Simultaneous hashing of multiple messages

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

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

July 3, 2012

Abstract. We describe a method for efficiently hashing multiple messages of different lengths. Such computations occur in various scenarios, and one of them is when an operating system checks the integrity of its components during boot time. These tasks can gain performance by parallelizing the computations and using SIMD architectures. For such scenarios, we compare the performance of a new *4*-buffers SHA-256 S-HASH implementation, to that of the standard serial hashing. Our results are measured on the 2nd Generation Intel® Core™ Processor, and demonstrate SHA-256 processing at effectively ~*5.2* Cycles per Byte, when hashing from any of the three cache levels, or from the system memory. This represents speedup by a factor of *3.42*x compared to OpenSSL (1.0.1), and by *2.25*x compared to the recent and faster *n*-SMS method. For hashing from a disk, we show an effective rate of ~*6.73* Cycles/Byte, which is almost *3* times faster than OpenSSL (1.0.1) under the same conditions. These results indicate that for some usage models, SHA-256 is significantly faster than commonly perceived.

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

1 Introduction

The performance of hash functions is important in various situations and platforms. One example is a server workload: authenticated encryption in SSL/TLS sessions, where hash functions are used for authentication, in HMAC mode. This is one reason why the performance of SHA-256 on modern x86_64 architectures was defined as a baseline for the SHA3 competition [13].

Traditionally, the performance of hash functions is measured by hashing a single message (of some length) on a target platform. For example, consider the 2nd Generation Intel® Core™ Processors. The OpenSSL (1.0.1) implementation hashes a single buffer (of length 8KB) at *17.55* Cycles per Byte (C/B hereafter). Recently, [5] improved the performance of SHA-256 with an algorithm that parallelizes the message schedule, and the use of SIMD architectures, moving the performance baseline to *11.58* C/B (code version from April 2012 is available from [6], and will be updated soon) on the modern processors, when hashing from the cache.

In this paper, we investigate the possibility of accelerating SHA-256 for some scenarios, and are interested in optimizing the following computation: hashing a number (*k*) of independent messages, to produce *k* different digests. We investigate the advantage of SIMD architectures for these parallelizable computations.

Such workloads appear, for example, during the boot process of an operating system, where it checks the integrity of its components (see [15] for example). This involves computing multiple hashes, and comparing them to expected values. Another situation that involves hashing of multiple independent messages is data de-duplication, where large amounts of data are scanned (typically in chunks of fixed sizes) in order to identify duplicates (see e.g., [2]). In these two scenarios, the data typically reside on the hard disk, but hashing multiple independent messages could also emerge in situations where the data is in the cache/memory.

A SIMD based implementation of hash algorithms was first proposed (in 2004) and described in detail by Aciiçmez [1]. He studied the computations of SHA-1, SHA-256 and SHA-512, and his investigation was carried out on Intel® Pentium™ 4, using SSE2 instructions. Two approaches for gaining performance were attempted: a) using SIMD instructions to parallelize some of the computations of the message schedule of these hash algorithms, when hashing a single message (see also later works (on SHA-1) along these lines, in [4] and [12]); b) using SIMD instructions to parallelize hash computations of several independent messages. Aciiçmez reports that he could not improve the performance of hashing a single buffer, using the SIMD instructions (while this could not be done on the Pentium 4, we speculate that it would be possible on more recent architectures). However, he reports speedup by a factor of *1.71*x for simultaneous hashing of four buffers, with SHA-256 (speedup by a factor of *2.3*x for SHA-512 is also reported, but it is less interesting in our context, because the comparison baseline was a (slow) *32*-bit implementation).

In this paper we expand the study conducted by Aciiçmez, by demonstrating the performance of Simultaneous Hashing of multiple independent messages, on contemporary processors. We detail a method for a "Simultaneous Update" that facilitates hashing of independent messages of arbitrary sizes. To account for different usages, we investigate the performance of hashing multiple messages (of variable sizes) from different cache hierarchies, system memory, and from the hard drive.

