Journal of Universal Computer Science, vol. 13, no. 3 (2007), 349-362
submitted: 30/11/06, accepted: 16/2/07, appeared: 28/3/07 © J.UCS

The Use of Runtime Reconfiguration on FPGA Circuits to Increase the Performance of the AES Algorithm Implementation

Oscar Pérez
(Université Henri Poincaré I, Nancy France
Laboratoire d'Instrumentation Electronique de Nancy
oscar.perez@lien.uhp-nancy.fr)

Yves Berviller
(Université Henri Poincaré I, Nancy France
Laboratoire d'Instrumentation Electronique de Nancy
yves.berviller@lien.uhp-nancy.fr)

Camel Tanougast
(Université Henri Poincaré I, Nancy France
Laboratoire d'Instrumentation Electronique de Nancy
camel.tanougast@lien.uhp-nancy.fr)

Serge Weber
(Université Henri Poincaré I, Nancy France
Laboratoire d'Instrumentation Electronique de Nancy
Serge.Weber@lien.uhp-nancy.fr)

Abstract: This article presents an architecture that encrypts data with the AES algorithm. This architecture can be implemented on the Xilinx Virtex II FPGA family, by applying pipelining and dynamic total reconfiguration (DTR). The originality of our implementation is that it computes sequentially in the FPGA the Key and Cipher part of the AES algorithm. This dynamic reconfiguration implementation allows a good optimization of logic resources with a high throughput. This architecture employs only 11619 slices allowing a considerable economy of the resources and reaching a maximum throughput of 44 Gbps.

Keywords: AES, FPGA, dynamic total reconfiguration, reconfiguration controller, pipeline, registers, iterative looping, unrolling looping, metrics, throughput, latency, reconfiguration time.
Categories: B.2.2, B.3.3, B.4.4, D.4.8, E.3, E.4

1 Introduction

The data security is a significant subject for which various algorithmic solutions have been proposed. In 2001, Advanced Encryption Standard (AES) was accepted as a FIPS (Federal Information Processing Standard) [NIST, 01]. AES is an encoding algorithm intended to replace DES, which had already showed some safety weaknesses in data protection. In October 2002 NIST (National Institute of Standards

and Technology) selected Rijndael cipher developed by two Belgian cryptographers as the AES algorithm. Since then, many achievements on hardware and software had been proposed by combining various architectures. In general, various architectures have been used to apply the AES algorithm on hardware. They seek to satisfy two metrics important in digital systems: the throughput, and the area or the amount of hardware resources required to achieve this throughput. The throughput reached goes from 20 Mbps to 70 Gbps according to the technology and the architecture used as described in [Elbirt et al, 01], [Standaert et al, 03], [Chodowiec et al, 01], [Hodjat et al, 04], [Jarvinen et al, 03] and [Kancharla et al, 03].

The technology of the circuits as well as the tools available for the design, the use and the implementation of the algorithms have played a significant role to achieve a high throughput, but with a high cost in terms of resources used. Nevertheless, the intrinsic parallelism of the algorithm is still well adapted to a hardware implementation. We chose to work on FPGAs because of their great design flexibility.

In this paper we propose one solution for the implementation of the AES algorithm in a pipelined and dynamically reconfigurable way. The originality of this approach is that this implementation can be realized using dynamic reconfiguration and allows obtaining a very good compromise between high speed and low area.

The paper is organized as follows. Section 2 gives a short description of the AES algorithm. Section 3 describes the related work and our approach. Section 4 details the choice of the implementation and section 5 presents the techniques suggested for the AES algorithm on FPGA technology. Section 6 presents the metrics used. Section 7 presents our experiments and results; in addition we describe and detail the different partitions, the synthesis aspects, our implementation results and a comparison with other works. Finally, we give conclusions and prospects about this work in section 8.

2 Description of the algorithm

The AES is a block cipher with possible block and key lengths of 128, 192 and 256 bits. The block to be encrypted and the key can be of different lengths. The encryption is comprised of a variable number of rounds (determined by the key and block lengths) with each round containing four transformations: *ByteSub*, *ShiftRow*, *MixColumn* and *Round Key Addition* (in the last round, the *MixColumn* is omitted). An initial key is expanded to form an Expanded Round Key based on the number of rounds. Since AES is a symmetric cipher, decryption is just the inverse of the encryption. If more details are needed see [NIST, 01], [FIPS, 99]. [Fig. 1] shows the operation of the algorithm [Angel, 00].

3 Related work

In general, various architectures have been used to apply the AES algorithm on hardware. Next we described some of the most interesting approaches.

