Performance improvements for SHA-3 finalists by exploiting microcontroller on-chip parallelism

Pal-Stefan Murvay and Bogdan Groza
Department of Automatics and Applied Informatics
Politehnica University of Timisoara, Romania
Email: stefan.murvay@gmail.com, bogdan.groza@aut.upt.ro

***Abstract*—As ubiquitous devices, microcontrollers are deployed in a great variety of applications many of which involve communication over insecure channels that require cryptography. Here we investigate the possibility of using on-chip coprocessors from currently available microcontrollers for increasing computational power by employing parallelism. For this we focus on the analysis of SHA-3 finalists in order to identify features that can lead to an efficient parallel implementation with on-chip coprocessors. Experimental results on a Freescale S12X family microcontroller equipped with an XGATE coprocessor are presented. In this two core environment, speedups between 18 and 73 percents are obtained for the five SHA-3 finalists. In our software implementations BLAKE proves to be the best performer, especially for short messages, followed at some range by Grøstl, then by Skein, Keccak and JH.**

***Keywords*-SHA-3; hash function; parallelism; coprocessor.**

I. Introduction and motivation

Microcontrollers are pervasive devices used in many categories of equipments ranging from home appliances to industrial control systems. As these microcontroller-driven systems become more and more interconnected, more emphasis falls on the security of the communication channels employed. Whether authenticity, confidentiality or another security objective is required, assuring it involves the use of cryptographic algorithms. Augmenting communication protocols with the use of these algorithms should involve small communication and computation overhead especially in safety critical systems. This mainly depends on the computational performance and architecture of the device being used and since microcontrollers usually have low computational power it is not a simple task to fulfill.

The problem of implementing cryptography on resource constrained devices has been widely studied and several proposed solutions were successfully used in practice. One category of solutions focus on devising secure protocols which require little computational power and reduced variants of cryptographic functions. A good example in this area comes as a result of the intense research activity in sensor networks which produced solutions ranging from efficient protocol design [16] to efficient cryptographic primitives [11]. Small scale variants of hash functions were also proposed for use in RFID environments which can be even more constrained than sensor networks [12]. However, collisions on these functions were already reported [19]. Another category of solutions are based on hardware implementations. Using ASIC or FPGA-based cryptographic hardware to perform the computation of required primitives increases performance along with the costs of production. Dedicated cryptographic coprocessors were developed to accelerate the execution of different primitives. Examples of such hardware implementations can be found in [15] and [20]. Some efforts were also made in enhancing the performance of general-purpose microcontrollers by extending their instruction set with application-specific instructions used in cryptographic algorithms [10]. Although they reach good performances, these hardware-based solutions are application dependent and require extra time to be spent on designing them in comparison to a software-based solution. Therefore software solutions based on microcontrollers that are already available on the market may be preferred in various contexts.

Microcontrollers are constantly evolving to handle the need for increased performance. Different manufacturers are already offering microcontrollers with on-chip general purpose coprocessors or even dual core microcontrollers [17], [7]. This category of microcontrollers could be used to enhance cryptographic performance by using parallelism. One way to put this into practice is by running multiple instances of a function in parallel on each core to achieve high throughput. However, applications do not always require this kind of parallelism. Most commonly, a single execution of a cryptographic primitive is needed at a certain time so in order to attain speedups we have to search inside each individual algorithm for steps that can be parallelized. Some frequently used cryptographic algorithms were studied in this respect for the implementation of an FPGA-based crypto processor [4] and similar solutions are needed for multicore microcontrollers. Here we also investigate methods of improving the performance of cryptographic primitives by using parallelism. We mainly focus our attention on the five SHA-3 finalists: BLAKE, Grøstl, JH, Keccak and Skein but also take into account some other well known hash functions: MD5, SHA-1 and the SHA-2 family. As block ciphers are commonly used as a building block for hash functions, we additionally look at this category of algorithms in search for parallelism. For this we keep RC5 as a reference for memory consumption as the lite nature of this algorithm is well known. Several common features present in some classes of cryptographic functions can be exploited for parallelism but the main speedup is provided by parallelizing the specific structure of each algorithm. We

