IMPROVEMENT OF AN EFFICIENT AES IMPLEMENTATIONS FOR ARM BASED PROCESSOR

Dr. S. Kishore Reddy*

Dr. Syed Musthak Ahmed**

A. Ravi Shankar***

ABSTRACT

The Advanced Encryption Standard (AES) contest, started by the U.S. National Institute of Standards and Technology (NIST), saw the Rijndael [13] algorithm as its winner [11]. Although the AES is fully defined in terms of functionality, it requires best exploitation of architectural parameters in order to reach the optimum performance on specific architectures. Our work concentrates on ARM cores [1] widely used in the embedded industry. Most promising implementation choices for the common ARM Instruction Set Architecture (ISA) are identified, and a new implementation for the linear mixing layer is proposed. The performance improvement over current implementations is demonstrated by a case study on the Intel StrongARM SA-1110 Micropro-cessor [2]. Further improvements based on exploitation of memory hierarchies are also described, and the correspond-ing performance figures are presented.

Keywords: *AES, ARM microprocessor, code optimisation, cache memories.*

* Associate Professor, ECE, SR Engineering College, Warangal, AP, India

** Professor and HOD, SR Engineering College, Warangal, AP, India

*** Assistant Professor, ECE, SITS, Khammam, AP, India

1. INTRODUCTION

The AES, developed to replace The old Data Encryption Standard (DES), is the result of a four-year effort involving the cooperation between the U.S. Government, private industry and academia from around the world. The call for algorithms made by NIST required that algorithms must be based on symmetric key cryptography and work as a block cipher. Among fifteen candidate algorithms, Rijn-dael's combination of security, performance, efficiency, ease of implementation and flexibility made it the most appropriate choice for NIST as the Advanced Encryption Standard.

With the increasing use of portable and wireless devices in the business and daily life, protecting sensitive information via encryption is becoming more and more crucial. High-performance, low-power ARM processor cores are now li-censed by many major semiconductor partners and widely used in smart cards, portable communication devices and by consumer electronics industry. Our work aims in fact at performing a comprehensive exploration of the solution space for efficient software implementations of AES on ARM processors. Efficient software implementations of AES pro-posed so far are listed in [13] [17] [12]. However, in this work, the peculiarities of the common ARM ISA suited for AES are identified and appropriately exploited, also allowing us to design a new implementation, which exhibits interesting features with respect to the known ones.

This paper is organised as follows: Sect. 2 describes the Rijndael algorithm. Sect. 3 presents the ARM ISA and the related optimisations. Sect. 4 presents Intel StrongARM SA-1110 Microprocessor architecture, real performance fig-ures on this processor, together with improvements due to ISA related optimisations, and improvements due to ex-ploitation of memory hierarchies and cache locking mech-anisms. Finally Sect. 5 summarizes our results.

2. THE RIJNDAEL ALGORITHM

Rijndael is an iterated block cipher with a variable block length and key length [14] [15]. The block length and the key length can be independently set to 128, 192 or 256 bits, whereas AES restricts the block length to 128 bits only [16].

The number of round transformations to be applied on the input blocks is either 10, 12 or 14, depending on the key length. The round transformation is composed of three dis-tinct invertible transformations, called layers: (1) the non-linear layer, (2) the linear mixing layer, and (3) the key ad-dition layer. The different transformations operate on the intermediate result called the State. The State can be con-sidered as a two dimensional array of bytes. The array has four rows, and the number of columns depend on the block length. The nonlinear layer makes a nonlinear byte substi-tution on each of the State bytes. The linear mixing layer rotates the rows of the state array over different offsets, and applies a linear transformation called the MixColumn trans-formation on each column of the State. The key addition layer simply XORs State bytes with subkey bytes that are generated after a keyschedule.

All the operations described above, with the exception of the MixColumn transformation, are quite straightforward to implement, therefore we will mainly concentrate on imple-mentation of the MixColumn transformation in this paper. This is also justified by the fact that in compact implementa-tions the MixColumn transformation is responsible for about 50% of the encryption time.

2.1 The MixColumn Transformation

The MixColumn transformation makes use of arithmetic operations in the finite field GF(2^n).

