Przemysław ZALEWSKI, Marcin ŁUKOWIAK, Stanisław RADZISZOWSKI

Rochester Institute of Technology, Rochester NY, U.S.A.

Case Study on FPGA Performance of Parallel Hash Functions

Streszczenie. *Funkcje haszujące (nazywane też funkcjami skrótu) odgrywają istotną role we współczesnej kryptografii. Funkcje te przekształcają dowolną ilość danych wejściowych w skrót o ściśle określonej długości. Typowe zastosowania funkcji haszujących obejmują weryfikację spójności danych i schematy komunikacji z uwierzytelnieniem. Poniższy artykuł jest głosem w dyskusji na temat jakimi właściwościami powinien sie charakteryzować nowy standard dla funkcji haszujących SHA-3. W naszym przekonaniu, niezmiernie istotną cechą nowego algorytmu powinna być możliwość do pracy w trybie równoległym. W niniejszym artykule szczegółowo opisujemy i analizujemy implementację funkcji haszującej PHASH w matrycy FPGA. Funkcja ta jest szkieletem który umożliwia przetwarzanie danych z wykorzystaniem szyfru blokowego. Głównym atutem PHASH jest to, że jego tryb pracy równoległej pozwala osiągnąć bardzo dużą wydajność. Nasze pomiary wykazały, że PHASH osiąga przepustowość 15Gbps w systemie z jednym szyfrem blokowym i 182Gbps przy wykorzystaniu 16 szyfrów blokowych.*

Abstract. *Hashing functions play a fundamental role in modern cryptography. Such functions process data to produce a small fixed size output referred to as a digest or hash. Typical applications of these functions include data integrity verification and message authentication schemes. We argue that high parallelizability of the forthcoming new SHA-3 hash standard should be a critical and achievable property of proposed algorithms. In this paper we present an FPGA design and performance analysis of a recently proposed parallelizable hash function PHASH. It is not a SHA-3 candidate but rather a hash template using tree hashing and a block cipher. The main feature of PHASH is that it is able to process multiple data blocks at once making it suitable for achieving ultra high performance. PHASH achieved a throughput over 15 Gbps using a single block cipher instance and 182 Gbps for 16 instances.*

Słowa kluczowe: skalowalna implementacja FPGA, funkcja haszująca, analiza wydajności
Keywords: scalable FPGA design, hash function, performance analysis

Introduction

A hash function $H : \{0,1\}^* \rightarrow \{0,1\}^m$ produces an m-bit digest of an arbitrary message, file, or even an entire file system. Typically, one wants hash functions to be easy to compute, but also infeasible to invert or to find collisions (pairs of inputs which hash to the same value). Hash functions are fundamental cryptographic primitives, and they are used extensively in authentication, preserving data integrity, digital signatures, and many other security applications [1], [2]. The two most widely used hash functions are MD5 (Message Digest, $m = 128$) and SHA-1 (Secure Hash Algorithm, $m = 160$), the latter supported by the US government as a standard FIPS-180-2 [3]. The collisions for MD5 were found five years ago [4], and by now they can be produced quickly by software available on the Net. The SHA-1 algorithm also seems to be in trouble (and other algorithms in the SHA-2 family, with $m = 256, 384, 512$, might follow). No collisions for SHA-1 have been found so far, but attacks much better than the simple birthday attack approach have been designed. Breaking SHA-1 soon is a likely possibility, which if happens, may lead to some dramatic tensions in computer security. The currently tentative SHA-2 doesn't have much support in the community. The computer security industry urgently needs a new high quality SHA-3 standard. On October 31, 2008, NIST (National Institute of Standards and Technology) closed the period of submitting proposals for the new hash function SHA-3 [5]. The design tune-up and the winner selection process is tentatively scheduled to conclude in December 2012.

The performance of hash function implementations, both in software and hardware, is critical because of the increasing need to hash very long inputs. As multi-core processors, and parallel systems are the dominant force in computing, some of the proposed hashing algorithms are attempting to take advantage of these resources by offering parallel hashing options. Allowing multiple parts of the data to be operated on simultaneously has the potential of reducing time complexity from $O(n)$ to $O(log(n))$. The algorithm PHASH hash function [6] is highly parallelizable, while Whirlpool and all of the SHA-family functions are not. PHASH is not a candidate in the NIST hash design competition. Still, the goal of this paper is to argue that during the evaluation process the community should put favorable attention to easily parallelizable candidates.

