About the Performances of the Advanced Encryption Standard in Embedded Systems with Cache Memory

[1]Guido Bertoni, [2]Aril Bircan, [1]Luca Breveglieri, [3]Pasqualina Fragneto, [2]Marco Macchetti, [1]Vittorio Zaccaria

[1]Politecnico di Milano - P.zza L. Da vinci 32 - I-20133 Milano, ITALY

[2]ALaRI Università della Svizzera Italiana, Lugano, SWITZERLAND

[3]ST Microelectronics - Via Olivetti 2 - I-20041 Agrate B., ITALY

ABSTRACT

Modern networked embedded systems represent a growing market segment in which security is becoming an essential requirement. The Advanced Encryption Standard (AES) specification is becoming the default choice for such type of systems; however, a proper software implementation of AES is of fundamental importance in order to achieve significant performance. Current implementations presented in literature differ in terms of the amount of look-up tables used for pre-computing the functions of the encryption/decryption phase. This raises some questions regarding which AES implementation is optimal for a specific system configuration that, up to now, have been only empirically solved. In this work, we present an analytical model to study and evaluate the performance of the possible AES implementations in the early phases of system development. We then show that the proposed high-level timing model captures, with significant accuracy, the actual performance of current AES applications and thus it can be used for early evaluation of optimal AES implementations and to support the design space exploration phase. Validating experiments have been carried out on the Lx architecture, a scalable and customizable VLIW architecture developed by STMicroelectronics and HP Labs. Some final considerations are eventually reported about the relevant characteristics of the analyzed implementations and the role of the cache memory.

1. INTRODUCTION

The Advanced Encryption Standard (AES) is becoming the default choice for secure data communication in current and future embedded systems [1,2,4]. For these systems the performance of AES is of crucial importance to full-fill efficiency constraints. Available Instruction Level Parallelism (ILP), amount of pre-computation and cache memory configuration are the most important parameters that can be used by the designer to achieve performance constraints.

All the known implementations of AES perform encryption and decryption by using various amounts of Look-Up Tables (LUT's) that store pre-computed functions to obtain high time performances [3]. These implementations have been generally studied only under best-case working hypotheses, and no particular attention is dedicated to the average or worst case [2].

In some systems encryption is also interleaved with other computational tasks, in a sort of producer-consumer chain. This situation could affect the time performances of an AES software implementation, in particular in the case it relies upon the use of a cache memory. This creates some performance issues related to the memory sub-system configuration that, up to now, have been only empirically solved.

In this paper we address specifically these problems by introducing an analytical methodology to evaluate the goodness of the possible AES implementations for a specific target platform configuration. We then apply this methodology to individuate a set of optimal implementations of the AES algorithm for the Lx architecture, a scalable and customizable VLIW architecture developed by STMicroelectronics and HP Labs. The paper is organised as follows. In section 2 a brief overview of the Rijndael AES algorithm is given. In section 3 background work on software AES implementations is proposed. In section 4 we introduce a methodology to evaluate analytically the time performances of AES, while in section 5 we derive some experimental data to validate the methodology by studying the AES implementation for the Lx architecture.

2. The Rijandael AES algorithm

This section provides a brief summary of the AES algorithm [1,2,4] (also called *Rijndael*, by the names of its authors), useful for understanding the study proposed in this paper.

The algorithm can be decomposed in three basic blocks, i.e., *encryption*, *decryption* and *key schedule*. The Rijndael cipher algorithm works by means of basic execution steps that are called *rounds*; a round is a fixed sequence of transformations that are applied to a so-called *state* array, containing the data block to encrypt. The number of rounds is selected depending on the key length; 10, 12 or 14 rounds must be performed, in the case of 128, 192 or 256-bits keys, respectively. The transformations applied in sequence during each round are four. According to the standard, they correspond to the *Byte Substitution*, the *Shift Rows*, the *Mixcolumns* and the *Add Round Key*. The decryption is computed by applying the inverse of all the transformations described above, to return to the plaintext; of course all the transformations must be applied to the state array in the reversed order. For both encryption and decryption, the secret key must be expanded in rounds key. This task is performed trough the key schedule algorithm.

3. Background on AES implementations

The encryption and decryption processes can be executed by analytically computing each transformation or using some pre-computed values stored in a look-up table (LUT).

The versions without pre-computed values are generally not considered in the literature, due to the high cost of the inversion in GF(2^8) required by the *Substitution Byte* transformation. This implementation was characterized by execution time about one

order of magnitude higher then the other implementations. Due to its inefficiency, this version will not be considered in the rest of the paper.