In an effort to achieve the maximum efficiency possible, some authors not implant the key scheduling. Rounds keys for encryption are loaded from the external keys bus and are stored in internal registers. Then, all keys must be loaded before

encryption may begin [see Elbirt et al, 01]. According to [Chodowiec et al, 01], they unroll all cipher rounds, together with their internal registers. [Hodjat et al, 04] present the architecture of a fully pipelined AES encryption processor on a single chip FPGA. By using loop unrolling, inner-round and, outer-round pipelining techniques. They use block RAM for their implementation.

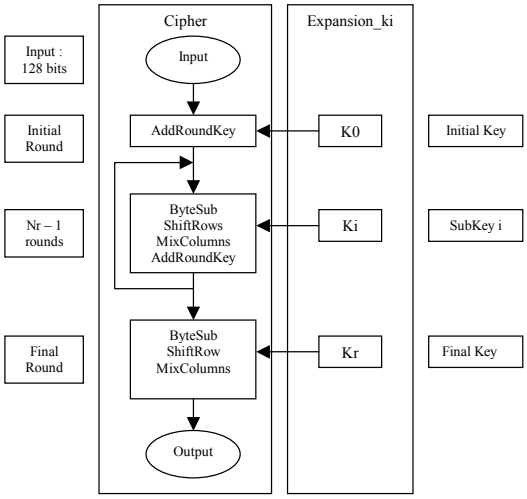

Figure 1: Operation of the two parts of the algorithm.

[Kancharla et al, 03] compute the key on the fly with the rounds. In these works the reconfiguration is used in order to change the functionality between encryption and decryption. The first configuration unrolls the key and encrypts the data, whereas the second configuration unrolls the key and decrypts the data.

4 Choice of implementation

We decided to split the AES algorithm into two partitions: *Expansion_ki* partition (expansion key) and the *Cipher* partition (data encryption), [see Fig. 1]. By contrast with other works, in this study we concentrate only on one aspect of the AES: encryption. Thus we do the following: in the first step, the *Expansion_ki* partition is loaded into the FPGA, in order to expand the key. The second stage consists in re-configuring the FPGA with the *Cipher* partition in order to encrypt the data. Furthermore, we combined this dynamic reconfiguration with pipelining. [Fig. 2] shows the comparison between a static implementations of the AES algorithm in FPGA and our proposal called P-DR (Pipeline- Dynamic Reconfiguration). Thus, the original algorithm was broken in two principal parts: *Expansion_ki* partition (key Expansion) and the *Cipher* partition (data Encryption). The choice of splitting the algorithm in two partitions was dictated by an optimizing methodology described in [Tanougast et al, 03]. This methodology can be adapted to different objectives, one of them being reducing the FPGA resources and the size of the memory needed for data

retention between the reconfigured partitions. By cutting the algorithm between the key expansion and the data cipher, we ensure a minimization of the memory size, because only the expanded keys are needed for the second partition [Liu et al, 04]. This separation also ensures that the expanded keys are located right where there are needed in the cipher. This is an advantage compared to the other works, where these keys need to be routed from the expansion key part to the cipher part.

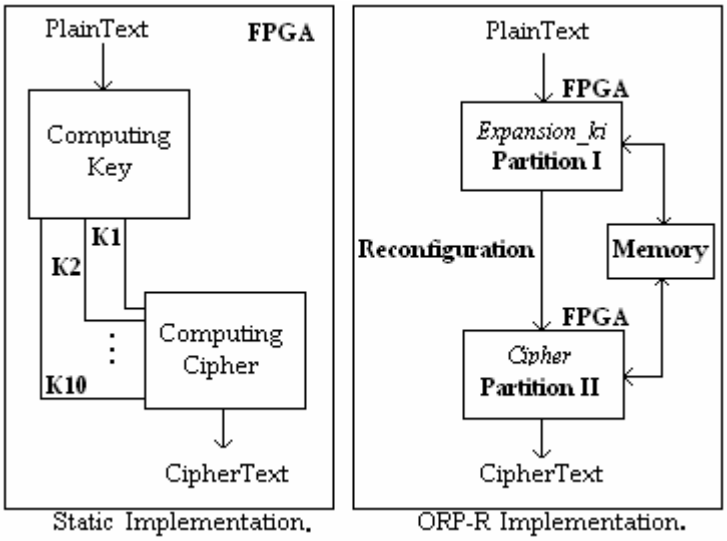

Figure 2: Comparison of the two implementations.

Furthermore, if the same key is used for several data blocks we also ensure a minimization of the number of reconfigurations.