In this work we design and analyze the performance of a scalable Field Programmable Gate Array (PFGA) hardware for parallel hashing function - PHASH [6]. For comparison purposes, we have also developed high performance implementations of standard serial hash functions - SHA-512 [3] and Whirlpool [7] targeting the same FPGA platform. According to currently published literature [8], the fastest SHA-512 implementation on FPGA achieves a throughput of 1550 Mbps. Hashing function Whirlpool [7], [9]. provides security comparable to SHA-512 but is able to achieve much better performance. According to currently published literature, the fastest Whirlpool implementation on FPGA achieves a throughput of 4790 Mbps [10]. In this paper we do not address cryptographic security of hash functions, which is a property of foremost importance of the designs. The reader is encouraged to consult [1], [2] or postings at [5] or [11].

Overview of PHASH

PHASH [6] is a hash function designed to allow computations on data blocks to be performed in parallel. It consists of three stages: message padding, message compression and message reduction.

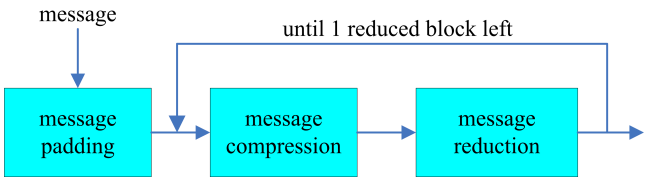

Fig. 1. High level block diagram of PHASH

Fig. 1 presents a high level diagram of the PHASH hashing function computed sequentially. The same function can be computed in parallel as outlined in the sequel. PHASH relies on a block cipher for message compression. Any sufficiently large block cipher could be used for this purpose. For completeness, the cipher W, as presented in the Whirlpool hashing function [7], is used here. The message reduction stage is responsible for combining the intermediate values generated by the message compression stage into a single hash.

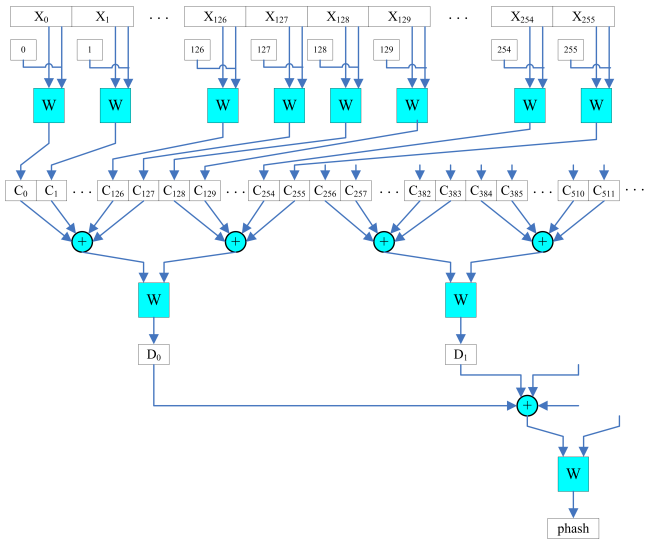

Fig. 2. Detailed block diagram of PHASH

Fig. 2 shows a detailed block diagram of the PHASH hashing function. The message compression stage takes each 896-bit block, X_i, and partitions it into two sub-blocks, Y_i and Z_i (not labeled explicitly in Fig. 2), where i represents the ith block of the message to be hashed, after padding. Y_i consists of the 384 most significant bits of X_i, and Z_i consists of the remaining 512 bits. The Y_i block is concatenated with an unsigned 128 bit representation of i, producing a 512-bit block, V_i. Recall that the cipher W[7] takes a 512-bit plaintext input and a 512-bit initial key. For each Y_i block the V_i block is used as the plaintext, and the Z_i block is used as the key. The 512-bit compressed block C_i is the resulting ciphertext. The computation of the C_i's can be performed in parallel, since the computations are performed on independent blocks of data. In the message reduction stage the previously computed C_i blocks are reduced into D_i blocks. Computation of each D_i block involves using the computed C_i blocks. The first 128 C_i blocks undergo the XOR operation with each other. The next 128 C_i blocks undergo the same operation. If less than 256 C_i blocks are computed, the remaining blocks consist of all '0' bits. As a result of the above process, 256 512-bit values are reduced to two 512-bit values. These two values are then fed into W as the key and plaintext, respectively, and a single 512-bit ciphertext D_0 is produced as a result. If more than 256 C_i blocks are computed, the next 256 C_i blocks are used in order to generate D_1 in the manner described above. The process continues until all D_i blocks have been generated. Then the same reduction process is applied to all D_i blocks until a single 512-bit value remains as a result of the reduction. This final 512-bit value represents the hash of the original message.