Table 1: Look-up tables used by different versions

Version	Encryption	Decryption
V1	Sbox	InvSbox
V1T	Sbox	InvSbox
V2	Sbox + Enc. table	InvSbox + Dec. table

Table 2: Sizes of the look-up tables in bytes

Look-up table	Size in bytes
Sbox	256
InvSbox	256
Enc. table	1K
Dec. table	1K

We assume that the reader has a basic background of Galois Fields, but for com-pleteness we recall that addition in GF(2^n) is equivalent to a simple bitwise XOR, while multiplication is obtained by reducing the result of standard multiplication (with XOR as sum) modulo a fixed polynomial. This polynomial must be irreducible to preserve the algebraic structure of field. In the MixColumn transformation, each column of the State is considered as a polynomial with coefficients in GF (2^8), and multiplied modulo $x^4 + 1$ with a fixed polynomial $\{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, coprime to the modulo. Assum-ing that the column before transformation consists of the bytes (b_0 , b_1, b_2 , b_3), each byte representing a polynomial in GF (2^8), the transformed column bytes (c_0, c_1, c_2, c_3) are computed as follows:

$$\begin{array}{l}
c_0 = \{02\} {\scriptstyle 3}\, b_0 \oplus \{03\} {\scriptstyle 3}\, b_1 \oplus \{01\} {\scriptstyle 3}\, b_2 \oplus \{01\} {\scriptstyle 3}\, b_3 \\
c_1 = \{01\} {\scriptstyle 3}\, b_0 \oplus \{02\} {\scriptstyle 3}\, b_1 \oplus \{03\} {\scriptstyle 3}\, b_2 \oplus \{01\} {\scriptstyle 3}\, b_3 \\
c_2 = \{01\} {\scriptstyle 3}\, b_0 \oplus \{01\} {\scriptstyle 3}\, b_1 \oplus \{02\} {\scriptstyle 3}\, b_2 \oplus \{03\} {\scriptstyle 3}\, b_3 \\
c_3 = \{03\} {\scriptstyle 3}\, b_0 \oplus \{01\} {\scriptstyle 3}\, b_1 \oplus \{01\} {\scriptstyle 3}\, b_2 \oplus \{02\} {\scriptstyle 3}\, b_3
\end{array} \quad (1)$$

where ₃ denotes polynomial multiplication in GF (2^8) de-fined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, and ⊕ denotes simple XOR at byte level. Multiplication by {02} in GF (2^8) can be implemented at byte level with a left shift followed by a conditional bitwise XOR with {1b}. Mul-tiplication by larger coefficients can be implemented with repeated multiplications by {02} and XORs with previously calculated results.

2.2 32-bit Implementation Strategies

On 32-bit processors byte-level operations can be done in parallel. In particular, the bytes within a column can be multiplied by {02} in parallel, and we will refer to this operation as the Xtime operation. Several efficient imple-mentations of the Xtime operation have been described by Brian Gladman in [17].

The original Rijndael submission defines two main imple-mentation strategies. The first strategy, which we will call Version 1 (V1), employs byte substitution tables only (Sbox and InvSbox). The second strategy, which we will call Ver-sion 2 (V2), employs additional encryption/decryption ta-bles in order to avoid the software computation of (1). Ver-sion 1 implements the nonlinear layer using byte substitu-tion tables and implements the MixColumn transformation making repetitive calls to the Xtime operation. Version 2, on the other hand, combines the nonlinear layer, and the linear mixing layer within 4 table lookups, 3 XORs and 3 rotates allowing very fast implementations on 32 bit processors.

The third strategy proposed recently by Bertoni et al. in [12], like Version 1, employs the nonlinear byte substitution tables only, however transposes the state array before ap-plying the round transformations, and transposes again at the end. In this way, considerable amount of computation is **saved in the implementation of the MixColumn and Inverse** MixColumn (InvMixColumn) transformations, resulting in higher performance than V1 in general but maintaining the same low memory requirements. We will refer to this strat-egy as the Transposed State Version (V1T).

The look-up tables used by the three described strategies are given in Table 1, and the sizes of the different look-up tables are given in Table 2. The use of pre-rotated encryp-tion and decryption tables, and word extended byte sub-stitution tables has also been proposed, respectively in the Rijndael submission and in [17] by Brian Gladman. These optimisations can eliminate a number of shifts and rotates on some architectures, and can be considered among other implementation strategies.