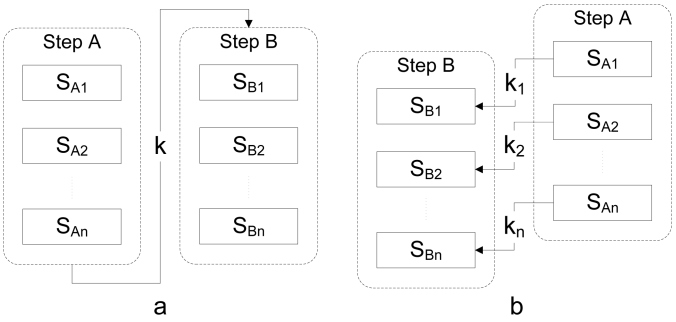

Fig. 1. Transforming sequential execution (a) into parallel execution (b)

show the performance improvements that were obtained for some algorithms on a Freescale S12X 16-bit microcontroller which incorporates an XGATE coprocessor as well as the memory requirements for each implemented function.

The remainder of this paper is organized as follows. Section II describes general aspects about implementing parallelism on different cryptographic functions while section III contains implementation details for three well known hash functions, the most recent SHA-3 finalists and the RC5 block cipher. Some details on the microcontroller used for testing and performance analysis are presented in section IV. Finally, section V holds the conclusion of our work.

II. Using parallelism to improve algorithm performance

The performance of cryptographic algorithms can be improved by making platform specific code optimization. However, these optimizations only offer limited improvements and largely depend on the individual features of each microcontroller: instruction set, bit size, endianness, etc. Features that characterize each microcontroller lead to a performance bottleneck for sequential implementations. Therefore parallelism has to be used in order to obtain further performance improvements.

Most cryptographic algorithms were not designed for parallel implementation and they mainly consist of chained steps, meaning that the output O of step S_{i-1} is used as an input for step S_i. However, some of the steps still can be executed in parallel. Even though in many cases only coarse-grained parallelism can be obtained, this is suitable for use on dual core microcontrollers.

In order to identify steps that can be executed in parallel the algorithm has to be divided in basic steps and input/output dependencies have to be determined. If the output of a step S_A is not needed in order to start execution of the next step S_B then the former can be run on a parallel processor. Another case for applying parallelism is when the output of a substep S_{Ai} is only needed by the a corresponding S_{Bi}, i.e., $S_{Bi} = f(O_{Ai})$. Figure 1 shows how sequential steps in a) can be executed in parallel in b). We use these two observations in what follows to identify parallelizable steps which are specific for several classes of cryptographical algorithms.

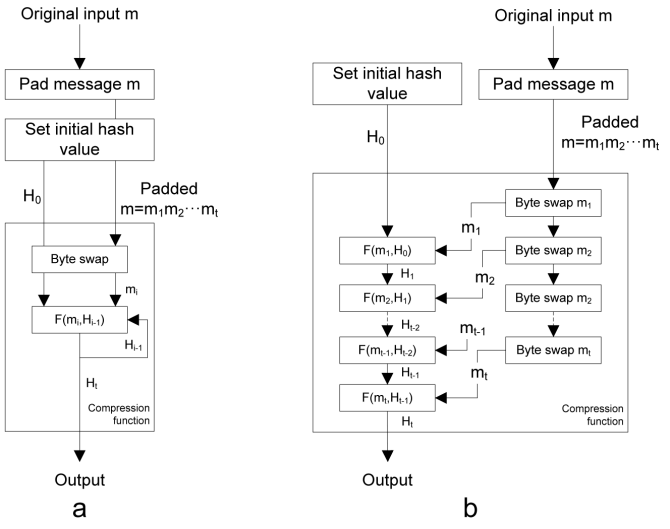

Fig. 2. General construction of an iterated hash function. Sequential (a) and parallel (b)

A. Hash functions

Hash functions are probably the most commonly used cryptographic primitives. They generate a fixed length binary output for a variable length binary input. We focus on iterated hash functions as they are the most prevalent in this category. Although they accept arbitrary length input, hash functions work generally on fixed length blocks. Therefore the input message first has to be padded so that its length is a multiple of the hash block size. An initialization step follows and after it the input is passed through the compression function one block at a time. Figure 2.a shows the basic construction of an iterated hash function.