Previous work on SHA-512 and Whirlpool

The SHA-512 and Whirlpool designs in this work have been derived as follows. The reference SHA-512 implementation descriptions were obtained from a combination of [12], [8], [14], [15], [16], [17], and [18]. In [8] a partially unrolled SHA-512 implementation is presented. The message compression stage is unrolled two times. In [16] a quasi-pipelined implementation of SHA-512 is introduced. The message compression stage is pipelined, as it dictates the critical delay of the algorithm. The basic message expansion stage is altered so as to decrease the delay of the individual computations. The delay balancing technique is used to achieve a decrease in the delay of this stage. To perform the required additions carry-save and carry look-ahead adders are used. In [12] yet another SHA-512 implementation is presented. It draws on the collective strengths of most of the previously published work. It is quasi-pipelined and unrolled. The implementation uses carry-save and carry propagation adders, as presented in [17]. BlockRAMs are used to store the K_t constants, as presented in [18]. In [15] operation rescheduling was used. This technique also involves precomputation in order to minimize the critical delay.

The reference Whirlpool implementation descriptions were obtained from a combination of [10] and [13]. In [10] the substitute bytes stage was implemented using both a full look-up table placed in BlockRAMs, as well as by implementing the three mini s-boxes in the distributed RAM located in CLB slices. The twice unrolled architecture that used the mini s-box approach to implement this stage was also presented. As a result three different Whirlpool implementations were obtained. The shift columns stage was implemented with no additional logic; the data paths were hardwired to perform the reordering of the data. The mix rows stage was implemented by shifting the input byte around and the modular reduction step was performed, if necessary, on the fly. The add key stage consisted of only XOR operations. In [13] Whirlpool was implemented using the architecture that duplicates the key schedule, as well as the architecture which integrates the key schedule into the design. The substitute bytes stage was implemented using both the mini s-box approach and the Boolean expression approach. As a result four different Whirlpool implementations were obtained. The shift columns stage was implemented using combinational shifters. The mix rows stage was implemented by using a look-up table containing the results of the multiplications of all possible inputs by the reduction polynomial in $GF(2^8)$. The add key stage consisted of simple XOR operations.

SHA-512, Whirlpool and PHASH design

A model for each of the three algorithms was created using VHDL (Very High Speed Integrated Circuits Hardware Description Language). The SHA-512 implementation was simple and straightforward. No unrolling or operation rescheduling was used. Distributed RAM or BlockRAM were used to store the required K_t constants for the message compression stage. The Whirlpool implementation was more involved. Three implementations of the substitute bytes stage were created. The first implementation stored the s-box entirely in BlockRAM. The second implementation implemented the s-box using mini s-boxes. These mini s-boxes were implemented as look-up tables in distributed RAM. The final, third implementation of the substitute bytes stage implemented the mini s-boxes using simple Boolean expressions. Two implementations of the mix rows stage were also created. The first implementation performed the required Galois field multiplications using look-up tables stored in distributed RAM. In the second implementation, the Galois field multipliers were realized using Boolean expressions. All combinations of the above described implementations were then synthesized. Post place and route timing information was used in order to determine the combination which provides the most efficient Whirlpool implementation. Since the cipher W is at the core of the Whirlpool hashing algorithm, it was assumed that the most efficient Whirlpool implementation also contains the most efficient cipher W implementation. This implementation was then used in PHASH. It should be noted that only cipher W implementations which do not use BlockRAM were

considered as it would not be possible to implement PHASH with more than 4 instances of W, using BlockRAM, on most currently available FPGAs. Post place and route timing information showed that the most efficient Whirlpool configuration proved to be the one in which the substitute bytes stage was implemented using mini s-boxes; they were implemented using look-up tables and stored in distributed RAM. The Galois field multipliers in the mix rows stage were implemented using Boolean expressions. The model of PHASH is highly configurable with two restrictions. The first one is that the number of instances of W is required to be a power of 2. The implementation in this work also restricts the maximum number of reduction levels in PHASH to three. The structure of the PHASH model is shown in Fig. 3. All components are described below.