3. OPTIMISATIONS FOR ARM PROCES-SORS

ARM is the leading provider of 32-bit embedded RISC mi-croprocessors with almost 75% of the market. ARM offers a wide range of processor cores based on a common architec-ture [9] [4], delivering high performance together with low power consumption and system cost.

ARM processors implement a load/store architecture. De-pending on the processor mode, 15 general purpose registers are visible at a time. Almost all ARM instructions can be executed conditionally on the value of the ALU status flags. Load and store instructions can load or store a 32-bit word or an 8-bit unsigned byte from memory to a register or from a register to memory.

The ARM arithmetic logic unit has a 32-bit barrel shifter that is capable of shift and rotate operations. The sec-ond operand to all ARM data-processing and single register data-transfer instructions can be shifted before data process-ing or data transfer is executed, as part of the instruction. The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register. When the shift amount is specified in the instruction, it may take any value from 0 to 31, without incurring any penalty in the instruction cycle time.

Use of word extended substitution tables in Rijndael im-plementations is unnecessary and inefficient on ARM proces-sors, since the architecture supports load byte instructions. Use of pre-rotated tables cannot improve the performance neither, since the barrel shifter that can be combined with data processing instructions reduces the effective cost of ro-tate instructions to zero. Use of such tables, in fact, in-creases the register pressure and possibility of cache misses, therefore degrading the performance. We will consider only V1, V2 and V1T described in Sect. 2.2 in the rest of this paper.

3.1 The Proposed Mix Column Implementation

The MixColumn implementation described by Gladman [17] in V1 requires 4 XORs, 3 rotates and one Xtime opera-tion, incurring 16 XORs, 12 rotates and 4 Xtime operations per AES round.

V1T by Bertoni et al. [12] eliminates the rotations, and requires 16 XORs and 4 Xtime operations per AES round. In fact, the advantage of using a transposed state is more evident in the decryption operation, because the InvMixCol-umn operation sees an important reduction in the number of XORs and Xtime operations.

We describe here a new MixColumn implementation that requires 3 XORs, 3 rotations and one Xtime, incurring 12 XORs, 12 rotations and 4 Xtime operations per AES round. However, using the ARM barrel shifter, the 12 rotations can be combined with 12 XORs without any penalty, resulting in 12 XORs and 4 Xtime operations effectively per round. The proposed MixColumn implementation, in addition to cutting down the number of logical operations, can support all block lengths multiples of 32-bits, unlike the Transposed State Version, which requires a State matrix of 128-bits.

Assuming that b = (b_0, b_1 , b_2, b_3) is the input column to be transformed, s and t are two 32-bit temporary variables, and c = (c_0 , c_1 , c_2 , c_3) is the result, the four steps of the MixColumn transformation are given as follows, where EOR is the ARM instruction for XOR, and ROL is the rotate left command for the barrel shifter:

1. **EOR s, b, b ROL 8**
   $s_0 = b_0 \oplus b_1$, $s_1 = b_1 \oplus b_2$
   $s_2 = b_2 \oplus b_3$, $s_3 = b_3 \oplus b_0$

2. **EOR t, s, b ROL 16**
   $t_0 = b_0 \oplus b_1 \oplus b_2$ $t_1 = b_1 \oplus b_2 \oplus b_3$
   $t_2 = b_2 \oplus b_3 \oplus b_0$ $t_3 = b_3 \oplus b_0 \oplus b_1$

3. **s = Xtime(s)**
   $s_0 = \{02\}_3 (b_0 \oplus b_1)$ $s_1 = \{02\}_3 (b_1 \oplus b_2)$
   $s_2 = \{02\}_3 (b_2 \oplus b_3)$ $s_3 = \{02\}_3 (b_3 \oplus b_0)$

4. **EOR c, s, t ROL 8**
   $c_0 = \{02\}_3 (b_0 \oplus b_1) \oplus b_1 \oplus b_2 \oplus b_3$ $c_1 = \{02\}_3 (b_1 \oplus b_2) \oplus b_2 \oplus b_3 \oplus b_0$ $c_2 = \{02\}_3 (b_2 \oplus b_3) \oplus b_3 \oplus b_0 \oplus b_1$ $c_3 = \{02\}_3 (b_3 \oplus b_0) \oplus b_0 \oplus b_1 \oplus b_2$

The final result is equivalent to (1).