4.1 Expansion_ki partition

The AES algorithm takes the Cipher Key *K*, and performs a Key Expansion routine to generate a key schedule (i.e. the ten different keys that will be used later by the Cipher module). The Key Expansion generates a total of *Nb*(*Nr*+*1*) words: the algorithm requires an initial set of *Nb* words and each of the *Nr* rounds requires *Nb* words of key data. The resulting key schedule consists in a linear array of 4-byte words, denoted [*wi*], with i in the range $0 < i < Nb(Nr + 1)$ [NIST, 01], [FIPS, 99].
The data are arranged in a linear vector of words of 4 bytes, indicated by *[wi*]. The data are put to the algorithm through *dato_e* (128 bits) and the result is provided at *dato_s*. Let us specify that *temp* is a variable of 32 bits wide and *w[i]* is the line of a matrix that has a dimension of 4 by 4 bytes. *RotWord* function takes a word of 32 bits [a0, a1, a2, a3] as input, carries out a cyclic permutation, and returns the word [a1, a2, a3, a0]. *SubWord* is a function that takes on its entry a word of four bytes and applies a look-up matrix *S_Box* to each four byte to produce a new word. This matrix has a size of 256 data of 8 bits each. The constant *Rcon*[i] contains already defined values. For word indices that are integer multiple of *Nk* (number of 32-bit words

comprising the Cipher Key), a transformation is applied to *w[i-1]*, followed by a XOR with a constant iteration, *Rcon*[i]. The transformation is composed of a circular shift of the bytes in a word (*RotWord*), followed by a look-up of each byte in a word (*SubWord*). [Fig. 3] shows the block diagram of the execution of a single round for this module.

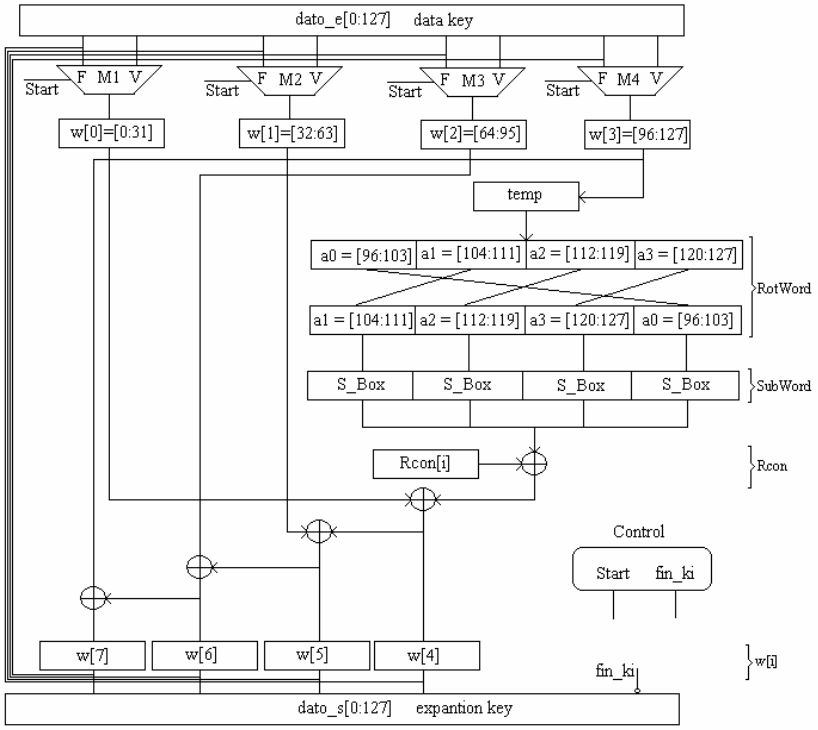

Figure 3: Diagram for the Expansion_ki module for only one round

4.2 Cipher partition

At the beginning of the Cipher module, the input is stored in the *State* array that has a size of 128 bits (16 bytes). Following the addition of the Ki key(i-th key), the *State* array is modified by applying the standard round (Nr-1 times) and a final round, which does not include the Mixcolumns transformation. Finally, *State* is sent to the output. The various transformations (*SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*) that treat the *State* array are described in the following sub-sections.

- *SubByte* is a non-linear function, operating independently on each byte from the State vector, known as a substitution box (*S-Box*).
- The *ShiftRows* function shifts the data (this function divides its input in 4 segments of 4 bytes each and makes a rotation towards the left of respectively 0, 1, 2, 3 bytes for segments 1, 2, 3 and 4).