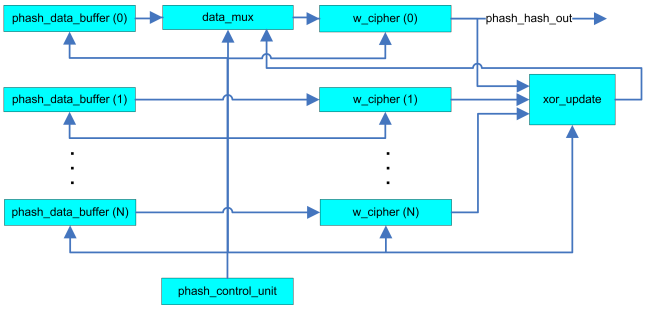

Fig. 3. Block diagram of PHASH design

- The $phash\_data\_buffer$ units are 896-bit shift registers. They simply collect input data when enabled.
- The $data\_mux$ unit consists of two 4-to-1 multiplexers. They are used to provide the appropriate plaintext and key to the cipher W units. The four inputs consist of the Z_i and V_i blocks and two accumulators at each of the three reduction levels. These accumulators are named c_left, c_right, d_left, d_right, e_left and e_right.
- The w_cipher unit contains an implementation of the cipher W. It takes as inputs entire 512-bit plaintext and key blocks and outputs the resulting 512-bit hash over a single cycle. A multiplexer is added in order to ensure that the newly specified key is to be used at the beginning of the first round.
- The xor_update unit accumulates the hashes of the compressed blocks at each reduction level. Two 512-bit registers are used, one to accumulate the results of the first 128 blocks, and the other to accumulate the results of the remaining 128 blocks at that reduction level. This unit also provides data to the first cipher W instance through the data multiplexer.
- The $phash\_control\_unit$ unit is responsible for synchronizing the operation of all components in order to implement the PHASH hashing algorithm.

The state diagram for the control unit is shown in Fig. 4. The initial state is called $idle$. Once the $phash\_run$ signal is asserted a transition to the $read\_data$ state occurs. At the same time data is ready to be read and the data buffer is enabled. Once an entire block of data has been read a transition to the rst_wp state occurs. At this time, the cipher W instances are ready to be reset and the data multiplexer unit passes Z_i and V_i to W. This discussion assumes that the final hash is not yet being computed. A discussion of the final hash computation now follows. During the rst_wp state certain cipher W instances are enabled, while others are disabled. If the total number of data blocks to be processed, $data\_num\_blocks$ is less than the number of instances,

