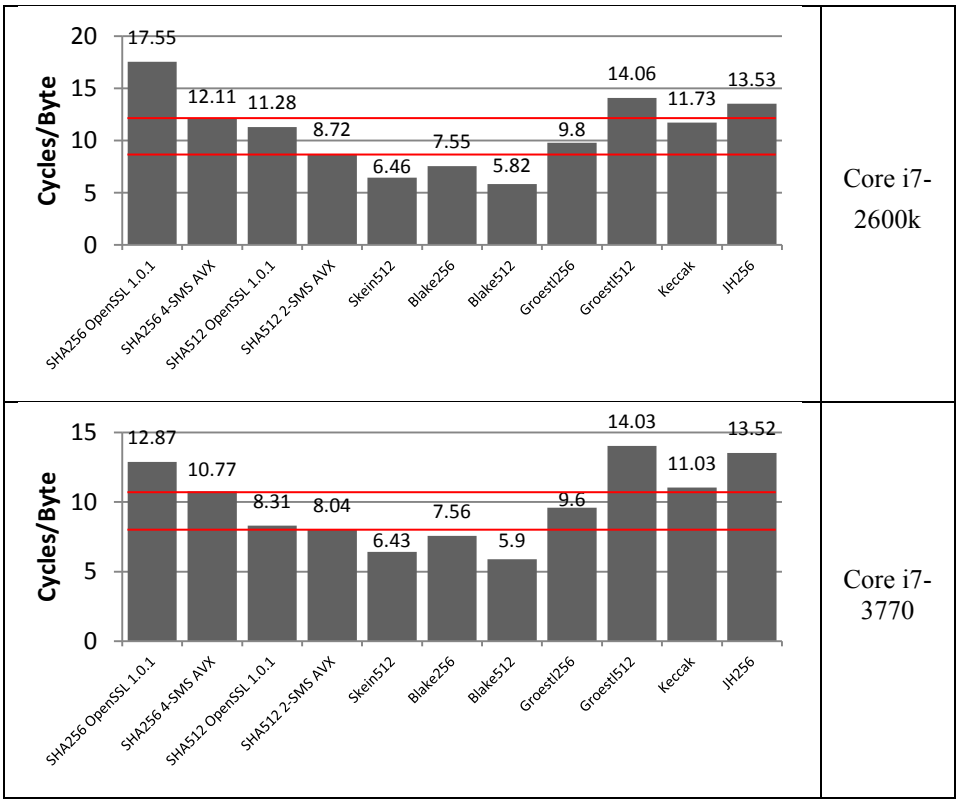

Fig. 7. Performance of SHA-256, SHA-512 and the five SHA3 finalists on the 2nd and the 3rd Generation Intel® Core™ Processors. See details in the text. The horizontal lines show the new "performance bar" of SHA-256 and SHA-512 with the n-SMS method.

8 References

1. Federal Information Processing Standards Publication 180-2: Secure Hash Standard. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
2. Gueron, S.: Speeding up SHA-1, SHA-256 and SHA-512 on the 2nd Generation Intel® Core™ Processors (to be published; ITNG 2012)
3. Gueron, S., Johnson, S., Walker, J.: SHA-512/256. IEEE Proceedings of 8th International Conference on Information Technology: New Generations (ITNG 2011), 354-358 (2011).
4. Gueron, S., Krasnov, V.: [PATCH] Efficient implementations of SHA256 and SHA512, using the Simultaneous Message Scheduling method, http://rt.openssl.org/Ticket/Display.html?id=2784&user=guest&pass=guest
5. Intel: Intel Advanced Vector Extensions Programming Reference. http://software.intel.com/file/36945
6. Intel: Software Development Emulator (SDE). http://software.intel.com/en-us/articles/intel-software-development-emulator/
7. Intel: Intel® Compilers. http://software.intel.com/en-us/articles/intel-compilers/

8. Intel (M. Buxton): Haswell New Instruction Descriptions Now Available! http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/
9. Kounavis, M.E., Kang, X., Grewal, K., Eszenyi, M., Gueron, S., Durham, D.: Encrypting the internet. In: Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM http://portal.acm.org/citation.cfm?id=1851182.1851200
10. Menezes, A.J., van Oorschot P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, 5th printing (2001).
11. NIST, cryptographic hash Algorithm Competition. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html
12. NIST: Secure Hash Standard. Draft Federal Information Processing Standards Publication 180-184 (2011).
13. OpenSSL, The Open Source toolkit for SSL/TLS, http://openssl.org/
14. SUPERCOP, http://bench.cr.yp.to/supercop.html
15. YASM, The YASM Modular Assembler Project. http://yasm.tortall.net/