- *MixColumns* is a function that transforms each byte of input into a linear combination of bytes. This function can be expressed mathematically as a matrix product in the body of Galois (2^8) [NIST, 01]. This matrix multiplication uses multiplications in "finite fields" by two and three, that reduce to an exclusive-OR function and thus makes the architecture more efficient [McLoone et al, 03].
- *AddRoundKey* transformation, *Ki* (previously generated by the *Expansion_ki* module) is added to *State* by an *XOR* operator. Each *Ki* is composed of Nb words that are generated by the module *Expansion_ki*. K*i* is the k^{th} sub-key calculated by the algorithm starting from the main key *K*. The application of the *AddRoundKey* transformation in the Nr rounds of *Cipher*, occurs when 1< *round* <= *Nr* [NIST, 01].

Finally, as it can be seen on [Fig. 4], the transformations *SubBytes, ShiftRows* and *MixColumns* are always used. So, the reusability of these operators can be exploited here.

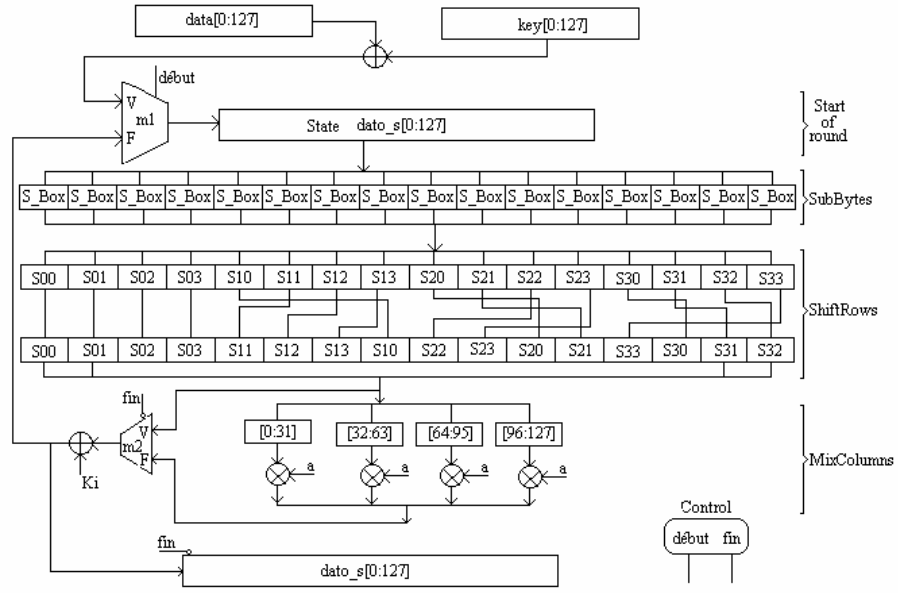

Figure 4: Diagram for the Cipher *module*

4.3 Reconfiguration Controller

The main role of the controller is to indicate to the processor (or the FPGA itself) when it must load the next bitstream. The controller of reconfiguration can be implemented in various manners. In the case of a SoC, this one can be done by a hardwired module (ASIC), by a processor or a portion of the FPGA. Here, this controller is a static module, i.e., the cells that implement it will not be modified in the case of an FPGA with partial reconfiguration. In the case of an FPGA with global reconfiguration, the controller will be present in an identical way in all the

configurations, but it will not be able to carry out the whole reconfiguration process. This is required, because the controller indicates when the FPGA must be reconfigured and a reconfiguration must take place at the end of each partition.

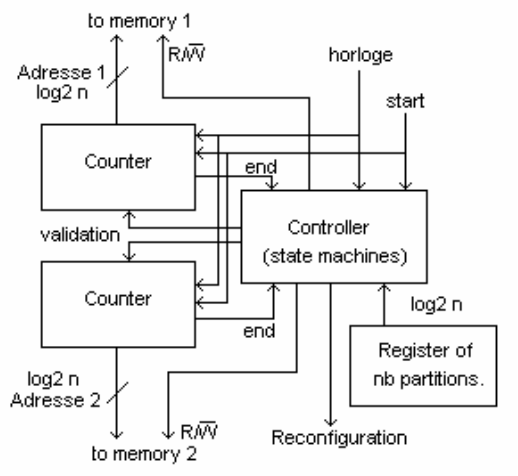

Figure 5: Diagram of the controller

This last case is presented here.

The method suggested requires at least two memory banks located on both sides of the reconfigurable array, in order to be able to read and write independently the input data and output of the various partitions. The controller allows writing and reading in the memory banks with an aim of permuting the data between configurations and loading the next configuration. It consists in two counters that point to read and write memory addresses and a state machine for the control of reconfiguration. This controller consumes 20 CLBs in each partition [see Fig. 5].