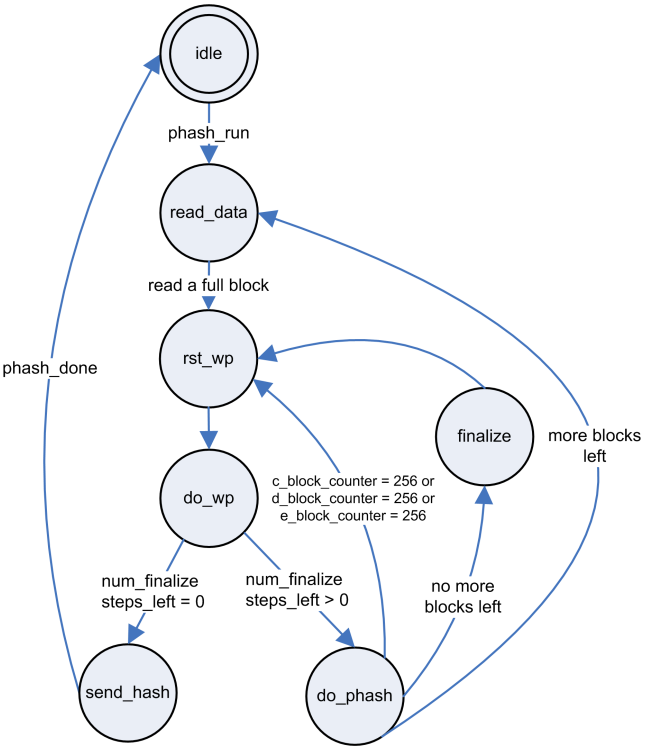

Fig. 4. State diagram of the PHASH control unit

$num\_instances$ only $data\_num\_blocks$ instances are enabled, while the others are disabled. Otherwise all cipher W instances are enabled until num_blocks_left becomes less than $num\_instances$. When this occurs only num_blocks_left instances are enabled, and the rest are disabled. After the appropriate instances are enabled a transition to the do_wp state occurs. The majority of the PHASH algorithm is controlled in this state. This state always waits for the wp_hash_ready signal from the cipher W instances. The control logic is broken down by the total number of reductions which need to be performed. If only a single reduction level is required $c\_block\_counter$ is incremented every time a block is processed, and num_blocks_left is decremented accordingly. After a block is processed and $num\_finalize\_steps\_left$ is greater than 0, a transition to do_phash occurs, otherwise if the final hash has been computed a transition to $send\_hash$ occurs.

In the do_phash stage, if $c\_block\_counter$, $d\_block\_counter$ or $e\_block\_counter$ is less than 128 the resulting hashes will be accumulated into c_left, d_left or e_left respectively. Otherwise the results will be accumulated into c_right, d_right or e_right respectively. The appropriate counters are also incremented. If there are still more blocks to process and if the final hash is not being computed a transition back to the $read\_data$ state occurs. If two or more reduction levels are required the algorithm progresses as described above. However once $c\_block\_counter$, $d\_block\_counter$ or $e\_block\_counter$ equals 256, an additional hash needs to be computed using the appropriate registers from the XOR update unit as the key and plaintext. This is accomplished by a transition back to the rst_wp state. The computation is performed in the do_wp state. During this computation only the first W instance is enabled. After the computation completes the appropriate set of registers is reset. On the next cycle a transition to the do_phash state occurs. When there are no more blocks to process a transition

from the do_phash state to the $finalize$ state occurs. After entering this state $num\_finalize\_steps\_left$ passes through the hashing states are performed. The value of $num\_finalize\_steps\_left$ depends on the highest reduction level attained. When finalizing the hash only the first cipher W instance is enabled. The initial key and plaintext inputs to this instance are c_left and c_right. If two reductions are to be done the next set of key and plaintext inputs are d_left and d_right. Likewise, if three reductions are to be done the final set of key and plaintext inputs are e_left and e_right. When the final hash is ready a transition to the $send\_hash$ occurs. The final hash is contained in the final hash output of the first cipher W instance. It is output over 8 clock cycles and upon completion the $phash\_done$ signal is asserted.

Testing functionality

Throughout the implementation process many tests were performed to verify the functionality of individual components and the entire system. The VHDL testbench was responsible for reading pre-padded data to be hashed from a file, sending the data at the appropriate time, as well as comparing the obtained hash to a known value. The SHA-512 and Whirlpool test data consisted of 240 tests, starting with 20 bytes of data and incrementing by 20 bytes for each consecutive test, for a total of 4800 bytes. The PHASH test data consisted of three separate tests. The first consisted of 128 test cases. The first case ensured only a single block is hashed, and each consecutive test hashed an extra block. As a result 128 tests ranging from a single block to 128 blocks were performed. The second test consisted of hashing 256 and 257 data blocks. These block counts cause the transition between the first and second reduction levels. The final test consisted of hashing 65536 and 65537 blocks. These block counts cause the transition between the second and third reduction levels.

Hardware verification

The available hardware platform to verify the functionality of the designs was the Xilinx ML410 development board. The ML410 is a complete development system containing a Virtex-4 FX60 FPGA as well as a multitude of additional peripherals. The Virtex-4 FX60 FPGA is a hybrid FPGA, containing not only FPGA logic, but also two PowerPC 405 cores. Several other notable features of this FPGA include 232 18-Kbit BlockRAMs, up to 20 DCMs, 25280 CLB slices, equivalent to 56880 logic cells, 128 XtremeDSP slices as well as multi-gigabit RocketIO transceivers. Additional features of the ML410 development board used in these implementations include 256 MB DDR-2 external memory, 512 MB CF card and a single UART. The features of the Virtex-4 FX60 FPGA used include CLB slices, BlockRAMs and DCMs. Xilinx ISE 9.2.04i was used in order to obtain utilization and timing information. A total of 10 hardware systems were created, two for SHA-512, six for Whirlpool and two for PHASH. The limitation for PHASH was due to the fact that the Virtex4 FX60 FPGA can accommodate PHASH with up to two cipher Ws only. Table 1 summarizes the implementation results. Since all implementations achieved an operating frequency of at least 100 MHz it was possible to integrate them into a system which utilized a PowerPC processor with its communication bus operating at 100 MHz. Xilinx EDK 9.2.02 software was used in order to design and implement the final hardware systems required to test the functionality of the implementations.