2 Preliminaries and notations

The detailed definition of SHA-256 can be found in FIPS180-2 publication [3]. Schematically, the computational flow of SHA-256 can be viewed as follows: "Init" (setting the initial values), a sequence of "Update" steps (compressing a *64* bytes bloc the message, and updating the digest value), and a "Finalize" step (takes care of the message padding). The padding requires either one or two calls to the Update function, depending on the message's length (see more details in [5]). For SHA-256, the performance is almost linearly proportional to the number (*N*) of Update function calls, which. For a message of *length* bytes, the value of *N* is:

$$\text{SHA-256: } N = \begin{cases} \left\lfloor \frac{length}{64} \right\rfloor + 2 & length \bmod 64 \geq 56 \\ \left\lfloor \frac{length}{64} \right\rfloor + 1 & else \end{cases}$$

For sufficiently long messages, we can approximate *N~floor (length/64)*. For example, this approximation for a *4*KB message gives *floor (length/64) = 64*, while actual hashing of a *4*KB message requires *65* Update function calls (i.e., a *~1.5*% deviation).

3 Simultaneous hashing (S-HASH) of multiple messages

SIMD architectures [7] are designed to execute, in parallel, the same operations on several independent chunks of data (called "elements"). Modern architectures have variants of SIMD instructions that operate on elements of sizes *1*, *2*, *4*, or *8* bytes. By the nature of the algorithms, SHA-256 (and SHA-1) requires operations on *4* bytes elements, while SHA-512 requires operations on *8* bytes elements.

Fig. 1 describes the Simultaneous Hashing algorithm (S-HASH) that hashes k messages and generates k digests, with some hash function. Suppose that the implemented hash function operates on t-bit "words" (elements), and that the architecture has s-bit SIMD registers. Then, the number of words that fit into a SIMD register is m = s/t, which we assume to be an integer. We also assume that $k > m$. Algorithm 1 starts with the Initialize step for the first m buffers. Then, it invokes the "Simultaneous Update" function (for the specific hash function) every time there are m blocks ready for processing. This is repeated until the shortest buffer (from the m processed buffers) is fully consumed. At this point, a padding block is fed to the Simultaneous Update function, to "Finalize" (that buffer). If the hash is already finalized, a block from a new buffer is fed (after the proper "Init").

Here, we use the AVX architecture [7], with *128*-bit registers (i.e., $s=128$). SHA-256 (and SHA-1) algorithms have $t=32$, while SHA-512 has $t=64$, implying $m=4$ for SHA-1 and SHA-256, and $m=2$ for SHA512. For our SHA-256 study, we can hash *4* buffers in parallel. We call this implementation *4*-buffers SHA-256 S-HASH.

The near-future AVX2 architecture [9] has integer instructions that operate on *256*-bit registers. This allows for doubling the number of independent messages that can be hashed in parallel and would lead to, for example, *8*-buffers SHA-256 S-HASH or *4*-buffers SHA-512 S-HASH.

```
Algorithm 1: Simultaneous Hashing (S-HASH)
Input:
Buffers – a list with pointers to k buffers to be hashed.
Lengths – a list with the lengths (in bytes) of the k buffers.
Hashes – a list with pointers to store the k generated hash values.
Notations:
The number of t-bit “words” (elements) that fit in a register is m. (for
SHA-256, t=32, and with AVX, m=128/32=4).
It is assumed that k > m.
The number of bytes, hashed by one “Update” operation is denoted by p.
Output: k hash values of the k buffers, stored the at memory locations
pointed by Hashes.
Flow:
Init:
        L[0] = Lengths[0]; L[1] = Lengths[1];… L[m-1] = Lengths[m-1]
        B[0] = Buffers[0]; B[1] = Buffers[1]; … B[m-1] = Buffers[m-1]
        H[0] = Hashes[0]; H[1] = Hashes[1]; … H[m-1] = Hashes[m-1]
        Last[0] = 0; Last[1] = 0; … Last[m-1] = 0
        HashInit(Hashes[0])
        HashInit(Hashes[1])
        …
        HashInit(Hashes[m-1])
        i = m;

Simultaneous Update:

Repeat
        n = min(L)/p
        S-UPDATE(H, B, n)
        L = L – [n×p|n×p|…|n×p]
        For j = 0 to m-1
                If L[j]<p AND Last[j]=0 then
                        LastBlock[j] = PreparePaddingBlock(B[j])
                        B[j] = LastBlock[j]
                        Last[j] = 1
                        L[j] = Length(LastBlock[j])
                Else If L[j]<p AND Last[j]=1 then
                        If i=k then
                                Break
                        Else
                                L[j] = Lengths[i]
                                B[j] = Buffs[i]
                                H[j] = Hashes[i]
                                Last[j] = 0
                                HashInit(Hashes[i])
                                i++
                        End If
                End If
        End For
End Repeat

If unfinished buffers still remain, finish hashing serially
```