The process of runtime reconfiguration is described next. First, the *Expansion_ki* partition is loaded into FPGA in order to expand the sub keys. Then, the FPGA is reconfigured with the *Cipher* partition in order to cipher all the data. Thus, the encryption process is finished. But if a new key is detected, then all the process is started again by configuring the FPGA with the *Expansion_ki* partition.

5 Techniques suggested

Pipelining increases the throughput, but also the used resources [Patterson et al (96)]. Pipelining is an economic form of parallelism especially in FPGAs with flip-flops that are already presents in each cell whether they are used or not.

With the purpose of reaching high values of throughput, different ways to do pipelining have been proposed for the AES. All of them are based on Iterative Looping. Loop unrolling is a very known technique of speed improvement for iterative algorithms. Nevertheless, full loop unrolling leads to a significant increase of

processing resources. This method can be combined with pipelining and is known as outer-round pipelining [Chodowiec et al, 01], [Hodjat et al, 04]. When the hardware device does not have sufficient space for a complete loop unrolling or when resource reduction is needed, then a Partial Combination of Iterative Looping is used with Loop Unrolling. Another way to increase the throughput without an excessive cost of resources is to apply Sub-pipelining or Inner-round Pipelining. This consists in adding pipeline registers inside the loop without unrolling the later.

Reconfiguration or re-configurable computing, allows system designers implementing more hardware than they physically have. Re-configurable computing involves the manipulation of the logic within the FPGA at run-time. In other words, the design of the hardware may change in response to the demand placed upon the system while it is running. Re-configurable computing has several advantages: First, it is possible to reach a better functionality with a smaller hardware. The second advantage is a lower system cost. The final advantage of re-configurable computing is reducing the time-to-market [Barr, 98]. The reconfiguration makes possible using the same circuit to carry out various operations that leads to a reduced and better exploited hardware.

6 Metrics used

The combination of these two techniques: *Pipeline* and *Dynamic Reconfiguration*, are very interesting. On the one hand, the *Pipelining* offers an increase in the throughput and the reconfiguration provides saving in resources (that shrinks the side effect of the pipeline). On the other hand, some FPGA, like the Xilinx XC2V3000, allow dynamic reconfiguration with an acceptable time overhead, proportional to the used resources [Xilinx, 05].

In order to calculate the throughput, we considered two cases. For the first one (case a), the number of bits to encrypt is 128 or one data word; for the second one (case b), it is 3840000 bits (an image of 600 x 800 pixels) or 30000 data words. In both cases the size of the key is 128 bits and we assume that the same key is used for the entire data block.

The throughput is defined as the number of bits encrypted per the time duration of the clock period. Of course, the actual throughput will depend on the size of the data block processed with the same key. The latency of encryption is defined as the time it takes to obtain the first encrypted word after the moment the first data word is read. Next we show the equations used:

1. $Latency\_{T} = Latency\_{ki} + Latency\_{ci}$
2. $Latency\_{ki} = t\_{conf\_ki} + n\_{prolog\_ki} * t\_{clk\_ki} * iters\_{ki}$
3. $Latency\_{ci} = t\_{conf\_ci} + n\_{prolog\_ci} * t\_{clk\_ci} * iters\_{ci}$
4. $T\_{process\_total} = Latency\_{T} + (t\_{clk\_ci} * iters * S\_{data} / 128)$
5. $Throughput = S\_{data} / T\_{process\_total}$

Where:

n_{prolog_ki} is number of pipeline stages for one iteration of the key expansion. n_{prolog_ci} is number of pipeline stages for one iteration of the data cipher. t_{clk_ki} is the minimal clock period for the key expansion processing part. t_{clk_ci} is minimal clock period for the data cipher processing part.

$iters_{\_ki}$, $iters_{\_ci}$ are the number of iterations for the key expansion and the data cipher processing part respectively. $t_{\_conf\_ki}$, $t_{\_conf\_ci}$ are configuration time for the key expansion and the data cipher processing part respectively. $S_{\_data}$ = 128 bits for the case a and $S_{\_data}$ = 3840000 bits for case b. Taking $S_{\_data}$ of the equation 4 and replacing it in equation 5, the next equation is obtained:

6. $$\text{Throughput} = \frac{S_{\_data}}{m + n * S_{\_data}}$$

Where $m = Latency_{\_T}$; $n = t_{\_clk\_ci} * iters /128$; and iters = 1.