Some parallelism is possible for this general hash function algorithm model. The padding step for example doesn't have to be executed sequentially. The only constraint here for is that each padding byte has to be added to the input string before the compression function needs to start processing it. A common substep of the compression function is one related to the endianness on which the algorithm is based and the endianness of the used microcontroller. If the endianness differs, a byte swap to the correct endianness has to be made before processing the input and this can also be done in parallel with the main compression substeps. This byte swap is common not only to hash functions but to other types of cryptographic functions as well so we will not mention it for each of the following subsections. Further improvements could be made on the compression function depending on each design and we detail this in section III for several chosen hash functions. Figure 2.b depicts the general parallelized construction for iterated hash functions.

Message authentication code (MAC) algorithms are cryptographic primitives built for the purpose of providing message authentication. Given an input message of variable length and a key, these algorithms generate a fixed length output called MAC. Some of the most commonly used MAC constructions are based on other cryptographic primitives such as hash

functions or block ciphers. There exists also custom algorithms which are specifically designed for providing message authentication. Whether parallelism can be used in MAC algorithm implementations in general mainly depends on their individual construction and since they are quite differently built a rule of thumb cannot be stated. However, for the MAC algorithms based on other cryptographic primitives, parallelism comes from the algorithms used as a building block where the rules given previously for hash functions apply. This also holds in the case of HMAC which is the most commonly used MAC construction.

B. Block ciphers

Block ciphers are symmetric key algorithms used in many cryptographic systems to provide confidentiality. The user's secret key is generally passed through an initial step for key expansion or key derivation depending on the algorithm. Usually this key processing step is completely independent of the encryption/decryption step. So if the output of the key processing step is $k = (k_1 k_2 \cdots k_n)$ the encryption/decryption algorithm will first use k_1 then k_2 and so on. The aforementioned fact can be exploited by executing the two steps in parallel provided that k_i is generated by the time the encryption/decryption algorithm needs it. This approach greatly increases encryption speed when keys are often changed.

Depending on the mode of operation employed for each cipher block further parallelism can be achieved. When running in electronic codebook (ECB) mode each plaintext block is encrypted independently into a ciphertext. By processing each block on a separate processor the overall encryption speed is increased by a factor depending on the number of processing units. Unfortunately, this method is insecure in general. Cipher block chaining or CBC is a secure mode of operation but it's construction does not allow parallelism. Still, parallelism is possible in counter mode which is secure and exploited in various scenarios.

C. Estimating performance improvement

After identifying the steps that can be parallelized it is important to get an estimation on the improvement that could be obtained in order to decide whether or not to implement the changes. To be more accurate with our exposition, we first define what a *step* is.

Definition 1. *A step S_i corresponding to a logical part of an algorithm is a sequence of instructions that operate together as a group on a given input $I = (I_1 I_2 \cdots I_n)$ to generate an output $O = (O_1 O_2 \cdots O_m)$. Each step can be comprised of one or more substeps S_{ij}.*

If we consider T_{CPU} as the time (measured in clock cycles) needed by the main CPU to execute a certain step then a coprocessor CPU will execute that same step in $s \cdot T_{CPU}$ clock cycles. Here $s > 0$ is a constant defining the coprocessor speed factor in relation to the main CPU. This element is influenced by a number of processor-specific features like: operation frequency, instruction set, type of memory used, etc. Having the execution time of a step S_i both on the main processor as T_i and the coprocessor as $s \cdot T_i$ we can assert that if $c_i \geq s \cdot T_i$ the entire step can be executed on the coprocessor, hence a speed gain of $T_i - \tau_i$. Here c_i denotes the maximum number of clock cycles that can be spent until the output of S_i is needed on the main processor while τ_i is the time needed for exchanging data between the processors. Otherwise, if $c_i < s \cdot T_i$ then only a fraction of c_i/s from the step was executed on the coprocessor while the rest can be executed either on the main CPU or further parallelized, etc. So we can sumarise this gain as:

$$\begin{cases} T_i - \tau_i & \text{for} \quad c_i \geq s \cdot T_i \\ c_i/s - \tau_i & \text{for} \quad c_i < s \cdot T_i \end{cases} \tag{1}$$

However, in a two core environment such as S12X it is not efficient to swap the execution of a particular code sequence from the main CPU to the coprocessor. Therefore, in the case that a particular step once assigned to the coprocessor must be finalized there, we have to change this relation to:

$$\begin{cases} T_i - \tau_i & \text{for} \quad c_i \geq s \cdot T_i \\ T_i - (s \cdot T_i - c_i) - \tau_i & \text{for} \quad c_i < s \cdot T_i \end{cases} \tag{2}$$