Fig. 1. The Simultaneous Hashing (S-HASH) algorithm.

4 Results

This section describes the *4*-buffers SHA-256 S-HASH results

4.1 The system's characteristics

The system that was used for generating the reported measurements had the following characteristics:

- An Intel® Core™ i5-2500 processor (2nd Generation Intel® Core™ Processor; Sometimes referred to as Architecture Codename "Sandy Bridge")
- *8*GB RAM (DDR3 *1600*, *2* Channels).
- A RAID0 array of two Intel® SSD *320* drives, each one of *80*GB and combined throughput of *400*MB/sec (indicated by "hdparm –t" [10]).
- Fedora 16 OS.

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*.

4.2 Simultaneous hashing of multiple *4*KB buffers, from different cache levels and main memory

For profiling the performance of the *4*-buffers SHA-256 S-HASH, we wrote a new implementation which processes four buffers in parallel. In order to estimate the advantage of the parallelization, we compare the resulting performance to serial implementations that hash the same amount of data.

To measure the performance of hashing data that resides in different cache levels, or in memory, we note that the processor has ([8]): a) First Level Data Cache of *32*KB (per core); b) Second Level Cache of *256*KB (per core); c) Last Level Cache of *6*MB (shared among all the cores). Therefore,

- For data that resides in the First Level Cache, we hashed a total of *16*KB of data, split to *4* chunks of *4*KB each.
- For data that resides in the Second Level Cache, we hashed a total of *256*KB of data, split to *64* chunks of *4*KB each.
- For data that resides in the Last Level Cache, we hashed a total of *2*MB of data, split to *512* chunks of *4*KB each.
- For data that resides in the main memory, we hashed a total of 32MB of data, split to *8192* chunks of *4*KB each.

Prior to the actual measurements, we ran the hash, in a loop, *500* times, in order to make sure that our data resides in the desired cache level (or memory).

For comparison, we used the OpenSSL (version 1.0.1) SHA-256 (serial) [14] implementation, and the faster implementation, based on the *n*-SMS method [5] (a version from April 2012, can be retrieved from [6]; An update will be posted soon).

The results, illustrated in Fig. 2, show that hashing from all three cache levels can be performed at roughly the same performance, and there is only some small performance degradation when the data is hashed from the main memory. The *4*-

buffers SHA-256 S-HASH method is *3.42*x faster than OpenSSL (1.0.1), and *2.25*x times faster than the *n*-SMS method.

