Implementation of AES/Rijndael on a dynamically reconfigurable architecture

Claudio Mucci, Luca Vanzolini
Andrea Lodi, Antonio Deledda, Roberto Guerrieri
ARCES - University of Bologna
Viale Pepoli 3/2, Bologna, Italy

Fabio Campi, Mario Toma
FTM, STMicroelectronics
Viale Olivetti 2, Agrate Brianza (MI)

Abstract

Reconfigurable architectures provide the user the capability to couple performance typical of hardware design with the flexibility of the software. In this paper, we present the design of AES/Rijndael on a dynamically reconfigurable architecture. We will show a performance improvement of three order of magnitude compared to the reference code and up to 24× speed-up figure wrt *fast C implementations over a RISC processor. A maximum throughput of 546 Mbit/sec is achieved. Compared to prior art, we show better energy efficiency with respect to the other programmable solutions, obtaining up to 3 Mbit/sec/mW.*

1. Introduction

Security of data is becoming an important challenge for a wide spectrum of applications, including communication systems (with high privacy requirements), secure storage supports, digital video recorders, smart cards, cellular phones. Resistance against known attacks is one of the main properties that an encryption algorithm needs to provide. When a new attack is demonstrated as effective (also in term of computation time), the update of the encryption system is a real necessity to guarantee the security of data.

In November 2001, the National Institute of Standard Technology (NIST) announced the Advanced Encryption Standard (AES) [1], as a replacement of the Data Encryption Standard (DES). The Rijndael algorithm [2], selected among 15 candidates, is a symmetric key algorithm based on a substitution-permutation network, where most of the calculations are done using Galois Field (GF) arithmetic defined over the field GF(2^8) with the *irreducible polynomial* $x^8+x^4+x^3+x+1$.

Applications requiring high performance and/or low power consumption are today implemented using dedicated hardware accelerators with the downside of higher development costs and lack of flexibility (i.e. algorithm update or parameter changes) with respect to software implementations. In this context, reconfigurable hardware such as Field Programmable Gate Arrays (FPGAs) seems to bridge the gap between performance and flexibility required to guarantee the necessary updates. For complex System-on-Chip, where the area budget dedicated to a single computational island is a constraints, reconfigurable architectures (RAs) for embedded applications were proposed as hardware accelerators, including embedded FPGAs, reconfigurable processors and reconfigurable data-paths.

In this papers we show an implementation of the AES/Rijndael algorithm on the DREAM architecture. The DREAM architecture is composed of a reconfigurable datapath (the 3^{rd} generation Pipelined Configurable Gate Array, or PiCoGA-III) controlled by a 32-bit RISC processor. PiCoGA-III is directly interfaced to a high-bandwidth memory sub-system through programmable address generators, featuring for example vectorized and modulo addressing. An important key point is that the PiCoGA-III features a native support for operations in GF(2^4), thus allowing easy and effective implementations of composite fields that provide the mathematical back-ground for many applications, including Reed-Solomon Codes.

2. Overview: the AES/Rijndael algorithm

The Rijndael algorithm [2] is a symmetric key cipher implementing a substitution-permutation network. The size of both ciphered block and key depends on the security level required, as well as the number of iterations (rounds) required to encrypt the plain-text. As an example, the U.S. Government requires 128-bit keys for SECRET data, while the TOP-SECRET level requires 196 and 256-bit keys. While Rijndael supports a large range of block and key sizes, the NIST standardized a subset of them, using only 128-bit blocks and 128, 196 and 256-bit keys [1]. For ciphering a stream, AES/Rijndael can be applied in many schemes, including ECB (Electronic Codebook) and CBC

The work presented in this paper is done within the MORPHEUS project (IST FP6, project no. 027342), which is sponsored by the European Commission under the 6th Framework program.

978-3-9810801-2-4/DATE07 © 2007 EDAA

(Cipher Block Chaining) [3]. While the EBC mode ciphers each block independently to the other ones, the CBC XORs the plain-text with the previously ciphered block, preventing the coding of equal plain-blocks with equal ciphered-blocks. On one hand, the CBC mode introduces an additional level of security *wrt* EBC, but on the other hand we have an additional feedback that limit the peak performance, especially for hardware implementation.