Therefore, in the best case scenario the speed gain is T_i if the coprocessor can execute S_i in up to c_i cycles. Otherwise, if $s \cdot T_i > c_i$ the main processor will have to wait until the step is finished but still with a certain speed gain. It is clear that for very slow coprocessors the gain will be a negative number denoting that executing S_i on the coprocessor can't bring a speedup. Relation (2) can thus be written in a more compact form as:

$$min\left(T_i, c_i + T_i\left(1 - s\right)\right) - \tau_i \tag{3}$$

Next by considering that each S_i is executed r_i times on the coprocessor, we can calculate the overall computational time reduction as:

$$\sum_{i=0,n} r_i \left(min\left(T_i, c_i + T_i\left(1 - s\right)\right) - \tau_i\right) \tag{4}$$

As an example let us take BLAKE and try to estimate the speed gain for an input of length 0. Let s be $1/4$ as XGATE is approximately 4 times faster than the main CPU. We identify four basic steps that can be parallelized on BLAKE: byte swapping (BS), message padding (MP), round operation (RO) on a column or a diagonal. On the main S12X CPU the computation time is:

- $T_{MP} = 2213$ cycles for message padding,
- $T_{BS} = 2005$ cycles for byte swap,
- $T_{RO} = 1020$ cycles for one round function.

The corresponding repetition times for each of these steps is: $r_{MP} = 1$ as the padding is only done once, $r_{BS} = 1$ because we have only one block and $r_{RO} = 14 \cdot 6$ because we have 14 rounds in which 3 columns and 3 diagonals are computed on XGATE. The execution time for all steps fits in relation $c_i \geq s \cdot T_i$ therefore the actual speed gain can be computed

as $T_{MP} + T_{BS} + 84 \cdot T_{RO}$ which gives 89898 cycles. This is close to the value presented in the experimental results, the difference coming from the τ_i which we did not consider in our calculus. We underline that this synthetic evaluation can provide only a rough measure of the speedup and only careful measurements can give the exact gain which depends on several other aspects such as memory, instruction set, etc.

III. Implementation details

We analyze in more detail several specific cryptographic functions in order to find algorithm-specific features that can be exploited for parallel implementations. In what follows, parallel implementation details are given for each of the algorithms included in our study.

A. MD5

MD5 is a 128 bit output hash function developed by RSA Inc. as an improved version of MD4 [18]. It is still used in many applications although MD5 was proven to be insecure and collisions have been found since 2005 [22]. The MD5 algorithm is based on the general iterative hash function construction depicted in Figure 2. The compression function of MD5 operates on 512 bit message blocks divided in 16 4-byte words which have to be set in little-endian order before processing. As each step of the compression function uses the output of the previous step the construction is not suitable for parallelism. Thus, in general only the padding can be done in parallel. Also, on big-endian machines the words can be swapped to their little-endian form in parallel with the compression round.

B. SHA-1

Since its release by NIST as a Federal Information Processing Standard, SHA-1 [14] was deployed in various protocols and security standards. The construction of SHA-1 is similar to the one of MD5 so the remarks on parallelism for MD5 also applies to SHA-1. The main differences are represented by the size of the message digest, the functions used by the compression function and initial hash value. Additionally in each compression round, the 64 bytes message block is expanded to words is expanded to a block of 320 bytes or 80 words. As the words in the message schedule obtained by expansion are processed one by one this expansion is independent of the compression process. Therefore the message expansion can be executed in parallel with the compression function.

C. SHA-2

SHA-2 consists of a set of four hash functions each having a message digest size denoted by it's suffix: SHA-224, SHA-256, SHA-384 and SHA-512. The block size on which they operate is 512 bits for the first two and 1024 for the last two. As the algorithm structure of the SHA-2 family components is similar to the structure of SHA-1 (including the message expansion step) the same approach is to be used for adding parallelism.