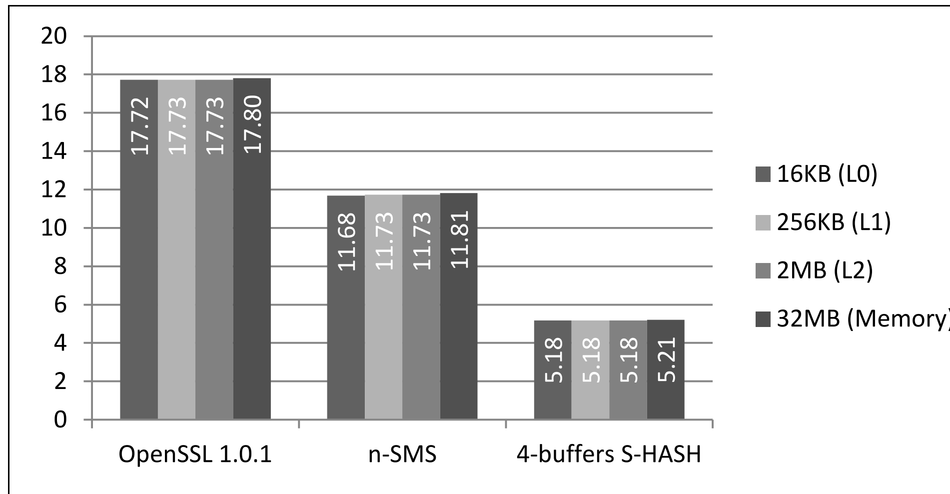

Fig. 2. SHA-256 hashing from different cache levels and memory, on the Intel® Core™ i5-2500 (Architecture Codename Sandy Bridge). The performance of the 4-buffers SHA-256 S-HASH is compared to the (standard) serial hashing with the OpenSSL 1.0.1 implementation, and to the *n*-SMS method (see explanation in the text).

4.3 Simultaneous hashing of files from the hard-drive

The following results account for the performance of hashing from the disk. The numbers were obtained using the following methodology.

For the experiments, we prepared two directories with a different combination of files. The first directory (DIVERSE hereafter) contained *350* files occupying *79*MB (*82,833,132* bytes) in total(1). The files sizes range from *3* Bytes to *7.18*MB (*7,533,568* bytes), with the average size of *0.22*MB (*236,666* bytes). The detailed size distribution of the file is provided in the Appendix. The second directory (UNIFORM hereafter) contained *8* (large) files of equal size, each one of *17.76*MB (*18,623,835* bytes)(2) For each directory, we prepared, in advance, the list of its files.

To measure the performance of hashing from the hard drive, we flushed the OS “pagecache” and “dentries” and “inodes” caches, before the measurements were taken (using the Linux directive `echo 3 > /proc/sys/vm/drop_caches`) [11].

We measured the following operations: scanning the list (in the prescribed order), opening the files in the list, reading the size of each file, mapping the files to memory, calculating the SHA-256 values and storing them in appropriate location.

(1) These files were the drivers from a Windows 7 directory “Windows\Systen32\drivers\”.
(2) The files were copies of the same file, namely “supercop-20120219.tar.gz”, retrieved from http://hyperelliptic.org/ebats/supercop-20120219.tar.bz2).

Fig. 3, top panel, provides the performance for the "DIVERSE" directory in C/B (which is a *frequency-agnostic* metric). The performance is shown for several processor frequencies, to demonstrate how the hard-drive's throughput limits the overall observed performance. The figure shows that at the native processor speed (*3.3*GHz), the S-HASH method outperforms the OpenSSL (1.0.1) implementation by a factor of 1.73x. When the processor is down-clocked to *1.6*GHz, all three implementations improve their C/B count, but the S-HASH improves by a larger margin, becoming *2.16*x faster than OpenSSL. The bottom panel of Fig. 3 shows the same performance, measured in MB/sec. It is interesting to observe that although the frequency of the processor is reduced by factor of two, from *3.3*GHz to *1.6*GHz, the S-HASH throughput reduces only by a factor of *1.28*x.