Algorithm	Configuration	Slices	Freq. (MHz)	Throughput (Mbps)
SHA-512	Distr. RAM	2073	106.65	1365
SHA-512	BlockRAM	1917	103.17	1321
Whirlpool	1	6605	122.09	6251
Whirlpool	2	6597	123.50	6323
Whirlpool	3	7327	105.28	5390
Whirlpool	4	7937	112.38	5754
Whirlpool	5	4833	138.96	7115
Whirlpool	6	3914	137.08	7018
PHASH 1	1 W	11010	126.71	11353
PHASH 2	2 W	16901	124.24	22264

Table 1. Implementation results for Virtex-4 FX60 FPGA

Algorithm	Slices	Freq. (MHz)	Throughput (Mbps)
SHA-512	1102	142.88	1829
Whirlpool	2892	162.34	8312

Table 2. SHA-512 and Whirlpool results for Virtex-5 LX330 FPGA

High performance design

To exploit parallelizability of PHASH and determine its maximum possible throughput on the state-of-the-art FPGAs the VHDL model was synthesized into Virtex-5 LX330 device. For comparison purposes SHA-512 and Whirlpool were also implemented on the same platform. Table 2 shows the results obtained for SHA-512 and Whirlpool. It is important to note that the slice counts in Virtex-5 devices are not directly comparable to those in Virtex-4 devices. The Virtex-5 devices have less slices, however each slice contains about twice as many resources as a single Virtex-4 slice. It is evident that the performance optimized Whirlpool implementation greatly outperforms the SHA-512 implementation in terms of throughput. However, it requires more than twice as many slices to implement. Table 3 shows a similar set of metrics for several PHASH implementations. For each implementation a different number of cipher W instances were included. By increasing the number of instances a significant increase in throughput was observed, at the cost of increased slice usage. The speedup factor represents the efficiency at which the implementation scales. The throughput obtained with a single W instance is used as the baseline and therefore its speed-up factor is 1.00. The maximum theoretical speedup factor would be equal to the number of W instances used. Looking at the results in Table 2 and Table 3 it is also evident that the PHASH implementations greatly outperform both SHA-512 and Whirlpool in terms of throughput. The PHASH implementation with a single cipher W instance requires slightly more slices than the Whirlpool implementation; however it is able to achieve nearly twice as much throughput. For a visual comparison, a bar graph of the maximum throughput of all three algorithms targeted for the Virtex-5 LX330 FPGA is shown in Fig. 5.

By looking at the number of slices required in order to achieve 1 Mbps of throughput both the area and operating frequency can be incorporated into a single metric. The SHA-512 implementation requires 0.60 slices per 1 Mbps of throughput, whereas the Whirlpool implementation requires

cipher W instances	Slices	Speedup Factor	Freq. (MHz)	Throughput (Mbps)
1	4469	1.00	168.07	15059
2	8031	1.92	161.29	28903
4	13537	3.69	155.04	55566
8	24427	6.70	140.85	100958
16	42362	12.13	127.39	182624

Table 3. PHASH result for Virtex-5 LX330 FPGA

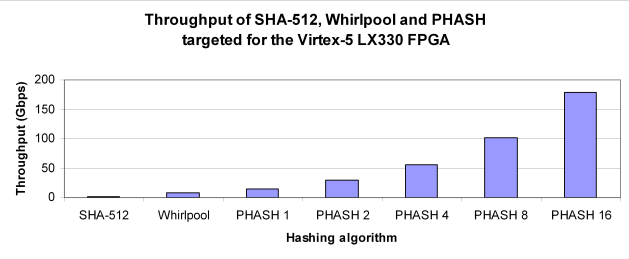

Fig. 5. Throughput comparison

only 0.35 slices per 1 Mbps of throughput. A PHASH implementation with a single cipher W instance requires only 0.30 slices per 1 Mbps of throughput. The remaining implementations require between 0.28 and 0.24 slices per Mbps. This shows that the PHASH implementations utilize the available slices very efficiently, even as the implementation scales.