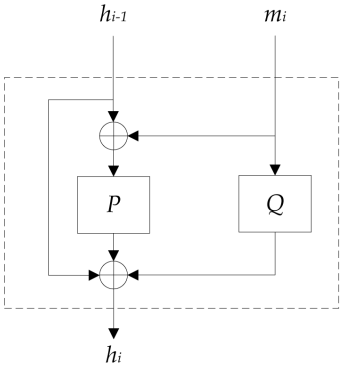

Fig. 3. Grøstl compression function

D. SHA-3 finalists

In 2007 NIST announced a public competition for a new cryptographic hash algorithm which will be part of the next secure hash standard under the name SHA-3. As one of the evaluation criteria for the submitted algorithms was the computational efficiency (referring to the speed of the algorithm) the candidates were built so that they best fit this measure. Some algorithms also exploited parallelism of different granularities suitable for hardware and/or software implementations [21]. We studied the candidates accepted for the final round of the SHA-3 contest in search for parallelism. Reference implementations available at [5] were used for benchmarking and as a starting point for our parallel implementations. We outline that our results show performances close to the evaluation made by NIST, but this is only a benchmark made on a specific platform and different performances can be obtained on other platforms or implementations. One such example, which establishes a different performance order can be found in [9].

BLAKE [2] is based on the HAIFA iteration mode having a compression function which receives as inputs: a message block, a chain value, a salt and a counter representing the number of hashed bits so far and generates a new chain value. An inner round of the BLAKE compression function is a modified version of the ChaCha stream cipher and operates on a 4×4 state matrix of words. Independent operations are made on all four columns followed by operations on distinct disjoint diagonals. As the BLAKE specification states, this construction allows four-way parallelism so that operations on all four columns can be computed in parallel and identically for the diagonals. It is clear that for two core architectures (both cores equal in computational power) each core will take care of updating two columns and diagonals respectively. In our particular case, due to the increased speed of the coprocessor, we were able to execute one column/diagonal update on the main CPU in parallel with updating three columns/diagonals on XGATE.

Grøstl [8] is another iterated hash function which borrows some components from the AES construction. The Grøstl compression function is based on two fixed permutation. The two underlying permutations P and Q are combined as illustrated in Figure 3. These two permutations are com-

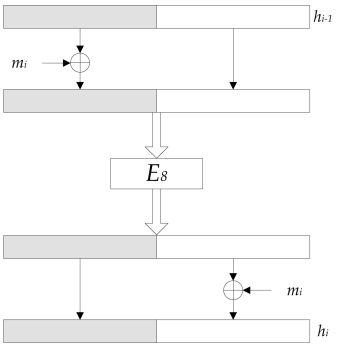

Fig. 4. The F_8 compression function of JH

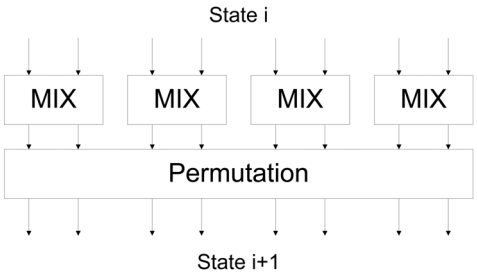

Fig. 5. One basic round of Threefish-512

pletely independent of each other. Thus, implementing the compression function on two-way parallel architectures is quite straightforward especially as P and Q involve the same amount of computation. More in depth optimizations, by exploiting platform specific features, were done in [1].

JH [23] was built around a compression function F_8 which employs a large block cipher structure E_8. The first half of the $2m$ bit state is XORed with the current message block before feeding it to the E_8 permutation. The second half of the state computed by E_8 is also XORed with the current message block as shown in Figure 4. The E_8 construction uses 4×4 bit S-boxes, a linear 8-bit transformation L and a permutation P. An efficient implementation of JH utilizes the bit-slicing technique. In the bit-slice implementation, the 1024-bit input of E_8 is split in eight 128-bit words on which seven round functions (two S-boxes, one linear transformation L and four permutations ω) are applied. On a 128-bit processor, basic operations on each word would be done with one instruction while on a smaller bit size processor they will have to be constructed out of multiple instructions operating on smaller words. As the S-box and L functions can be executed independently on each corresponding bit of the eight words, parallelism can be used to speed-up the computation. The ω permutation swaps bits in a 128 words depending on the round number. This prevents bit level or nibble level parallelism for the full E_8 permutation. Therefore, we can only use nibble level parallelism in each round. Additionally, although not resulting in a large speed gain, the two XOR operation of the state and message block can also be made in parallel with other computations (E_8 round function for the first XOR and message truncation for the second).