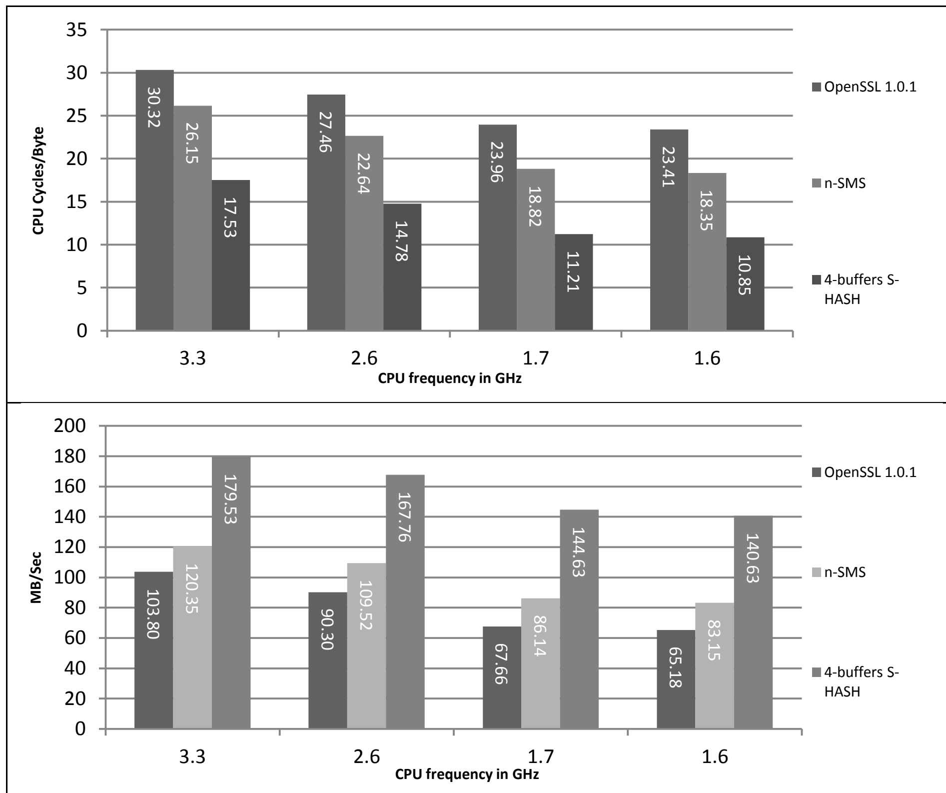

Fig. 3. Hashing the files in the directory DIVERSE (see explanation in the text). Measurements are taken on the Core i5-2500, operating at different CPU frequencies. Panel a shows the performance in Cycles per Byte. Panel b shows the performance in MB/sec.

Fig. 4 illustrates the performance for the UNIFORM directory. In this scenario, Tte performance of OpenSSL and of the *n*-SMS method are not limited by hard drive, because we see that reducing frequency does not improve the speed in C/B. On the other hand, the faster *4*-buffers SHA-256 S-HASH implementation is affected by the hard drives. It improves (in C/B) when the frequency is reduced, although not as

much as it does in the DIVERSE test. The figure shows that the *4*-buffers S-HASH is *2.86*x faster than OpenSSL, when the processor is clocked at *1.6*GHz, and *2.26*x faster at the native processor's frequency.

In general, all implementations improve when the hashed files are large. The reasons are that the overheads for opening files are reduced, and the reads from hard drive are sequential. In addition, the S-HASH is faster when the processed files have equal lengths (UNIFORM directory). This happens because the computations for all the four buffers terminate concurrently, allowing four new buffers to be scheduled together. By contrast, in the DIVERSE directory, when a certain buffer is consumed, operations on the remaining buffers are stopped until a new buffer is scheduled.