The encryption process starts arranging the block in a matrix form termed *State*. Let us consider as reference the 128-bit (block and key) Rijndael. In this case, the *State* (S) is a 4×4 array of bytes in which the 128-bit block is arranged by rows. The *State* is thus encrypted by the iterative application of 4 operations, as described in the following pseudo-code.

```
S=in; Nb = 128;
S=AddRoundKey(S, key[0,Nb-1]);
for (i=1; i<Nround; i++) {
  S = SubBytes(S);
  S = ShiftRows(S);
  S = MixColumns(S);
  S = AddRoundKey(S,key[i*Nb,(i+1)*Nb]);
}
S = SubBytes(S);
S = ShiftRows(S);
S = AddRoundKey(S,key[i*Nb,(i+1)*Nb]);
out = S;
```

The number of iteration (Round) depend on the key size, and ranges from 10 to 14. Four basic operations are applied to the *State*:

SubBytes: is a non-linear substitution step applied to each byte of the *State* array, that is substituted with its inverse multiplicative over GF(2^8). Then, an affine transformation ($a' = M \times a + c$) is applied, as described by the following equivalent equation:

$$a'_i = a_i + a_{(i+4)\ mod\ 8} + a_{(i+5)\ mod\ 8} + a_{(i+6)\ mod\ 8} + a_{(i+7)\ mod\ 8} + c_i \quad (1)$$

where a' and a are bytes of the *State* array, c is the vector (01100011). The non-linear substitution applied to each byte is also known as S-Box.

ShiftRows: operates on the rows of the *State*, rotating them to the left by a shift step equal to the row index.

MixColumns: operates on the four bytes of each column of the *State* array, that are treated as the coefficient of a 4-th order polynomial over GF(2^8). The MixColumns step performs a multiplication (modulo $x^4 + 1$) with the fixed polynomial $3x^3 + x^2 + x + 2$.

AddRoundKey: represents the last operation of each Round and performs an addition over GF(2^8) between the *State* and the *Round Key*, a 4×4 array generated from the original key by an expansion step in order to provide different key-words to different rounds.

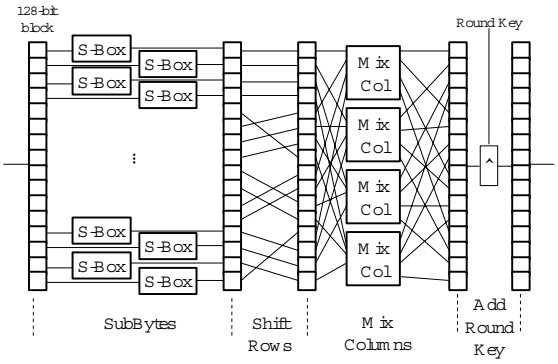

Figure 1. Common AES-Round block diagram

The key expansion step, also known as *Key Schedule*, is performed before the encryption, and is described with mathematical operations, mainly based on the application of S-Box and word rotation [1, 2].

All the operations previously described are invertible in a very straightforward manner, resulting a decoding schema very similar to the encoding one. In particular, the computational complexity is more or less the same, since the kind of applied operations is the same.

3. Related work

The Advanced Encryption Standard implemented by the Rijndael algorithm can be efficiently implemented in both software and hardware. 8-bit processors can directly implement most of the operations required by AES since they are natively working on 8-bit variables (e.g. ShiftRows, AddRoundKey and MixColumns), while the S-Box is more efficiently implemented using a 256-entry 8-bit hash table. 32-bit processors implement fast Rijndael combining the different step of a round transformation in a single set of hash-tables. As a result, 4 tables with 256 32-bit values (termed T-Box) substitute most of the round operations, leaving to the dynamic computation XORs and rotations [2]. Comparing this optimized version with the basic one, about one order of magnitude in performance is gained on a RISC processor. Implementations on TI DSPs are discussed in [6]: a 112.3 Mbit/sec throughput (@ 200MHz) is achieved on the C62x architecture for the encoder, 1.6× faster than a Pentium-Pro working at the same frequency. Moreover, instruction set extensions dedicated to Rijndael are present in the literature, such as [4, 5].