9 Appendix

9.1 Code snippets

This appendix contains two C code examples. The first one implements *4*-SMS for SHA-256, using SSE intrinsics, and the second one implements SHA-512 using AVX2 intrinsics. This code only illustrates the discussed method. The performance code is written in assembly.

```
// this function takes a word from each chunk, and puts it in a single register
inline __m128i gather(unsigned int *address)
{
   __m128i temp;
   temp = _mm_cvtsi32_si128(address[0]);
   temp = _mm_insert_epi32(temp, address[16], 1);
   temp = _mm_insert_epi32(temp, address[32], 2);
   temp = _mm_insert_epi32(temp, address[48], 3);
   return temp;
}
// this function calculates the small sigma 0 transformation
inline __m128i sigma_0(__m128i W)
{
   return
      _mm_xor_si128(
         _mm_xor_si128(
            _mm_xor_si128(
               _mm_srli_epi32(W, 7),
               _mm_srli_epi32(W, 18)
               ),
            _mm_xor_si128(
               _mm_srli_epi32(W, 3),
               _mm_slli_epi32(W, 25)
               )
            ),
         _mm_slli_epi32(W, 14)
         );
}
// this function calculates the small sigma 1 transformation
inline __m128i sigma_1(__m128i W)
{
```

```
   return
      _mm_xor_si128(
         _mm_xor_si128(
            _mm_xor_si128(
               _mm_srli_epi32(W, 17),
               _mm_srli_epi32(W, 10)
               ),
            _mm_xor_si128(
               _mm_srli_epi32(W, 19),
               _mm_slli_epi32(W, 15)
               )
            ),
         _mm_slli_epi32(W, 13)
         );
}
// the message scheduling round
#define SCHEDULE_ROUND(w1, w2, w3, w4) \
            s0 = sigma_0(w1); \
            s1 = sigma_1(w2); \
            schedule[i] = _mm_add_epi32(w3, Ki[i]); \
            w3 = _mm_add_epi32( \
                    _mm_add_epi32(w3, w4), \
                    _mm_add_epi32(s0, s1) \
                    ); \
            i++;
void SHA256_QMS(__m128i schedule[64], uint32_t message[64])
{
   __m128i bswap_mask =
            _mm_set_epi8(12,13,14,15,8,9,10,11,4,5,6,7,0,1,2,3);
   __m128i W0, W1, W2, W3, W4, W5, W6, W7, W8, W9, W10, W11, W12, W13, W14, W15;
   __m128i s0, s1, Wi, *Ki = (__m128i*)k;
   int i;
   W0 = gather(message);
   W1 = gather(&message[1]);
   W2 = gather(&message[2]);
   W3 = gather(&message[3]);
   W4 = gather(&message[4]);
   W5 = gather(&message[5]);
   W6 = gather(&message[6]);
   W7 = gather(&message[7]);
   W8 = gather(&message[8]);
   W9 = gather(&message[9]);
   W10 = gather(&message[10]);
   W11 = gather(&message[11]);
   W12 = gather(&message[12]);
   W13 = gather(&message[13]);
   W14 = gather(&message[14]);
   W15 = gather(&message[15]);
   W0 = _mm_shuffle_epi8(W0, bswap_mask);
   W1 = _mm_shuffle_epi8(W1, bswap_mask);
   W2 = _mm_shuffle_epi8(W2, bswap_mask);
   W3 = _mm_shuffle_epi8(W3, bswap_mask);
   W4 = _mm_shuffle_epi8(W4, bswap_mask);
   W5 = _mm_shuffle_epi8(W5, bswap_mask);
   W6 = _mm_shuffle_epi8(W6, bswap_mask);
   W7 = _mm_shuffle_epi8(W7, bswap_mask);
   W8 = _mm_shuffle_epi8(W8, bswap_mask);
   W9 = _mm_shuffle_epi8(W9, bswap_mask);
   W10 = _mm_shuffle_epi8(W10, bswap_mask);
   W11 = _mm_shuffle_epi8(W11, bswap_mask);
   W12 = _mm_shuffle_epi8(W12, bswap_mask);
   W13 = _mm_shuffle_epi8(W13, bswap_mask);
   W14 = _mm_shuffle_epi8(W14, bswap_mask);
   W15 = _mm_shuffle_epi8(W15, bswap_mask);
   for(i=0; i<32; ) {
      SCHEDULE_ROUND(W1 , W14, W0 , W9 );
      SCHEDULE_ROUND(W2 , W15, W1 , W10);
      SCHEDULE_ROUND(W3 , W0 , W2 , W11);
```

```
      SCHEDULE_ROUND(W4 , W1 , W3 , W12);
      SCHEDULE_ROUND(W5 , W2 , W4 , W13);
      SCHEDULE_ROUND(W6 , W3 , W5 , W14);
      SCHEDULE_ROUND(W7 , W4 , W6 , W15);
      SCHEDULE_ROUND(W8 , W5 , W7 , W0 );
      SCHEDULE_ROUND(W9 , W6 , W8 , W1 );
      SCHEDULE_ROUND(W10, W7 , W9 , W2 );
      SCHEDULE_ROUND(W11, W8 , W10, W3 );
      SCHEDULE_ROUND(W12, W9 , W11, W4 );
      SCHEDULE_ROUND(W13, W10, W12, W5 );
      SCHEDULE_ROUND(W14, W11, W13, W6 );
      SCHEDULE_ROUND(W15, W12, W14, W7 );
      SCHEDULE_ROUND(W0 , W13, W15, W8 );
   }
   SCHEDULE_ROUND(W1 , W14, W0 , W9 );
   schedule[48] = _mm_add_epi32(W0, Ki[48]);
   SCHEDULE_ROUND(W2 , W15, W1 , W10);
   schedule[49] = _mm_add_epi32(W1, Ki[49]);
   SCHEDULE_ROUND(W3 , W0 , W2 , W11);
   schedule[50] = _mm_add_epi32(W2, Ki[50]);
   SCHEDULE_ROUND(W4 , W1 , W3 , W12);
   schedule[51] = _mm_add_epi32(W3, Ki[51]);
   SCHEDULE_ROUND(W5 , W2 , W4 , W13);
   schedule[52] = _mm_add_epi32(W4, Ki[52]);
   SCHEDULE_ROUND(W6 , W3 , W5 , W14);
   schedule[53] = _mm_add_epi32(W5, Ki[53]);
   SCHEDULE_ROUND(W7 , W4 , W6 , W15);
   schedule[54] = _mm_add_epi32(W6, Ki[54]);
   SCHEDULE_ROUND(W8 , W5 , W7 , W0 );
   schedule[55] = _mm_add_epi32(W7, Ki[55]);
   SCHEDULE_ROUND(W9 , W6 , W8 , W1 );
   schedule[56] = _mm_add_epi32(W8, Ki[56]);
   SCHEDULE_ROUND(W10, W7 , W9 , W2 );
   schedule[57] = _mm_add_epi32(W9, Ki[57]);
   SCHEDULE_ROUND(W11, W8 , W10, W3 );
   schedule[58] = _mm_add_epi32(W10, Ki[58]);
   SCHEDULE_ROUND(W12, W9 , W11, W4 );
   schedule[59] = _mm_add_epi32(W11, Ki[59]);
   SCHEDULE_ROUND(W13, W10, W12, W5 );
   schedule[60] = _mm_add_epi32(W12, Ki[60]);
   SCHEDULE_ROUND(W14, W11, W13, W6 );
   schedule[61] = _mm_add_epi32(W13, Ki[61]);
   SCHEDULE_ROUND(W15, W12, W14, W7 );
   schedule[62] = _mm_add_epi32(W14, Ki[62]);
   SCHEDULE_ROUND(W0 , W13, W15, W8 );
   schedule[63] = _mm_add_epi32(W15, Ki[63]);
}
```