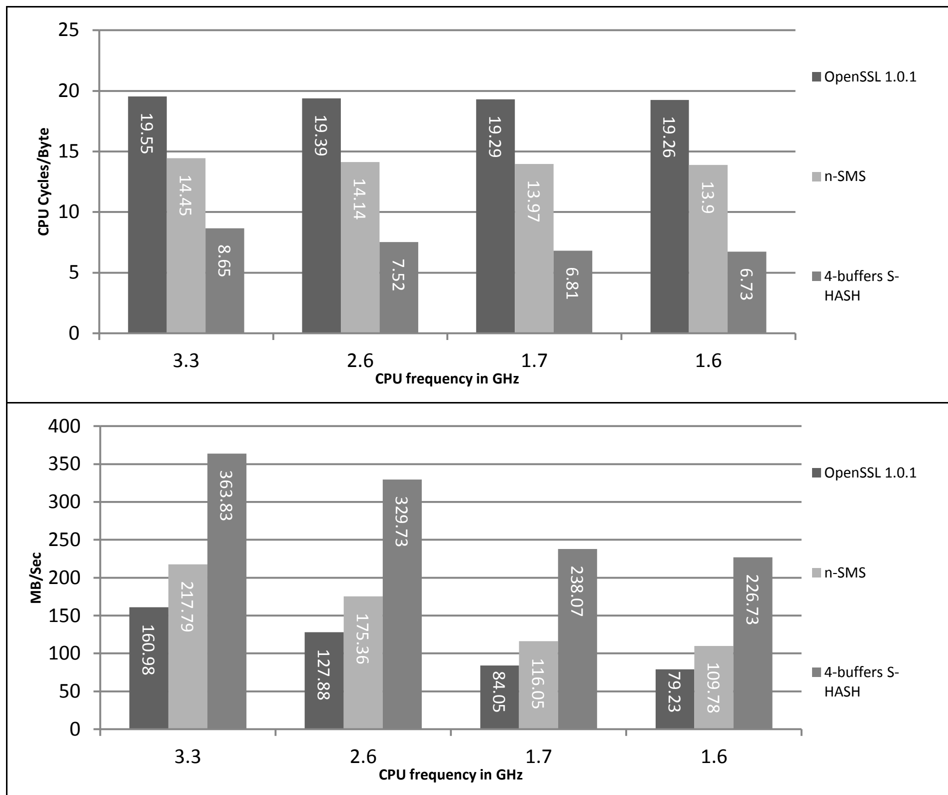

Fig. 4. Hashing the files in the directory UNIFORM (see explanation in the text). Measurements are taken on the Core i5-2500 operating at different CPU frequencies. Panel a shows the performance in Cycles per Byte. Panel b shows the performance in MB/sec.

5 Conclusion

We illustrated the general S-HASH approach, and demonstrated the advantage of a *4*-buffers SHA-256 S-HASH, running on the AVX architecture. The speedups we observe depend on the location of the data, but are significant in all cases. When

hashing equal length messages from any of the three levels of the processor's cache, or from main memory, the *4*-buffers SHA-256 S-HASH performs at ~*5.2* C/B. This is ~*2.25*x times faster than the best known serial hashing implementation. When hashing data from the hard-disk, the CPU performance is not the (only) limiting factor, because the disk's read performance becomes a bottleneck. Here, the *4*-buffers S-HASH method executes at effectively *8.65* C/B at the native processor speed, *3.3*GHz. This performance is *2.26*x faster than OpenSSL (1.0.1) and *1.67*x faster than the *n*-SMS method [5] under the same conditions (*19.55* C/B and *14.45* C/B, respectively).

We mentioned above two scenarios that require hashing of multiple messages, and can enjoy an S-HASH implementation: An OS check of the integrity of its components (during boot time), and data de-duplication. In addition, SSL/TLS servers that need to support multiple connections could also take advantage of an S-HASH implementation, if their software is set to process data from multiple connections in parallel. We suggest that the potential performance gain might be worth the hassle of tweaking the software to accommodate such parallelization.

Since the *4*-buffers S-HASH operates on *4* buffers in parallel, one might wonder why it does not achieve the theoretical four-fold speedup factor, compared to the alternative implementation. We mention here two of the reasons: a) the 2nd Generation Core™ Processors have an efficient ALU unit that can process data at a faster rate than the SIMD unit. This closes some of the theoretical four-fold gap that AVX can offer; b) SHA-256 algorithm has a significant amount of rotations. Compared to a single ALU instruction (ROR), the S-HASH method needs to implement rotation by a flow of two (SIMD) shifts, followed by a (SIMD) xor.