Keccak [3] is defined as a family of sponge functions built on a set of seven permutations. Permutations are applied in two phases: one called absorbing phase, the second called squeezing phase, each based on the sponge construction. The $Keccak - f[1600]$ studied here consists of 24 rounds that operate on a state matrix of 5×5 lanes, where a lane is considered to be a 64-bit word. $Keccak-f[b]$ is a permutation over $\mathbb{Z}_2^b$. Each round consists of five steps. Some of these steps allow parallel execution as they are applied over elements of the state matrix. When using interleaving as in the optimized implementation submitted by the Keccak designers (on which we based our implementation), the process of setting bytes into interleaved words and vice versa can be also parallelized as it is applied on each word independently.

Skein [6] uses Threefish, a tweakable block cipher, running in Unique Block Iteration mode (UBI) to build its compression function . The core of Threefish employs three mathematical operations (one addition, one rotation by a constant and an XOR) to construct a simple mixing function called MIX on 128 bits. One round of Threefish in Skein-512 corresponds to the application of four MIX functions followed by a permutation of all 64-bit words in the state as depicted by Figure 5. Once every four rounds a subkey is injected in the current state. Skein-256 and Skein-1024 are similar, the only difference being that two mix functions are applied for a 256-bit state and eight for 1024. For parallel implementations each of the MIX functions in one round could be computed on a separate processor as they operate on independent 64-bit words of the state. As stated before, XGATE can handle several iterations of a certain block of code during a single execution of the same block on the S12X CPU. This allowed us to run 3 MIX functions on XGATE in parallel with one on S12X.

E. RC5

The RC5 encryption algorithm is a well known block cipher based on a simple structure. It's light construction does not leave much place for optimizations and presents no parallelizable features. An improvement could be obtained if the key processing operation is done in parallel with the actual encryption/decryption having the restriction that each element of the expanded key table S is ready by the time it is needed for processing.

IV. Performance analysis

In order to assess the performances that can be achieved for each function by employing parallelism, we implement these functions on an automotive grade microcontroller. Our choice is motivated by the increased interest shown in past years for automotive security. As the range of applications made available in modern vehicles is increasing, so are the security threats, emphasizing the need for using cryptography in these systems.

The microcontroller used is a Freescale 16 bit microcontroller from the S12X family which includes an XGATE coprocessor (that can be up to 4.5 times faster [13]. According

to the MC9S12XDT512 datasheet [7] the micro can run at a frequency of 40 MHz and is equipped with 512Kbytes of FLASH memory and 20Kbytes of RAM offering enough space for a wide range of applications.

All cryptographic primitives described in the previous section were implemented on our S12X microcontroller both in the sequential and parallel form. These implementations were analyzed in relation to two metrics: execution time and memory utilization.

A. Execution time

As a first measure of performance we chose to evaluate the execution time of the algorithms in their sequential versus parallel implementations. The execution time was measured in number of clock cycles needed to generate the output for inputs of different sizes. Depending on the function type we made tests for certain input sizes.

For all the hash functions we wanted to see if the effect of the padding step over the total runtime can be alleviated. Hence the input sizes were selected so that best/worst cases (minimum/maximum number of padding bytes needed) are covered in the tests. The block size of each hash function implementation was selected to be 512 bits in order to allow comparative evaluation. For Keccak we used $\lfloor Keccak\,[r = 1088, c = 512]\rfloor_{256}$ which we denote by Keccak-256. Table I shows the execution time for the chosen inputs. It can be easily observed that the parallel implementations spend the same amount of time for all inputs with the same block count after padding. As the padding rules differ from one hash function to another, the block size changes at different input sizes. For example in JH each message is padded so that its length is a multiple of 512 bits adding at least 512 bits (when the message is already a multiple of 512) and at most 1023 bits are added to the message. However, some exceptions from this rule exist in the case of BLAKE which contains an additional step in the compression function executed in case the last block only contains padding bytes, hence the results obtained.

Table II sumarises the improvements obtained for parallel implementations of each function given different input lengths. The best improvement, around 73%, was obtained for BLAKE and JH, followed by Keccak and Skein. The improvement of Grøstl is comparable with the ones obtained for SHA-1 and SHA-256 for small inputs, but it increases for larger inputs. MD5 shows the smallest improvement as the compression function does not offer significant means for applying parallelism.