In order to carry out a comparison with other works, CLB-slices was chosen as an area measurement, but for the calculation of the configuration time, CLB were used. 1 CLB = 4 CLB-slices for Virtex-II FPGAs [Xilinx , 05]. The configuration times $t_{\_conf\_ki}$ and $t_{\_conf\_ci}$ are calculated by considering the number of CLBs in the module (*Area*), multiplied by the time to configure a CLB, here the configuration time is 5 μs, [Xilinx , 05].

7 Experimentation and results

We propose two different strategies: IRP-R (Inner-round Pipelining and Reconfiguration) and ORP-R (Outer-round Pipelining and Reconfiguration) to implement the AES algorithm that combine the pipelining with dynamic reconfiguration. For each one of the two strategies, two cases are evaluated. The first one, case a: the number of data to process it is 128 bits and the second one, case b: 3840000 bits. Of course, in both cases the size of the key is 128 bits and we assume that the same key is used for the entire data block, as it was already mentioned in the previous section. Table 1 summarizes this situation.

IRP-R implements the *Expansion_Ki* and *Cipher_ki* partitions, using only iterative loops (the resources of a single round of each module are used and a loop is applied to them, i.e., $n_{\_prolog\_ki} = n_{\_prolog\_ci} = 1$ and $iters_{\_ki} = iters_{\_ci} = 10$). This strategy saves area but reduces the throughput.

ORP-R use full loop unrolling with complete pipelining for both modules (i.e., $n_{\_prolog\_ki} = n_{\_prolog\_ci} = 10$ and $iters_{\_ki} = iters_{\_ci} = 1$). Thus, the throughput is increased, but more area is used. Table 2 shows the obtained results of both implementations after place and route.

The tools used for these implementations were:

- VHDL.
- ISE 8.1i.
- The RC203 board of Celoxica for the evaluation, which include a Xilinx Virtex-II device XC2V3000.
- FTU2 (Celoxica File Transfer Utility) allows you to configure the FPGA on a RC100, RC200 or RC203 board via a parallel port cable.

Architecture	Case study	Size of data (bits)	Size of Key (bits)
IRP-R	a	128	128
IRP-R	b	3840000	128
ORP-R	a	128	128
ORP-R	b	3840000	128

Table 1: Description of IRP-R and ORP-R.

Architecture	Module	Speed Grade	Area				BRAM	$t_{\_clk\_ki}$ (ns)
			Slices	Slice F/F	4 input LUTs	CLBs		
IRP-R	*Expansion_ki*	*4*	618	128	1234	155	0	2,8
	Cipher_ci	*4*	1285	128	2575	322	0	3,0
ORP-R	*Expansion_ki*	*6*	2973	128	5951	744	0	2,7
	Cipher_ci	*6*	11619	128	23236	2905	0	2,9

Table 2: Resources used by IRP-R and ORP-R after place and route.

Architecture	$t_{\_conf\_ki}$ (ms)	$t_{\_conf\_ci}$ (ms)	$Latency_{\_T}$ (ms)	$S_{\_data}$ (bits)	$T_{\_process\_total}$ (ms)	Throughput
IRP-R (a)	0,775	1,6	2,4	128	2,4	53,3 Kbps
IRP-R (b)				3840000	3,3	1.2 Gbps
ORP-R (a)	3,7	14,5	18,2	128	18,2	7 Kbps
ORP-R (b)				3840000	18,3	211 Mbps

Table 3: Obtained results by IRP-R and ORP-R.

The values of throughput obtained by IRP-R and ORP-R, for both case a and case b, are shown in table 3. The results of this table indicate that IRP-R architecture is better than ORP-R architecture for this quantity of data. However, for larger quantities of data to cipher, the throughput of ORP-R architecture is larger than the throughput of IRP-R architecture. This situation can be seen from equation 6, where maximum throughput = $128 / t_{\_clk\_ci}$, when $S_{\_data} \rightarrow \infty$. In this way, the maximum throughput obtained by IRP-R is 4,41 Gbps and 44,1 Gbps by ORP-R. Figure 6 shows this behavior. We can also derive from equation 6, the $S_{\_data}$ value for which the throughput is maximum.