Hashing from a hard-drive introduces a different consideration. The RAID array (of two Solid State Drives) that we used in our experiments had throughput of *400*MB/sec. At *3.3*GHz, this throughput is equivalent to processing at the rate of *7.15* C/B. This explains the results that we obtained: while the processor can hash data at *5.18* C/B with the 4-buffers S-HASH method if the data read from the cache (or memory), this performance cannot be reached when the data is fetched from the disk. This is why we get only *8.65* C/B (for the UNIFORM case), but as already noted, this is still significantly faster than the serial alternative. When the processor is clocked to *1.6*GHz, the same disk throughput becomes equivalent to processing at the rate of *3.81* C/B. Thus, on the under-clocked systems, we were able to hash at *6.73* C/B, which is closer (only *1.31*x slower) to the processor's hashing capability (*5.18* C/B). The remaining gap between the system-wise performance and the maximal processing capability can be attributed to OS overheads, and to the fact that the accessing data stored in the disk is non-sequential (but rather distributed between four areas).

The soon to be released Haswell architecture [9] will support AVX2 with integer instructions that operate on *256*-bit registers. With this architecture, we could upgrade our method to implement *8*-buffers S-HASH efficiently - in theory, doubling the performance of the *4*-buffers S-HASH. However, for hashing data from the disk, we note that the SSD drives are not expected to double their throughput (at least in this time frame), so we should expect less than a twofold speedup.

Note that we intentionally did not study an S-HASH implementation of SHA-512. The reason is that SHA-512 operates on *64*-bit "words", and therefore, the current AVX architecture can support only a *2*-buffers SHA-512 S-HASH. This makes the S-

HASH method less attractive because a) the SHA-512 ALU implementations are already fast with the *n*-SMS method (*8.72* C/B); b) while each SHA-512 Update compresses *128* bytes of the message and a SHA-256 Update compresses only *64* bytes, SHA-512 involves *1.25x* more rounds in the processing than SHA-256 (*80* rounds versus *64*). We therefore speculate that SHA-512 S-HASH implementations would become useful only on the AVX2 architectures (doing a *4*-buffers S-HASH), but will be slower than *8*-buffers SHA-256 S-HASH on that architecture.

We conclude this study by stating that our results show that for some usages, SHA-256 is significantly faster than commonly perceived.

Finally, we add a few related remarks on the five SHA3 finalists [13]. Skein and Keccak use *64*-bit words, and the remark we made on SHA-512 holds similarly. Blake, JH and Grøstl, already use SIMD instructions in their better performing implementations. Therefore, applying the S-HASH method to these algorithms would create a delicate tradeoff with the S-HASH and the benefits of their current use of the SIMD instructions. Such optimization would be an interesting study to carry out.

6 References

1. Aciicmez, O.: Fast Hashing on Pentium SIMD Architecture. M.S. Thesis, School of Electrical Engineering and Computer Science, Oregon State University (2004).
2. Chuanyi Liu, Yingping Lu, Chunhui Shi, Guanlin Lu, David H.C. Du , Dong-Sheng Wang: ADMAD: Application-Driven Metadata Aware De-duplication Archival Storage System. Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os, 29-35 (2008).
3. Federal Information Processing Standards Publication 180-2: Secure Hash Standard. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
4. Gaudet, D.: SHA1 using SIMD techniques, http://arctic.org/~dean/crypto/sha1.html.
5. Gueron, S., Krasnov, V.: Parallelizing message schedules to accelerate the computations of hash functions (2012), http://eprint.iacr.org/2012/067.pdf
6. Gueron, S., Krasnov, V.: [PATCH] Efficient implementations of SHA256 and SHA512, using the Simultaneous Message Scheduling method (2012). http://rt.openssl.org/Ticket/Display.html?id=2784&user=guest&pass=guest
7. Intel: Intel Advanced Vector Extensions Programming Reference. http://software.intel.com/file/36945
8. Intel: 2nd Generation Intel® Core™ Processor Family Desktop Datasheet, Vol. 1, page 11. http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html,
9. 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/
10. Linux Manual, hdparm, http://linux.die.net/man/8/hdparm
11. LinuxMM: Drop Caches, http://linux-mm.org/Drop_Caches
12. Locktyukhin, M.: Improving the Performance of the Secure Hash Algorithm (SHA-1), Intel, http://software.intel.com/en-us/articles/improving-the-performance-of-the-secure-hash-algorithm-1/ (March 2010).
13. NIST, cryptographic hash Algorithm Competition. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html
14. OpenSSL, The Open Source toolkit for SSL/TLS, http://openssl.org/.