For our RC5 implementation with 12 rounds and a 128 bit key, the key generation step takes 73109 clock cycles on S12X while the encryption of one block takes 8031 and 8640 for decryption. On XGATE the key is generated in 25530 cycles. By running the key generation on XGATE in parallel with an encryption on S12X (26016 cycles) the resulting speedup is 67,94 %. Because the key generation is more computational intensive the encryption step is still delayed until the first words are available in the expanded key table so the actual improvement achieved only due to parallelism is just 9.30%.

Function	Code size (bytes)	
	Sequential	Parallel
MD5	4694	4466
SHA-1	8185	4644
SHA-256	17601	6257
BLAKE-32	5754	7121
Grøstl-256	24905	36973
JH-256	6345	17123
Keccak-256	31982	33506
Skein-512-256	13919	11476
RC5	1570	2095

TABLE III
MEMORY CONSUMPTION

B. Memory utilization

In Table III we present the code size of our sequential and parallel implementations. From the SHA-3 finalists, Blake and JH show the lowest memory consumption in sequential implementation followed by Skein, Grøstl and Keccak. For parallel execution BLAKE remains the best performer as it requires the smallest amount of memory. Where it was permitted we took advantage of the increased speed of XGATE to optimize code not for speed but for size (e.g., we gave up loop unrolling) resulting in a smaller overall memory consumption.

V. Conclusion and future work

Several commonly used cryptographic algorithms were analyzed in order to evaluate the possibility of introducing parallelism for increased execution speed. Some general aspects concerning parallelization, common to members of the same class, were presented as well as specific details for several selected primitives. Experimental analysis of our implementations show that the execution speed can be improved with factors in the range of 10 - 70% depending on the algorithm and coprocessor used. In particular, parallel implementations of the SHA-3 finalists can run 18% to 73% faster. This values hold for a scenario on which the XGATE coprocessor is up to 4.5 times faster than the CPU (the actual speed of XGATE depends on various parameters: memory type used for storing code and data, instructions used, etc.). In most cases this improvement comes at the cost a tradeoff in what concerns the used memory. However for some algorithms and on coprocessors that run faster than the main CPU, speedups can be obtained along with a decrease in code utilization when applying code size optimizations.

As future work we first look at analyzing and implementing parallelized versions of other symmetric and asymmetric primitives. Additional platforms will also be considered for implementation.

Acknowledgement

This work was partially supported by the strategic grant POSDRU 107/1.5/S/77265, inside POSDRU Romania 2007-2013 co-financed by the European Social Fund Investing in People and by CNCSIS-UEFISCDI project number PNII-IDEI 940/2008.

Input size(bytes)		0	55	56	64	119	1024
MD5	Sequential	29837	29987	57239	55371	55445	437241
	Parallel	25672	25672	49378	49378	49378	404968
SHA-1	Sequential	93826	93976	185111	183243	183317	1487315
	Parallel	60791	60791	119537	119537	119537	1000727
SHA-256	Sequential	241691	241841	478219	476351	476425	3995111
	Parallel	146812	146812	288978	288978	288978	2421468
BLAKE-32	Sequential	122755	122680	242263	240172	240093	2001367
	Parallel	33780	33931	65038	64887	65038	531492
Grøstl-256	Sequential	239973	240248	398361	394769	395044	2716709
	Parallel	160765	160765	242124	242124	242124	1462509
JH-256	Sequential	496546	989139	989135	987062	1479655	8344802
	Parallel	133826	262992	262992	262992	392158	2200482
Keccak-256	Sequential	613917	614684	614693	614765	615260	4532859
	Parallel	257281	258048	258057	258129	258624	1908699
Skein-512-256	Sequential	433565	434150	434159	434177	641616	3541491
	Parallel	354395	354395	354395	354395	522861	2868658