4. A CASE STUDY ON INTEL STRONG-ARM SA-1110 MICROPROCESSOR

The Intel StrongARM SA-1110 Microprocessor (SA-1110) is a highly integrated 32-bit RISC microprocessor that incor-porates Intel design and process technology along with the power efficiency of the ARM architecture [10] [7]. The SA-1110 implements the ARM V4 ISA, together with a Harvard architecture memory system. There are separate instruc-tion and data caches. The instruction and data streams are translated through independent memory-management units (MMUs) [6]. The 16 Kbytes Icache has 512 lines of 32 bytes (8 words), arranged as a 32-way set associative cache. There are two logically separate data caches: the main data cache and the mini data cache (minicache). The main data cache, an 8 Kbyte write-back Dcache, has 256 lines of 32 bytes (8 words) in a 32-way set-associative organization.

Table 3: Number of clock cycles, 128-bit key length

Version	Enc. cycles	Dec. cycles
V1	1020	1605
V1T	1033	1312
Proposed	943	1605

The mini-cache provides a smaller and logically separate data cache that can be used to enhance caching performance. The mini-cache is a 512-byte write-back cache having 16 lines of 32 bytes (8 words) in a two-way set-associative organization. The Dcaches are accessed in parallel and the design ensures that a particular line entry will exist in only one of the two at any time. Both Dcaches use the virtual address generated by the processor and allocate only on loads according to the set-tings of certain bits in the MMU translation tables. Stores are made using a four-line write buffer. The performance of specialized load routines is enhanced with the four-entry read buffer that can be used to prefetch data. The Stron-gARM pipeline has 5 stages: fetch, decode, execute, buffer and writeback. Most instructions normally spend a single cycle in each stage. The StrongARM core contains a num-ber of result bypasses. These normally allow the processor to use the results of one instruction in the following instruc-tion as soon as it has been generated. In particular, almost every instruction can read its inputs from the bypasses as it enters the execute stage if these inputs are not yet in the register file in the decode stage.

4.1 Performance Figures

We worked with Intel StrongARM SA-1110 Microproces-sor Development Board, Hardware Build Phase 5. We used ARM Developer Suite (ADS) Version 1.1 and Angel Version 1.2 Revision 2.08a as the development software.

We implemented in C the three strategies explained in Sect. 2.2. We unrolled the loops that implement the series of round transformations by one, and we kept the State array explicitly in registers. We applied the standard techniques described in [8] in order to optimise the machine codes gen-erated by the ARM compiler. We calculated the clock cycle information reading the Operating System Timer clocked by the 3.6864 MHz oscillator, that also feeds the CPU phase-locked loop (PLL). In order to remove the effects of initial cache misses, we made a large number of consecutive en-cryptions or decryptions on random input blocks, and we took the averages.

Performance figures with 128-bit key length based on the three MixColumn implementations described in Sect. 3.1 are given in Table 3. We underline the fact that in Table 3 we compare the speed of compact implementations, i.e. we exclude V2 because its code does not carry out the MixCol-umn step; in V2 the Mixcolumn operation is hardwired in the bigger look-up tables.

The experimental results are parallel to the theoretical claims: encryption with the proposed MixColumn imple-mentation is the fastest. V1T encryption is slightly slower than V1 encryption due to the initial and the final trans-positions on the State matrix. Instead, V1T is winning in decryption. We conclude that the best compact implementation is obtained combining proposed encryption code and V1T decryption code; we will refer to this version as BC.

A comparison of performances between BC and V2 is given in Table 4 for different key lengths. The results show that V2 is the fastest implementation on the processor, both for encryption and decryption. We note here that the per-formance figures of V2 compare well with the StrongARM results given in [3]. However, an exact comparison has not been possible, since the development platform and the spe-cific processor are not specified in the latter case.