Fig. 8. 4-SMS message scheduling for SHA-256 using C intrinsics for the SSE3 instruction set.

```
#define vpbroadcastq(vec, k) vec = _mm256_broadcastq_epi64(*(__m128i*)k)
// this function calculates the small sigma 0 transformation
inline __m256i sigma_0(__m256i W)
{
   return
      _mm256_xor_si256(
         _mm256_xor_si256(
            _mm256_xor_si256(
               _mm256_srli_epi64(W, 7),
               _mm256_srli_epi64(W, 8)
               ),
            _mm256_xor_si256(
               _mm256_srli_epi64(W, 1),
               _mm256_slli_epi64(W, 56)
               )
            ),
```

```
         _mm256_slli_epi64(W, 63)
         );
}
// this function calculates the small sigma 1 transformation
inline __m256i sigma_1(__m256i W)
{
   return
      _mm256_xor_si256(
         _mm256_xor_si256(
            _mm256_xor_si256(
               _mm256_srli_epi64(W, 6),
               _mm256_srli_epi64(W, 61)
               ),
            _mm256_xor_si256(
               _mm256_srli_epi64(W, 19),
               _mm256_slli_epi64(W, 3)
               )
            ),
         _mm256_slli_epi64(W, 45)
         );
}
// the message scheduling round
#define SCHEDULE_ROUND(w1, w2, w3, w4) \
            vpbroadcastq(Ki, &k[i]);\
            s0 = sigma_0(w1); \
            s1 = sigma_1(w2); \
            schedule[i] = _mm256_add_epi64(w3, Ki); \
            w3 = _mm256_add_epi64( \
                    _mm256_add_epi64(w3, w4), \
                    _mm256_add_epi64(s0, s1) \
                    ); \
            i++;

void SHA512_QMS(__m256i schedule[80], uint64_t message[64])
{
   __m256i gather_mask = _mm256_setr_epi64x(0, 16, 32, 48);
   __m256i bswap_mask =
_mm256_set_epi8(8,9,10,11,12,13,14,15,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0,1,2,
3,4,5,6,7);
   __m256i W0, W1, W2, W3, W4, W5, W6, W7, W8, W9, W10, W11, W12, W13, W14, W15;
   __m256i s0, s1, Ki, Wi;
   int i;

   W0 = _mm256_i64gather_epi64((const long*)message, gather_mask, 8);
   W0 = _mm256_shuffle_epi8(W0, bswap_mask);
   W1 = _mm256_i64gather_epi64((const long*)&message[1], gather_mask, 8);
   W1 = _mm256_shuffle_epi8(W1, bswap_mask);
   W2 = _mm256_i64gather_epi64((const long*)&message[2], gather_mask, 8);
   W2 = _mm256_shuffle_epi8(W2, bswap_mask);
   W3 = _mm256_i64gather_epi64((const long*)&message[3], gather_mask, 8);
   W3 = _mm256_shuffle_epi8(W3, bswap_mask);
   W4 = _mm256_i64gather_epi64((const long*)&message[4], gather_mask, 8);
   W4 = _mm256_shuffle_epi8(W4, bswap_mask);
   W5 = _mm256_i64gather_epi64((const long*)&message[5], gather_mask, 8);
   W5 = _mm256_shuffle_epi8(W5, bswap_mask);
   W6 = _mm256_i64gather_epi64((const long*)&message[6], gather_mask, 8);
   W6 = _mm256_shuffle_epi8(W6, bswap_mask);
   W7 = _mm256_i64gather_epi64((const long*)&message[7], gather_mask, 8);
   W7 = _mm256_shuffle_epi8(W7, bswap_mask);