Hardware implementations of AES are optimized by the exploitation of the available parallelism. Hence, the design of hardware accelerators for AES begins from the 1-to-1 unfolding of the Round definition, as shown in Figure 1. For the ECB mode, the Rijndael algorithm can be completely unrolled and pipelined, thus improving the available throughput up to the technological limit. The undeniable

drawback is the considerable augment in area occupation. Examples of AES implementations for stand-alone FPGAs are [7–11], providing 2-30 GBit/sec throughputs. Hybrid solutions, coupling a processor with FPGA technology, are implemented in the Xilinx Virtex II Pro platform [8, 12], achieving performance up to 1.2 GBit/sec. For embedded applications, where the area budget is a constraints, devices with restricted size are proposed. Embedded FPGAs (e.g. [14]) are the most direct "translation" of the traditional field-programmable technology to the market of IPs suitable for SoC integration. Alternatively, and depending on the application field, reconfigurable data-paths (e.g. [15, 16]) are used as hardware-programmable accelerators. As an example, in [17] a reconfigurable datapath challenges a set of cryptographic applications.

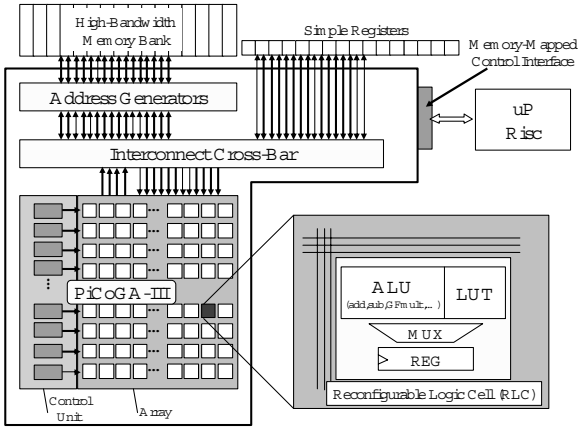

Figure 2. Simplified DREAM architecture

4. DREAM Architecture

DREAM architecture [18] is a dynamically reconfigurable platform coupling the PiCoGA-III reconfigurable device with a RISC processor using a loosely-coupled memory mapped co-processor schema. A high bandwidth memory sub-system provides/receives data to/from PiCoGA-III allowing one to either maximize the throughput and interface the DREAM architecture with for example external computational blocks. Figure 2 shows the simplified DREAM block diagram.

The processor, a 32-bit RISC core with 4+4Kbyte of data/instruction memory, is responsible of DREAM management, although it could be also used to implement standard code, such as the control part of an application. The high-bandwidth memory sub-system is composed of 16 4Kbyte 32-bit memory banks, each of them accessed independently to the other ones by programmable address generators. A fully-populated interconnect cross-bar allows the user to modify the connection with PiCoGA-III I/Os (12 32-bit inputs and 4 32-bit output). Furthermore, an additional simple 32-bit register file is provided. This memory sub-system represent an evolution of that one implemented in [19]. In particular, we have introduced the capability of handling power-of-2 modulo addressing, with respect to standard step/stride addressing modes. A 64-entry configuration cache is provided for the interconnect, allowing to switch among different connection topologies without any additional overheads, while the same is not provided for the address generators.

PiCoGA-III is the 3^{rd} generation of the Pipelined Configurable Gate Array (PiCoGA) architecture described in [20]. The current version is implemented in 90nm STM technology (area $\sim$11mm^2, frequency 200MHz). PiCoGA-III is composed of an island-style 16x24 array of Reconfigurable Logic Cells (RLCs) implementing the computational part of the logic, and a Row-based configurable Control Unit responsible of the pipeline evolution under a dataflow paradigm. Each RLC provides a 64-bit Look-Up Table (LUT) capable of supporting many configuration schemes, including 6×1 and 4×4, and a 4-bit ALU, including adder and Galois Field multiplier over GF(2^4). Four contexts of configuration can be stored in the PiCoGA-III, as in an internal configuration cache, and the configuration switch requires only 1 cycle.