Table 4: Number of clock cycles for different key lengths

Version	Key length	Encryption	Decryption
BC	128-bit	943	1312
	192-bit	1119	1559
	256-bit	1295	1806
V2	128-bit	659	638
	192-bit	767	766
	256-bit	875	874

4.2 Memory Requirements

The three different versions have different memory re-quirements. V1 and V1T employ only byte substitution ta-bles, whereas V2 employs additional encryption/decryption tables. It is clear that V2 requires more data memory com-pared with V1 and V1T. However, use of the look-up tables reduces its code size. Table 5 summarizes the memory re-quirements for the different versions, containing the code for keyschedule for the three different key lengths, the code for encryption and the code for decryption; RO stands for read only and ZI stands for zero initialised.

Table 5: Memory requirements for different versions in bytes

Version	Code	RO Data	ZI Data	Total
V1	3620	522	248	4390
V1T	4440	522	248	5210
V2	3148	2570	248	5966

V2 again has the largest memory requirements, when code and data are considered together, and V1 is the most com-pact implementation. Although V1T uses the same look-up tables as V1, it has larger memory requirements; this can be explained by the fact that it requires a more complex key schedule, which increases its code size.

4.3 The Mini-cache Solution and Effects of Cache Misses

The mini data cache, having a size of 512 bytes, is an ideal place to lock the byte substitution tables used by all candidate implementations. By manipulating memory man-agement translation tables, it is possible to assign the sub-stitution tables to the mini data cache, and all the other data area to the main data cache [5].

Table 6: Effects of the Mini-cache use on the per-formance, clock cycles, 128-bit key length

Version	Operation	With MC	Without MC
V1	keysch. + 1 enc.	3877	4204
V2	keysch. + 1 enc.	4355	4734
V1T	keysch. + 1 dec.	4719	5365
V2	keysch. + 1 dec.	5614	6215

This ensures that there will be no interference by other data structures, or other applications on the mini-cache, and once loaded, sub-stitution tables will always remain there. When there is an operating system running on the platform, resulting in fre-quent context switches, or when the number of consecutive encryptions or decryptions are small and must be succeeded (or preceded) by other applications, cache interference can present significant penalties on AES performance.

We tried to reduce this penalty by making use of the SA-1110 mini-cache. Table 6 describes the effects of the mini-cache use on the performance with the instruction cache and the main data cache initially empty. According to Table 6 results, a reduction of 8 to 12 percent in the number of clock cycles can be obtained with a single keyschedule and a single encryption or decryption by making use of the mini-cache. The improvement is more significant for decryption since it makes use of both Sbox and InvSbox (Sbox in keyschedule). Each byte substitution table, having a size of 256 bytes, oc-cupies 8 cache blocks with proper alignment. Keeping them in the mini-cache, 8 cache misses are saved for encryption and 16 cache misses are saved for decryption. However, the performance figures are still quite far from those given in Sect. 4.1. The reason is the instruction cache misses, which cannot be avoided in any way. The improvement due to the mini-cache is still convincing.

Another observation that can be made from Table 6 is that V2 encryption requires more clock cycles than V1 en-cryption, and V2 decryption requires more clock cycles than V1T decryption in case of a single encryption or decryp-tion. This is expected, since V2 is subject to additional data cache misses due to the encryption/decryption tables it uses. Moreover, V2 decryption keyschedule is more complex than others as it requires additional InvMixColumn transforma-tions applied on the expanded key. However, as the number of encryptions or decryptions are increased, we would ex-pect the speed of V2 to compensate for the cache miss or keyschedule overheads at some point.

Table 7 shows that V2 encryption is faster than BC en-cryption only if the number of consecutive encryptions is larger than 2. The decryption results, shown in Table 8, are similar: V2 decryption is faster than BC decryption when the number of consecutive decryptions is larger than 2.

We conclude that, for a particular application encrypting or decrypting messages smaller than or equal to 2 blocks (32 bytes) at a time and subject to cache block replacements between two executions, it is better strategy to use version BC instead of V2, not only from the memory requirements point of view, but also considering performance.