TABLE I
EXECUTION SPEED OF IMPLEMENTED HASH FUNCTIONS FOR DIFFERENT INPUT SIZES

Input size	Improvement (%)							
(bytes)	MD5	SHA-1	SHA-256	BLAKE-32	Grøstl-256	JH-256	Keccak-256	Skein-512-256
0	13.96	35.21	39.26	72.48	33.01	73.05	58.09	18.26
55	14.39	35.31	39.29	72.34	33.08	73.41	58.02	18.37
56	13.73	35.42	39.57	73.15	39.22	73.41	58.02	18.37
64	10.82	34.77	39.33	72.98	38.67	73.36	58.01	18.38
119	10.94	34.79	39.35	72.91	38.71	73.50	57.97	18.50
1024	7.38	32.72	39.39	73.44	46.17	73.63	57.89	19.00

TABLE II
IMPROVEMENT OBTAINED FOR DIFFERENT INPUT SIZES

REFERENCES

[1] K. Aoki, G. Roland, Y. Sasaki, and M. Schläffer. Byte Slicing Grøstl - Optimized Intel AES-NI and 8-bit Implementations of the SHA-3 Finalist Grøstl. In *Proceedings of The Sixth International Conference on Security and Cryptography, SECRYPT 2011*. SciTePress, 2011.

[2] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 3), 2010.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The Keccak SHA-3 submission. Submission to NIST (Round 3), 2011.

[4] R. Buchty, N. Heintze, and D. Oliva. Cryptonite a programmable crypto processor architecture for high-bandwidth applications. In C. Moeller-Schloer, T. Ungerer, and B. Bauer, editors, *Organic and Pervasive Computing ARCS 2004*, volume 2981 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin / Heidelberg, 2004.

[5] ECRYPT. The SHA-3 Zoo. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo, 2011.

[6] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein Hash Function Family. Submission to NIST (Round 3), 2010.

[7] Freescale. *MC9S12XDP512 Data Sheet, Rev. 2.21, October 2009, [Online]. Available: http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf*, 2004.

[8] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlffer, and S. S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST (Round 3), 2011.

[9] D. Gligoroski, S. J. Knapskog, J. Amundsen, and R. E. Jensen. Internationally Standardized Efficient Cryptographic Hash Function. In *Proceedings of The Sixth International Conference on Security and Cryptography, SECRYPT 2011*. SciTePress, 2011.

[10] J. Groschdl and E. Savas. Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2m). In *Cryptographic Hardware and Embedded Systrems CHES 2004*, pages 133–147. Springer Verlag, LNCS 3156, 2004.

[11] C. Karlof, N. Sastry, and D. Wagner. Tinysec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 162–175, New York, NY, USA, 2004. ACM.

[12] M. Macchetti and P. Rivard. Small-scale variants of the secure hash standard. In *ECRYPT workshop on RFID and lightweight cryptography*, Graz, Austria, July 14-15 2005.

[13] R. Mitchell. *Tutorial: Introducing the XGATE Module to Consumer and Industrial Application Developers, March 2006, [Online]*. Freescale, 2004.

[14] National Institute of Standards and Technology. FIPS 180-3, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-3. Technical report, Department of Commerce, Aug. 2008.

[15] S. Okada, N. Torii, K. Itoh, and M. Takenaka. Implementation of Elliptic Curve Cryptographic Coprocessor over GF(2m) on an FPGA. In *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '00, pages 25–40, London, UK, 2000. Springer-Verlag.

[16] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. The TESLA broadcast authentication protocol. *RSA CryptoBytes*, 5, 2002.

[17] Renesas Electronics. *SH7205 Group Hardware Manual, Rev. 2.00, March 2010, [Online]. Available: http://documentation.renesas.com/eng/products/mpumcu/rej09b0372_sh7205hm.pdf*, 2010.

[18] R. Rivest. The MD5 message-digest algorithm, 1992.

[19] M. E. Steurer. Multicollision attacks on iterated hash functions. Technical report, July 2006.

[20] G. E. Suh, C. W. O'Donnell, and S. Devadas. AEGIS: A single-chip secure processor. *Information Security Technical Report*, 10(2):63 – 73, 2005.

[21] M. S. Turan, R. Perlner, L. E. Bassham, W. Burr, D. Chang, S. jen Chang, M. J. Dworkin, J. M. Kelsey, S. Paul, and R. Peralta. Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition. NIST Interagency Report 7764, 2011.

[22] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Advances in Cryptology - EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 561–561. Springer Berlin / Heidelberg, 2005.

[23] H. Wu. The Hash Function JH. Submission to NIST (round 3), 2011.