Operations on the PiCoGA (or PGAOPs) are described using Griffy-C, a C subset featuring a single-assignment manually-dismantled sequential form [21]. Extra instructions, such as the GF(2^4) multiplication, can be referred by means of *built-in* functions (e.g. *out = GFmult(a,b);*). Extensions to the basic C language (by `#pragma` directives) are introduced to resize at bit-level the size of each variable. Griffy-C, and the corresponding tool-chain, was designed to provide the user a C-based algorithm development environment, where standard C operators and *built-in* functions can be mapped 1-to-1 into PiCoGA. The developer describes the PGAOP using a fully sequential code, while tools are responsible of the pipeline organization, through an automatic scheduling phase (based on ASAP policy) capable to optimize *routing-only* operations (e.g. shift with constant amount). Optimizations of the pipeline structure (e.g. tree balancing) can be made in Griffy-C by the insertion of retiming registers.

5. Implementation of basic GF(2^8) operations

An important property of Galois Fields is that they are univocally defined by the number of elements. What can be changed, depending on the *irreducible polynomial*, is the representation. Therefore, the GFs are isomorphic with respect to an *irreducible polynomial* change and a transformation matrix can be defined in order to change the representation. As described in Paar' PhD Thesis [22],

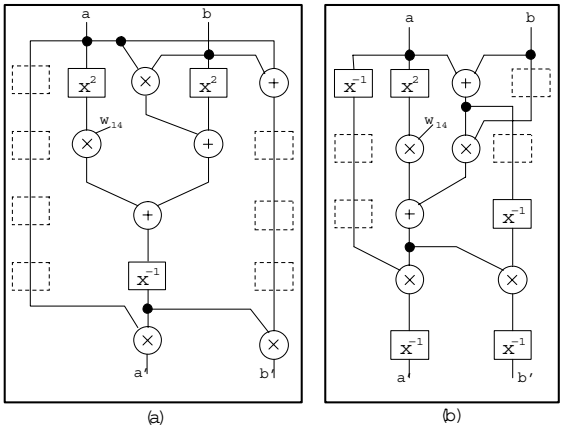

Figure 3. Inverse multiplicative on composite fields schemes

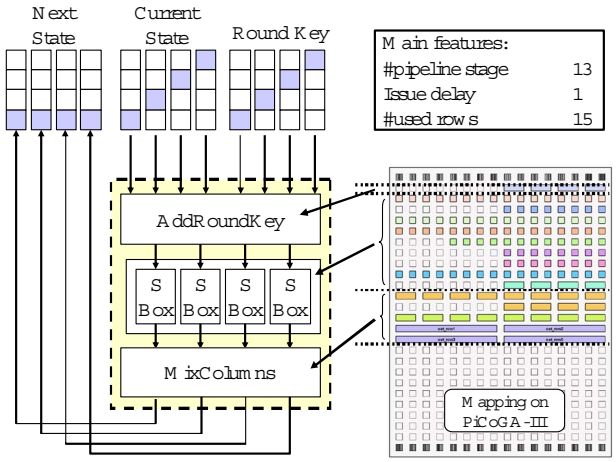

Figure 4. AES/Rijndael selected kernel and implementation

this implies that GF(2^8) can be seen as a composite field GF((2^4)^2) whose elements are represented by 1-order polynomials $\alpha x + \beta$ with $\alpha, \beta \in$ GF(2^4). PiCoGA-III features a native support of GF(2^4) with the *irreducible polynomial* x^4+x+1. This means that each RLC can be programmed to perform both the *sum* ($\oplus$) operation, implemented by LUT as a 4-bit XOR, and the *multiplication* ($\otimes$) operation, implemented by the dedicated GF multiplier.

The AES/Rijndael algorithm requires to implement three operations on GF(2^8): the *sum*, the multiplication by constant amount, and the inverse multiplicative. While the *sum* and the multiplication with constant amount can be described (in Griffy-C) and implemented (on the PiCoGA) with standard C (XORs, ANDs and shifts), the implementation of the inverse multiplicative over GF(2^8) benefits from the GF capabilities of PiCoGA-III. By definition [23], the inverse multiplicative on the composite field GF((2^4)^2) (using the *irreducible* $x^2 + x + \omega_{14}$) is:

$$\begin{aligned}(\alpha x + \beta)^{-1} &= \alpha \otimes \theta^{-1} x + (\alpha \oplus \beta)\theta^{-1} \qquad (2)\\ \theta &= \alpha^2 \otimes \omega_{14} \oplus \alpha \otimes \beta \oplus \beta^2\end{aligned}$$