We may add that a reduction in the number of clock cycles probably lead also to a reduction in energy consumption, although we did not make such measurements. The use of BC is thus recommended in several use-cases, such as:

Table 7: BC encryption vs V2 encryption, clock cy-cles, 128-bit key length, substitution tables in mini-cache

Operation	BC	V2
keysched. + 1 enc.	3877	4355
keysched. + 2 enc.	4831	5021
keysched. + 3 enc.	5796	5680
keysched. + 4 enc.	6712	6328

Table 8: BC decryption vs V2 decryption, clock cy-cles, 128-bit key length, substitution tables in mini-cache

Operation	BC	V2
keysched. + 1 dec.	4719	5614
keysched. + 2 dec.	6027	6229
keysched. + 3 dec.	7390	6882
keysched. + 4 dec.	8691	7507

- The case when the running application must encrypt or decrypt and send the value of some counters or some flags, for the purpose of getting synchronized with a remote host.
- The case when data to be sent over an encrypted con-nection consist of key strokes from the keyboard or from dedicated pads (very common when using remote shells or interfaces)
- The case when small data from sensors must be pe-riodically sent to a sink, and other applications are running on the device.

5 CONCLUSIONS

In this work, the peculiarities of the common ARM ISA suited for AES are identified and appropriately exploited. A new implementation for the encryption linear mixing layer is formulated, which enhances the performance of AES on all ARM cores. The new implementation is the most compact implementation, preserves the flexibility of Rijndael and ex-hibits the highest encryption performance on ARM proces-sors, compared with similar implementations claiming low memory use. It exploits the ISA features of ARM by efficiently rearranging some operations and reducing their num-ber.

A case study on Intel StrongARM SA-1110 microproces-sor presents the performance figures for different candidate implementations and some ways of improving the perfor-mance by exploiting memory hierarchies and cache lock-ing strategies. Moreover, the case study demonstrates that the most compact implementations are also the fastest ones when there is potential cache interference, and identifies the smallest number of consecutive encryptions/decryptions that make implementations using additional look-up tables advantageous over the more compact ones on the specific architecture.

6. REFERENCES

[1] Arm Ltd. website. http://www.arm.com.

[2] Intel Ltd. website. http://www.intel.com.

[3] A survey of Rijndael implementations.

[4] http://www.tcs.hut.fi/~helger/aes/rijndael.html.

[5] ARM7. Data Sheet ARM DDI 0020C, ARM Limited, Dec 1994.

[6] Configuring ARM Caches. Application Note ARM DAI 0053B, ARM Limited, Feb 1998.

[7] Memory Management on the StrongARM SA-110. Application Note 278191-001, Intel Corporation, Sep 1998.

[8] StrongARM SA-110 Microprocessor Instruction Timing. Application Note 278194-001, Intel Corporation, Sep 1998.

[9] Writing Efficient C for ARM. Application Note ARM DAI 0034A, Jan 1998.

[10] ARM Architecture. Reference Manual ARM DDI 0100D, ARM Limited, Feb 2000.

[11] Intel StrongARM SA-1110 Microprocessor. Developer’s Manual 278240-003, Intel Corporation, Jun 2000.

[12] Announcing the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standard FIPS 197, National Institute of Standards and Technology (NIST), Nov 2001.

[13] G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient Software Implementation of AES on 32-Bit Platforms. In B. S. K. Jr., C¸ etin Kaya Ko¸c, and C. Paar, editors, Cryptographic Hardware and Embedded Systems - CHES 2002, volume 2523 of Lecture Notes in Computer Science, pages 159–171. Springer, Berlin, Aug. 2002.

[14] J. Daemen and V. Rijmen. AES Proposal:Rijndael. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/, Sep 1999.

[15] J. Daemen and V. Rijmen. Efficient Block Ciphers for Smartcards. In USENIX Workshop on Smartcard Technology (Smartcard ’99), pages 29–36, May 1999.

[16] J. Daemen and V. Rijmen. The Block Cipher Rijndael. In J.-J. Quisquater and B. Schneier, editors, Smart Card Research and Applications, volume 1820 of Lecture Notes in Computer Science, pages 288–296. Springer, Berlin, 2000.

[17] J. Daemen and V. Rijmen. Rijndael, the Advanced Encryption Standard. Dr. Dobb’s Journal, 26(3):137–139, Mar. 2001.

[18] B. Gladman. A Specification for Rijndael, the AES Algorithm. Available at http://fp.gladman.plus.com, May 2002