Conclusion and future work

The main point of this paper is that high parallelizability of the forthcoming new SHA-3 hash standard should be a critical and achievable property of the proposed algorithm. This is supported by demonstrating that the PHASH can be seen as a template of a general hash design with such properties. In the technical part we presented a case study on FPGA performance of PHASH. By exploiting parallelism PHASH is achieving a much higher throughput than any other currently available serial hashing function. A PHASH implementation using only a single cipher W is able to achieve over 15 Gbps of throughput. When the number of cipher W instances is increased to 16, a throughput of over 182 Gbps is achieved. A fair comparison between SHA-512 and Whirlpool on the same FPGA implies overall better performance of Whirlpool. Several assumptions were made during the implementation of PHASH in order to facilitate the development process. Future work should include removing most, if not all, of these assumptions in order to create a less constrained implementation. The most noticeable restriction in this implementation of PHASH is that the number of cipher W instances used is required to be a power of 2. If a new implementation were to remove this restriction several more instances would be able to fit onto the Virtex-5 LX330 FPGA. Finally, the implementation in this work restricts the maximum number of reduction levels in PHASH to three. Future implementations can remove this restriction.

BIBLIOGRAPHY

[1] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone: *Handbook of Applied Cryptography*, CRC Press, 1996.

[2] D. R. Stinson, *Cryptography: Theory and Practice*, third edition, CRC Press 2006.

[3] FIPS 180-2. Secure Hash Standard, August 2002. (NIST). http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[4] X. Wang and H. Yu, *How to Break MD5 and Other Hash Functions*, LNCS, 3494:19–35, 2005.

[5] NIST SHA-3 timeline, http://csrc.nist.gov/groups/ST/hash/timeline.html

[6] A. Kaminsky and S. Radziszowski: *A case for a parallelizable hash*, in Proceedings of IEEE MILCOM, San Diego, CA, November 2008.

[7] P. Barreto and V. Rijmen: *Whirlpool Hash Function*, 2006. http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html

[8] F. Aisopos, K. Aisopos, D. Schinianakis, H. Michail, and A. P. Kakarountas: *A novel high-throughput implementation of a partially unrolled SHA-512*, Proceedings of the Mediterranean Electrotechnical Conference, 61–65, 2006.

[9] W. Stallings, *The Whirlpool Secure Hash Function*, Cryptologia, 30:55–67, 2006.

[10] M. McLoone, C. McIvor, and A. Savage: *High-speed hardware architectures of the Whirlpool hash function*, Proceedings - IEEE International Conference on Field Programmable Technology, 147–153, 2005.

[11] SHA-3 Zoo, unofficial hash function candidates evaluations http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo

[12] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane: *Optimisation of the SHA-2 family of hash functions on FPGAs*, Proceedings - IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 317–322, 2006.

[13] P. Kitsos and O. Koufopavlou: *Efficient architecture and hardware implementation of the Whirlpool hash function*, IEEE Transactions on Consumer Electronics, 50(1):208–213, 2004.

[14] I. Ahmad and A. S. Das: *Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs*, Computers and Electrical Engineering, 31(6):345–360, 2005.

[15] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis: *Improving SHA-2 hardware implementations*, Lecture Notes in Computer Science, 4249 NCS:298–310, 2006.

[16] L. Dadda, M. Macchetti, and J. Owen: *An ASIC design for a high speed implementation of the hash function SHA-256 (384, 512)*, Proceedings of the ACM Great Lakes Symposium on VLSI, 421–425, 2004.

[17] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott: *Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512*, Information Security. 5th International Conference ISC 2002. Proceedings (Lecture Notes in Computer Science Vol.2433), 75–89, 2002.

[18] M. McLoone and J. V. McCanny: *Efficient single-chip implementation of SHA-384 and SHA-512*, 2002 IEEE International Conference on Field-Programmable Technology (FPT). Proceedings (Cat. No.02EX603), 311–14, 2002.

[19] N. Sklavos and O. Koufopavlou: *On the hardware implementations of the SHA-2 (256, 384, 512) hash functions*, Proceedings - IEEE International Symposium on Circuits and Systems, 5:153–156, 2003.

Authors:
Przemysław Zalewski, Marcin Łukowiak,
Department of Computer Engineering
Stanisław Radziszowski,
Department of Computer Science
Rochester Institute of Technology,
Rochester, NY 14623, U.S.A.
corresponding email: mxleec@rit.edu