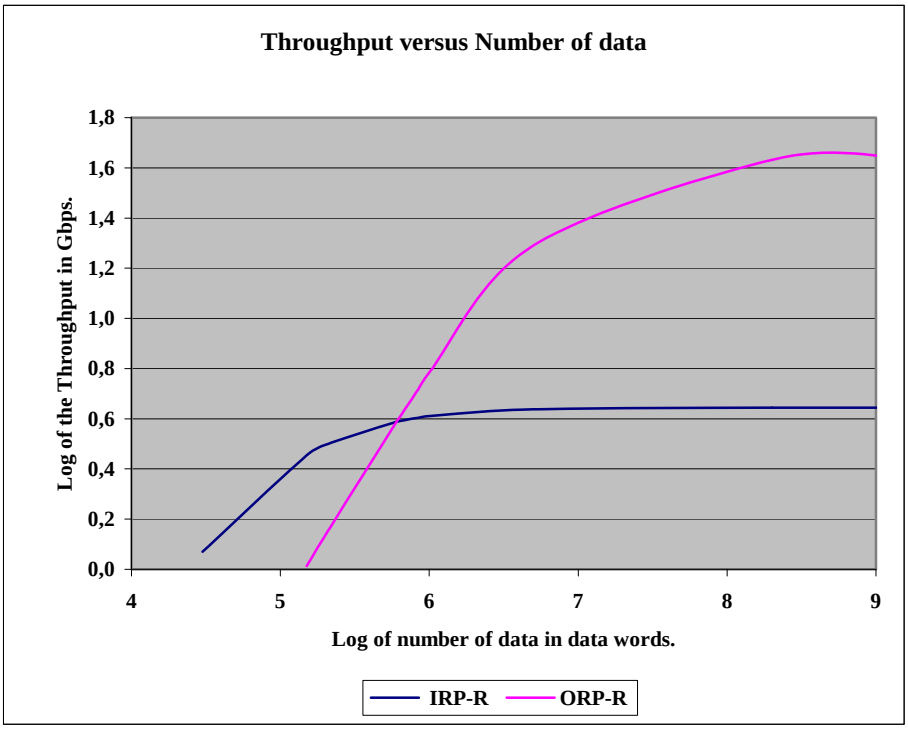

Figure 6: Shows the behaviour of IRP-R and ORP-R.

We can see that the maximum throughput is obtained where the number of data to process is very large by using the same key. Table 4 shows a summary of the obtained results and a comparison with other works done on the AES algorithm. We can see that the FPGA resources are not minimized in our implementation, because the two partitions have very different sizes. If we had cut the algorithm at another boundary (to half) with approximately 7296 slices {(2973 + 11619) / 2, see table 2 ORP-R} in each partition, we could implement it with 37 % less area. But as discussed in section 5, we might need a larger memory plus a reconfiguration for each new data block whether it needs a new key or not. The last point would dramatically reduce the throughput. Thus the compromise between area and performance is acceptable here.

It should be noted that in all cases, i.e. with or without reconfiguration, the initial bitstream loading increases the latency and thus reduces the throughput. Hence, if there is not key change, ORP-R should outperform every other implementation.

Unfortunately, there is always a key change, but the larger the amount of data to encrypt with the same key, the better the efficiency of the run time reconfiguration.

The following results are obtained from table 4. Implementation IRP-R obtains a throughput maximum of 2x-4x compared with those of [Kancharla et al, 03] and [Elbirt et al, 01], needing up to 10 times less resources for its implementation. Nevertheless, it suffers of the time of latency. Implementation ORP-R obtains throughput maximum of 2X compared with the implementation of [Hodjat et al, 04]

(who presents a larger throughput), consuming a 23% of resources of more. Its main disadvantage is the high latency. Like it was to be expected, implementation ORP-R reaches greater throughput than IRP-R.
In the other works, the reconfiguration time is not given. For this reason, we can not calculate their performance in the case studies a and b. The only thing that we can predict is that a key change, during a block encryption, will not affect their throughput because the key is expanded in the same configuration as the cipher (or decryption). Thus, the biggest impact on performance would come from the rate of change between encryption and decryption rather than from a key change.

Design	Device	Slices	T_conf (ms)	Latency (ns)	Mean throughput (Gbps)	Max. throughput (Gbps)
[Kancharla et al, 03]	XC2V6000	14062	?	?	?	1,2
[Elbirt et al, 01]	XC2V1000	10992	?	66	?	1,94
[Standaert et al, 03]	XCV3200E	15112	?	?	?	18,56
[Jarvinen et al, 03]	XC2V2000	10750	?	?	?	17,8
[Hodjat et al, 04]	XC2VP20	9446	?	420	?	21,64
IRP-R case a	XC2V3000	1285	1,6	2400000	0.0000533	-
IRP-R case b					1,2	4,3
ORP-R case a	XC2V3000	11619	15	18200000	0,000007	-
ORP-R case b					0,211	44,1

Table 4: Results obtained and comparison with other works.

There are two main reasons that explain why OPR-R has a higher maximum throughput than the other.