Figure 3(a) shows the straightforward implementation of the inverse multiplicative obtained from equation (2). Basic blocks are aligned per pipeline stage, and each basic block can be mapped on one RLC (the inverse on GF(2^4) is a 4-in 4-out function implemented by LUT). The full retiming, needed to maximize the throughput, requires 7 additional registers (dashed-line blocks), for a total of 17 RLCs distributed over 5 rows. Figure 3(b) shows an optimized inverse multiplicative generated by re-writing the equation (2) in the following form:

$$\begin{aligned}(\alpha x + \beta)^{-1} &= (\alpha^{-1} \otimes \delta)^{-1} x + ((\alpha \oplus \beta)^{-1} \otimes \delta)^{-1}\\ \delta &= \alpha^2 \otimes \omega_{14} \oplus \beta \otimes (\alpha \oplus \beta) \qquad (3)\end{aligned}$$

In this second case we have an issue-delay of 2 cycles, requiring only 4 additional registers (for a total of 15 RLCs) for the full retiming. The max-width of this implementation schema is 4 RLCs, allowing a better packing of multiple instances of the inverse multiplier in the PiCoGA rows (each of them composed by 16 RLCs). To complete an S-Box, we need to add the isomorphism matrix and the successive affine transformation. Two rows with respectively 4 and 2 RLCs are required for the input isomorphism, while the output isomorphism and the affine transformation can be collapsed together, with the same resources occupation (4+2 RLCs).

6. Implementation of AES/Rijndael

A goal of our AES/Rijndael implementation is to be flexible for both block and key size. Hence, we have analyzed, in relation with DREAM capabilities, the following properties of Rijndael algorithm. First of all, since the SubBytes operation does not depend on the position of each byte, the ShiftRows can be performed before the SubBytes. In addition, ShiftRows performs a rotation which can be implemented using modulo addressing. Hence, using different memory banks for storing the different rows of the *State* matrix, PiCoGA is able to load a new *State* column for each cycle. The rotation applied by ShiftRows is handled by changing the starting address of each bank, while the different number of columns (for the generic Rijndael) is handled by setting the address generator end-of-count. The organization by column allows the packing of the MixColumns function in the same PiCoGA operations.

Figure 4 shows the corresponding implementation scheme. This PGAOP performs AddRoundKey, SubBytes and MixColumns for the 4 bytes in a column concurrently, leaving the addressing engine to handle the ShiftRows for both block and key access. A different set of buffers is used

	Clock cycles per 1 block		
block/key size	Scalable Version	Optimized Version	Key Expansion
128/128	408	285	192
128/192	466	329	216
128/256	524	373	240
256/128	455	-	319
256/192	521	-	367
256/256	587	-	415

Table 1. AES/Rijndael encoder performance

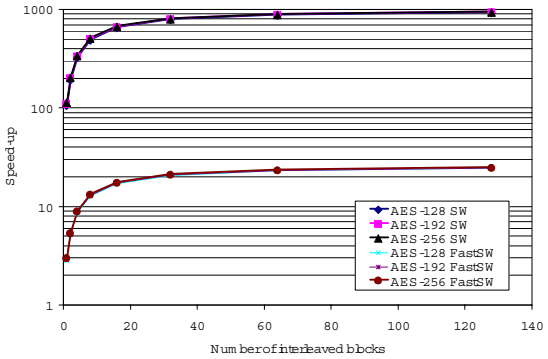

Figure 5. Speed-ups *wrt* RISC processor

to store PGAOP results, since it is not possible to read-and-write a memory bank in the same cycle. This implementation requires 4 PGAOP call in order to accomplish one AES/Rijndael Round, after that we need to re-configure the interconnect cross-bar in order to swap the used I/O buffers. Although this operation could be performed in parallel to the PGAOP computation (destination port are stored internally to the PiCoGA during the PGAOP triggering), this reconfiguration break the best pipeline evolution. For the EBC mode, there is not dependency among the encryption of successive blocks, thus it is possible to interleave the encryption of more than one block in order to mitigate the impact of the interconnect reconfiguration. The stride factor allows the address generator to jump to the next block when the Round is finished. The last Round requires the implementation of a dedicated PGAOP, without MixColumns and within an additional AddRoundKey before the SubBytes needed by the loop transformation introduced before. Only 11 pipeline stages are required for this goal, but the area occupation is increased to 17 rows because of an unfavorable requirement of additional retiming registers necessary to maintain the issue-delay equal to 1.