   W8 = _mm256_i64gather_epi64((const long*)&message[8], gather_mask, 8);
   W8 = _mm256_shuffle_epi8(W8, bswap_mask);
   W9 = _mm256_i64gather_epi64((const long*)&message[9], gather_mask, 8);
   W9 = _mm256_shuffle_epi8(W9, bswap_mask);
   W10 = _mm256_i64gather_epi64((const long*)&message[10], gather_mask, 8);
   W10 = _mm256_shuffle_epi8(W10, bswap_mask);
   W11 = _mm256_i64gather_epi64((const long*)&message[11], gather_mask, 8);
   W11 = _mm256_shuffle_epi8(W11, bswap_mask);
   W12 = _mm256_i64gather_epi64((const long*)&message[12], gather_mask, 8);
```

```
   W12 = _mm256_shuffle_epi8(W12, bswap_mask);
   W13 = _mm256_i64gather_epi64((const long*)&message[13], gather_mask, 8);
   W13 = _mm256_shuffle_epi8(W13, bswap_mask);
   W14 = _mm256_i64gather_epi64((const long*)&message[14], gather_mask, 8);
   W14 = _mm256_shuffle_epi8(W14, bswap_mask);
   W15 = _mm256_i64gather_epi64((const long*)&message[15], gather_mask, 8);
   W15 = _mm256_shuffle_epi8(W15, bswap_mask);

for(i=0; i<64; )
{
   SCHEDULE_ROUND(W1, W14, W0, W9);
   SCHEDULE_ROUND(W2, W15, W1, W10);
   SCHEDULE_ROUND(W3, W0, W2, W11);
   SCHEDULE_ROUND(W4, W1, W3, W12);
   SCHEDULE_ROUND(W5, W2, W4, W13);
   SCHEDULE_ROUND(W6, W3, W5, W14);
   SCHEDULE_ROUND(W7, W4, W6, W15);
   SCHEDULE_ROUND(W8, W5, W7, W0);
   SCHEDULE_ROUND(W9, W6, W8, W1);
   SCHEDULE_ROUND(W10, W7, W9, W2);
   SCHEDULE_ROUND(W11, W8, W10, W3);
   SCHEDULE_ROUND(W12, W9, W11, W4);
   SCHEDULE_ROUND(W13, W10, W12, W5);
   SCHEDULE_ROUND(W14, W11, W13, W6);
   SCHEDULE_ROUND(W15, W12, W14, W7);
   SCHEDULE_ROUND(W0, W13, W15, W8);
}
schedule[64] = _mm256_add_epi64(W0, _mm256_broadcastq_epi64(*(__m128i*)&k[64]));
schedule[65] = _mm256_add_epi64(W1, _mm256_broadcastq_epi64(*(__m128i*)&k[65]));
schedule[66] = _mm256_add_epi64(W2, _mm256_broadcastq_epi64(*(__m128i*)&k[66]));
schedule[67] = _mm256_add_epi64(W3, _mm256_broadcastq_epi64(*(__m128i*)&k[67]));
schedule[68] = _mm256_add_epi64(W4, _mm256_broadcastq_epi64(*(__m128i*)&k[68]));
schedule[69] = _mm256_add_epi64(W5, _mm256_broadcastq_epi64(*(__m128i*)&k[69]));
schedule[70] = _mm256_add_epi64(W6, _mm256_broadcastq_epi64(*(__m128i*)&k[70]));
schedule[71] = _mm256_add_epi64(W7, _mm256_broadcastq_epi64(*(__m128i*)&k[71]));
schedule[72] = _mm256_add_epi64(W8, _mm256_broadcastq_epi64(*(__m128i*)&k[72]));
schedule[73] = _mm256_add_epi64(W9, _mm256_broadcastq_epi64(*(__m128i*)&k[73]));
schedule[74] = _mm256_add_epi64(W10, _mm256_broadcastq_epi64(*(__m128i*)&k[74]));
schedule[75] = _mm256_add_epi64(W11, _mm256_broadcastq_epi64(*(__m128i*)&k[75]));
schedule[76] = _mm256_add_epi64(W12, _mm256_broadcastq_epi64(*(__m128i*)&k[76]));
schedule[77] = _mm256_add_epi64(W13, _mm256_broadcastq_epi64(*(__m128i*)&k[77]));
schedule[78] = _mm256_add_epi64(W14, _mm256_broadcastq_epi64(*(__m128i*)&k[78]));
schedule[79] = _mm256_add_epi64(W15, _mm256_broadcastq_epi64(*(__m128i*)&k[79]));
}
```