15. The Chromium Project, Verified Boot. http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot

7 Appendix : files size distribution for the "DIVERSE" director

Table 1. The lengths (in bytes) of the *350* files in the DIVERSE directory, when they are sorted by an alphabetic order of the file-names (from left column top to right column bottom).

14,336	122,960	15,440	73,280	6,150,304	77,312	183,872	80,464	100,352
5,120	651,264	15,440	116,224	44,112	140,800	12,352	93,184	7,936
68,096	277,624	64,512	5,632	16,960	157,696	48,720	20,992	51,712
227,840	169,080	60,928	55,128	62,464	287,744	220,752	15,472	343,040
497,152	50,808	106,576	16,896	82,944	126,464	50,768	35,456	25,600
61,008	256,120	194,128	98,816	116,224	30,272	230,400	3,531,136	324,608
80,384	7,680	28,752	982,912	120,320	140,352	60,416	19,008	25,088
18,432	7,680	61,440	265,088	17,920	26,112	40,512	426,496	31,744
8,704	8,192	87,632	301,784	20,544	8,192	1,524,816	461,312	30,720
286,720	41,472	97,856	294,064	119,680	15,424	128,592	401,920	184,832
47,104	303,464	23,040	530,496	50,768	224,832	46,592	161,792	36,432
14,976	307,560	24,128	9,728	33,280	11,136	14,848	24,656	29,184
14,720	311,640	155,728	3,286,016	243,712	7,168	130,048	185,936	29,184
293,376	311,656	270,848	195,072	95,312	6,784	92,672	34,896	217,680
740,864	30,760	28,240	204,800	153,160	367,168	111,616	68,864	17,488
1,481,216	393,264	6,656	29,696	20,992	32,320	83,968	12,496	129,024
178,752	13,104	45,056	70,224	60,928	8,064	309,248	23,552	200,272
17,664	64,080	90,624	34,304	114,752	60,496	24,064	30,088	6,656
38,912	64,592	95,232	24,576	106,560	947,776	165,376	199,168	46,672
30,320	32,896	41,984	290,368	65,600	35,328	204,800	199,168	71,760
27,008	89,600	72,192	23,104	115,776	24,064	214,096	192,256	363,584
28,736	21,760	118,784	55,376	113,152	56,320	158,720	192,256	294,992
288,336	292,864	552,448	223,448	22,016	164,352	55,296	29,184	24,248
48,840	740,864	98,344	3,440,660	17,024	57,856	145,920	1,897,328	161,872
65,088	1,485,312	132,648	646	35,392	44,544	11,264	44,544	24,576
70,168	146,036	35,104	31,232	158,712	259,072	76,800	26,624	59,904
143,792	112,128	21,160	122,368	228,752	374,864	104,016	15,872	17,920
350,208	172,544	468,480	26,624	9,984	51,264	29,696	23,552	27,776
195,024	654,928	92,160	100,864	481,504	44,032	171,600	99,840	88,576
77,888	42,064	147,456	76,288	642,952	24,576	109,056	62,544	42,496
78,848	412,672	45,568	46,592	75,672	1,659,984	23,040	38,400	21,056
158,720	10,240	17,488	32,896	100,904	6,144	23,552	38,400	12,800
271,872	334,416	460,504	30,208	283,744	149,056	94,208	125,440	22,096
54,824	12,288	21,584	751,616	40,448	167,488	26,624	41,536	52,304
15,360	491,088	39,504	14,416	30,208	318,976	14,336	327,680	40,448
284,736	339,536	24,144	105,472	49,216	72,832	13,824	48,640	14,336
3	182,864	514,048	537,112	31,232	131,584	14,336	9,728	16,464
7,533,568	499,200	102,400	410,688	94,784	97,280	16,896	19,968	21,504
7,533,568	60,416	40,448	39,024	155,216	75,840	43,584	98,816	

(SHA256_simultanious_hashing_2012_07_03_v02.docx)