For 128-bit block only, the PiCoGA-III is able to output a whole 4x4 block, then it is possible to implement an optimized PGAOP using only the simple registers. When blocks interleaving is not applicable (e.g. in CBC mode), we can achieve a further 1.4× speed-up reducing the configuration overhead, through the utilization of simple registers instead of address generators to exchange data with the PiCoGA. Two additional shift registers (and the corresponding control logic) shall be mapped on the PGAOP because the ShiftRows requires to be implemented internally. Data are loaded at the first PGAOP trigger, while other 3 three additional triggers are required to provide the correct result.

7. Experimental results and comparisons

We have implemented the AES/Rijndael algorithm on the DREAM cycle-accurate Instruction Set Simulator (ISS), based on CoWare technology. The RISC processor is modeled using LISA language, while the memory sub-system and the PiCoGA are modeled using a mix of SystemC and C/C++. Frequency and power consumption figures are estimated starting from measurement on the silicon prototype in [19], featuring a comparable design complexity. Both scalable and optimized implementations presented in the previous section were considered in our analysis and the cycle count obtained is reported in Table 1. Results are provided for the encryption of a single block, considering various block and key sizes. At the frequency of 200MHz, it is possible to achieve a throughput up to 90Mbit/sec using a scheme appliable in both EBC and CBC modes.

In EBC mode, the scalable solution can interleave the encryption of more than one blocks, exploiting as much as possible the computational efficiency of DREAM. Pipelining the computation on the PiCoGA-III, the obtained speed-up figures raises from 100× to 930× *wrt* the ANSI-C Reference Code (v. 2.2) running on a RISC processor at the same frequency, while it raises from 3× to 24× *wrt* a fast software implementation (by C. Devine, on-line available at the Rijndael Home Page [2]) working on the same RISC processor. Figure 5 shows the achieved speed-ups *versus* the level of interleaving applied, hence in relation to the number of block concurrently elaborated.

Figure 6 shows an analysis of the throughput with respect to the interleaving factor applied. As a consequence, ciphering 64 or 128 blocks, the benefit of pipelining the computation inside the PiCoGA-III mitigates the overhead due to interconnect configuration changes, allowing one to obtain up to 546 Mbit/sec of throughput. Considering the case of AES-128, the throughput increases from 63 to 546 Mbit/sec in a way that is proportional to the average number of active rows inside the PiCoGA. In fact, the average number of active rows growth from 1.5 rows/cycle to 12.8 rows/cycle, respectively corresponding to 10% and 85% of the PGAOP. With 256-bit block size, the memory utilization growths faster, then the 128-block interleaving cannot be applied.

Comparisons with other AES-128 implementations are reported in Table 2, including both fast software (with an as-

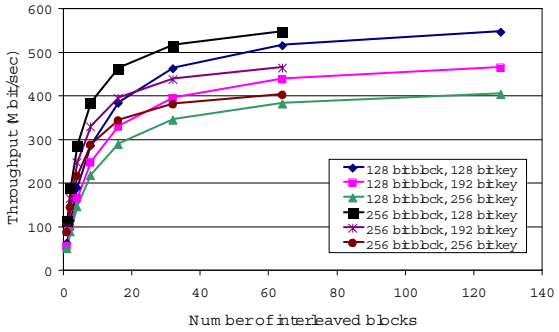

Figure 6. Throughput vs. interleaving factor

sembly handcoded Pentium-III) and hardware approaches. Furthermore, a processor with custom-designed ISA [4] is considered too. For the hardware approaches, we have taken into account folded schemes implemented on both FPGA and ASIC (0.18μm) prototype. The energy efficiency (Mbit/sec/mW) shows the density advantage of DREAM with respect to the other "programmable" solutions. For this purpose, the power consumption of DREAM is estimated in a range from 80 mW (CBC) to 180 mW (EBC), depending on the different PiCoGA-III utilization and correlated memory activity.