Fig. 9. 4-SMS message scheduling for SHA-512 using C intrinsics for the AVX2 instruction set.

9.2 Fig. 7 - Sources

Fig. 7 presents performance numbers for several hash algorithms. To facilitate reproducing the results, we provide the following details.

The source codes for Blake, Grøstl, JH, Keccak, and Skein were retrieved from "supercop" [14], and re-measured using the methodology described in Section 6.

The supercop version we used was 20120329 (SUPERCOP hereafter). It can be downloaded from http://hyperelliptic.org/ebats/supercop-20120329.tar.bz2. More details on the sources, including the compilation flags (when relevant) are:

SHA-256 openssl: OpenSSL 1.0.1
SHA-512 openssl: OpenSSL 1.0.1
SHA-256 *4*-SMS: the code posted in [4], applied to OpenSSL 1.0.1

SHA-512 *2*-SMS: the code posted in [4], applied to OpenSSL 1.0.1
Skein: SUPERCOP, "sandy", compiled using: gcc -m64 -march=core2 -msse4.1 -Os -fomit-frame-pointer
Blake256 - SUPERCOP, "avxicc", assembler
Blake512 - SUPERCOP, "avxicc", assembler
Grøstl256 - SUPERCOP, "avx", compiled using: gcc -funroll-loops -march=nocona -O3 -fomit-frame-pointer -DTASM
Grøstl512 – SUPERCOP, "aesni", compiled using: gcc -funroll-loops -march=nocona -O3 -fomit-frame-pointer -DTASM
JH256 - SUPERCOP, "bitslice_sse2_opt64", compiled using: icc -O3 -xAVX
Keccak – SUPERCOP, "x86_64_shld", compiled using: gcc -funroll-loops -O3 -fomit-frame-pointer

Compilers: we used gcc version 4.5.1, and icc version 12.