The first is, as stated in section 4, that we do not need to route the sub keys from the expansion part to the cipher part. Indeed, the second configuration localizes these sub keys directly where they are needed by the cipher.

The second is related to the design tool. Indeed, in the second partition, we have only the cipher part, which is mainly a data path. This allows the place and route tool (ISE in our case) to easily implement the operators in a regular pipelined dataflow manner.

8 Conclusions and prospects

By combining the run-time reconfiguration feature of FPGA with the pipeline processing, we can obtain two different architectures. One that uses very low computing resources at the cost of a low throughput and one that can achieve a very high throughput at the cost of a medium amount of resources. The combination of the outer pipeline with dynamic total reconfiguration (ORP-R), offers an interesting alternative for the implementation of the AES algorithm if a considerable amount of data must be ciphered with the same key. Of course, as it is well known, the biggest drawback of the run-time reconfiguration techniques is the high latency time due to the time needed for reconfiguring the FPGA. But with reconfigurable FPGAs associated with algorithm/data that do not need reconfiguration for each data word to process, this can in some cases be lowered to an acceptable level as shown in our

results for the encryption of an entire image with the same key. Current prospects focus on the use of BRAM to reduce both reconfiguration time and resource usage. We also investigate the possibility of hiding the reconfiguration time by interleaving processing and reconfiguration on two areas of the FPGA.

References

[Angel, 00] AES – Advanced Encryption Standard. José de Jesús Angel Angel. http://computacion.cs.cinvestav.mx/~jjangel/.

[Barr, 98] Barr, M., "A Re-configurable Computing Primer," Multimedia Systems Design, September 1998, pp. 44-47.

[Chodowiec et al, 01] P. Chodowiec P., Khuon P., Gaj K., "Fast implementation of secret-key block ciphers using mixed inner-and outer-round pipelining". ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA, FPGA'01 Monterrey, CA, February 11-13, 2001.

[Elbirt et al, 01] Elbirt A.J., Yip W., Chetwynd B., Paar C. "An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists". VLSI Systems, IEEE Transactions on Volume 9, Issue 4, Aug 2001 Pages: 545–557.

[FIPS, 99] "Data Encryption Standard", revised version issued as FIPS 46-3, National Institute of Standards and Technology, 1999.

[Hodjat et al, 04] Hodjat A. and Verbauwhede I.. "A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA". Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines 2004**,** Pages: 308–309.

[Jarvinen et al, 03] Jarvinen et al. "A fully pipelined memory less 17.8 Gbps AES-128 encryptor". International Symposium on Field Programmable Gate Arrays. In 2003 ACM/SIGDA 11th International.

[Kancharla et al, 03] Kancharla P. and Buell D.A., The Advanced Encryption Standard on the HC 36m Reconfigurable Computer, MAPLD 2003".

[McLoone et al, 03] McLoone M. and McCanny J.V., "Rijndael FPGA implementations utilising look-up tables", Journal of VLSI Signal Processing, July 2003, pp. 261-275.

[NIST, 01] Announcing the ADVANCED ENCRYTION STANDARD (AES). Federal Information. Processing Standards Publication 197 AES page available via http://www.nist.gov/CryptoToolkit.

[Patterson et al, 96] Patterson and Hennesy. "Computer organization and design (2nd ed.):The hardware/software interface". Morgan Kaufmann, 1996.

[SRC, 05] Society Reconfigurable Computing. 2005 http://www.srccomp.com/default.htm.

[Standaert et al, 03] Standaert et al, "Efficient Implementation of Rijndael Encryption Reconfigurable Hardware: Improvements and Designs Tradeoffs", CHES 2003 LNCS 2779, pp. 334-350, 2003.

[Tanougast et al, 03] Tanougast C., Berviller Y., Weber S., Brunet P.,"A partitioning methodology that optimizes the area on reconfigurable real-time embedded systems" EURASIP Journal on Applied Signal Processing, Special Issue on Rapid prototyping of DSP Systems April 2003.

[Liu et al, 04] Liu T., Tanougast C., Berviller Y., Weber S., "An Optimised FPGA Implementation of an AES Algorithm for Embedded Applications". Proceeding of the 2004 International Workshop on Applied Reconfigurable Computing, Algarve, Portugal, February 22, 2005.

[Xilinx, 03] Xilinx. Development System Reference Guide. April 30 2003.

[Xilinx, 04] Xilinx. ISE 6 In Depht Tutorial. 2004.

[Xilinx, 05] Xilinx. Virtex-II Platform FPGAs: Complete Data Sheet. 1 March 2005.

[Xilinx, 05]Xilinx. Virtex II Platform User guide. 23 March 2005.