8. Conclusions

In this paper we have presented an implementation of AES/Rijndael (supporting different block and key sizes) on a dynamically reconfigurable architecture, achieving up to 940× speed-up with respect to the reference code and up to 24× speed-up *wrt* fast C code implemented on a RISC processor. Powered by the hardware efficiency of a dynamically reconfigurable data-path, our implementation obtain a throughput of 546 Mbit/sec in the best case, while it shows better energy efficiency figures (up to 3.03 Mbit/sec/mW) with respect to other "programmable" solutions, including FPGA-based folded implementations.

	Frequency MHz	Throughput Mbit/sec	Energy eff. Mbit/sec/mW
DREAM (EBC)	200	546	3.03
DREAM (CBC)	200	90	1.12
ARM9$^{(1)}$ [24]	500	46.6	0.32$^{(1)}$
ARM9$^{(2)}$ [24]	250	23.3	0.67$^{(2)}$
TI C62x [6]	200	112	n/a
Pentium-III [13]	1130	645	0.015
Ravi [4]	188	17.2	n/a
Lu [12]	196	1197	n/a
Chaves [8]	100	1258	n/a
Sch. [13] FPGA	77	640	0.39
Sch. [13] ASIC	154	1280	22.8

(1) ARM926EJ-S Speed-Opt. 90nm 0.29 mW/MHz (www.arm.com)
(2) ARM926EJ-S Area-Opt. 90nm 0.14 mW/MHz (www.arm.com)

Table 2. AES-128 encryption comparisons

References

[1] NIST *Specification for the ADVANCED ENCRYPTION STANDARD (AES)*, FIPS PUBS 197, November 26, 2001.
[2] J. Daemen and V. Rijmen *AES Proposal: Rijndael*, NIST AES Proposal, www.esat.kuleuven.ac.be/˜rijmen/rijndael/.
[3] B. Schneier, *Applied Cryptography*, 2nd ed. John Wiley and Sons. New York, NY, 1996.
[4] S. Ravi et al. *System Design Methodologies for a Wireless Security Processing Platform*, Proceedings on the DAC 2002.
[5] S. Tillich et al. *An Instruction Set Extension for Fast and Memory-Efficient AES Implementation*, Communications and Multimedia Security, Springer Verlag, 2005.
[6] T. Wollinger et al. *How well Are High-End DSPs Suited for the AES Algorithms?* NIST AES-3 Conference, 2000.
[7] J. Zambreno et al. *Exploring Area/Delay Tradeoffs in an AES FPGA Implementation*, FPL 2004.
[8] R. Chaves et al. *Reconfigurable Memory Based AES Co-Processor*, Proceedings of the RAW, April 2006
[9] HELION. *High Performance AES (Rijndael) cores for Xilinx FPGA*, http://www.heliontech.com
[10] A. Wiesmaier, *The State of the Art in Algorithmic Encryption (2006)*, citeseer.ist.psu.edu/wiesmaier06state.html
[11] A. Hodjat and I. Verbauwhede *A 21.54 Gbit/s fully pipelined AES processor on FPGA*, FCCM 2004.
[12] J. Lu, J. Lockwood *IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application*, RAW 2005
[13] P.R. Schaumont, H. Kuo, I. Verbauwhede *Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor*, DAC 2002.
[14] M2000 *Embedded FPGA* www.m2000.fr
[15] M. Vorbach, J. Becker, *Reconfigurable processor architectures for mobile phones* Proceedings on the IPDPS, 2003.
[16] H. Singh et al. *MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications*, IEEE Transactions on Computers, May 2000.
[17] R.R. Taylor, S.C. Goldstein *A High-Performance Flexible Architecture for Cryptography*, CHES 1999.
[18] F. Campi et al. *A dynamically adaptive DSP for heterogeneous reconfigurable platforms*, DATE 2007.
[19] F. Campi et al. *A Stream Register File Unit for Reconfigurable Processors*, ISCAS 2006.
[20] A. Lodi et al. *A VLIW processor with reconfigurable instruction set for embedded applications*, IEEE Journal of Solid-State Circuit, Nov. 2003.
[21] C. Mucci et al. *A C-based Algorithm Development Flow for a Reconfigurable Processor Architecture*, IEEE International Symposium on System on Chip, November 2003.
[22] C. Paar *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*, Ph.D. Thesis, 1994.
[23] V. Rijmen *Efficient Implementation of the Rijndael S-box*,
[24] G. Bertoni et al. *Efficient Software Implementation of AES on 32-Bit Platforms*, CHES 2002.