Starting from the AES code developed by Gladman [2] we can identify six distinct versions, each differing in terms of amount of look-up tables. The first considered version is named **Version 1** and requires 3 different LUT's: two of them contain the values of the nonlinear transformation and its inverse, respectively, while the third one contains a small table storing *rcon*, a vector of constants necessary for the key schedule algorithm. The *rcon* is composed by 10 elements of 32 bits each, and its overall size is of 40 bytes for 128 bit secret key. The *rcon* table is used in all the considered AES versions. Therefore the total amount of look-up tables required by Version 1 is of 552 byte.

Version 2 introduces some improvements on version 1 by avoiding the calculation of the *Mixcolumns*. In this version one main table is used, called *enc_table,* storing the result of the application in sequence of the *Substitution Byte* and *Mixcolumns* transforms to a byte. The table is composed by 256 elements and each element is a word of 32 bits. In this version, each byte of the state columns is used to address *enc_table*. The four 32 bits words read from the table are rotated and then XORed. The first word does not require any rotation, the second one must be rotated by 1 byte position, the third by 2 bytes and the fourth by 3 bytes. The amount of precomputation data gives a total of 2600 bytes. **Version 3** is an enhancement of version 2, and introduces two new tables, one for encryption and one for decryption, both used in the last round to replace the *Substitution Byte*. These tables contain identical data, arranged in 32 bit words, which are easily accessed in 32 bit platforms, e.g. the Intel IA32 platform. The total amount of precomputation data is of 4136 bytes.

Version 4 eliminates the rotation of the elements read from the table *enc_table*. This is done by using four different tables *enc_table0*, *enc_table1*, *enc_table2* and *enc_table3*. These tables are the rotated versions of *enc_table*. For the last round *Substitution Byte* is computed with a S-box table. **Version 5** is similar to version 4, but a table for the last round is used as in version 3. **Version 6** is obtained from version 4 by introducing 4 new tables for the computations needed in the final round. The 4 new tables contain the S-box values, arranged in 32 bits words, ready to be XORed, thus avoiding the need of shift operations.

Another version of AES (called **AEST**) requires only S-box tables but uses a transposed state matrix. This method allows to speed-up encryption and decryption, as shown in [6]. The amount of LUTs is the same of version 1 but the performance is better. This means that AES1 is dominated by AEST, so we will not consider the former in the rest of the paper.

A similar approach can be used for version 3 and 5; in fact those version are particular suitable for systems characterized by a byte access time higher than the word access time. However, this is not the case of our target architecture (Lx). In fact, our target CPU has the same latency for byte and word access, so we will not consider version 3 and 5 in the rest of the paper since their performance is equivalent to version 2 and 4 performance respectively.

Table 1 reports for the two types of tables (Sbox and Enc_table) the absolute size in bytes and the number of blocks of the cache used to store it (called k, in the rest of the paper). Table 2 summarizes the specific configuration of tables necessary by each version of AES considered in this paper. Each column reports the number of tables per type (Sbox and Enc_table) and the number of accesses executed in one encryption call. For instance SBOX table is used by version AEST, AES2 and AES4. While AES2 uses one Enc_table performing 144 reads, AES4 uses 4 enc_table*s* and for each it performs 36 acesses. AES6 uses two sets of Enc_table*s*, for a total of 8 tables. Each table of the first set is read 36 times, while each table of the second set is read 4 times.

Table characteristic		**Block size**	
	Size	16 bytes (128 bits)	32 bytes (256 bits)
Sbox	256 bytes	16	8
Enc_table	1024 bytes	64	32

Table 1 Data cache blocks required by tables, versus different block sizes.

	AEST		**AES2**		**AES4**		**AES6**	
SBOX	1	160	1	16	1	16		
Enc_table's			1	144	4	36	4+4	36/4

Table 2 Relations between implementations, number of tables and number of accesses.

For what concerns the performance estimation of AES, Gladman's study [2] is the only reference in the literature in which the execution time of AES is analyzed. However, Gladman measures the performance when the execution is free from cache misses. The only constraint is that the cache should be large enough to store both the input data and the LUT's used by the generic function. As we will show, in actual systems *compulsory* cache misses cannot be neglected and could impact strongly the performance of the AES routines.

4. Analytical evaluation of AES performance

In the rest of the paper we focus on encryption performance estimation because decryption has the same behavior. Moreover, in many systems the key schedule is called few times and its performance is very predictable so it can be accounted for with a constant value.

The execution time of the considered AES implementation depends, of course, on the structure of the system and the operations performed by the algorithm. Here we assume that calls to the AES encryption algorithm are performed in *bursts,* i.e., continuous regions of time characterized exclusively by calls to the AES encryption algorithm. Bursts are separated by calls to other application functions that are assumed to trash the data and instruction cache in a random way.

In the rest of the section we will show that the performance of a specific AES implementation depends on the number of AES calls performed within the burst and that a general optimum implementation does not exists. When characterizing the execution time of a burst we assume that the instruction cache can contain entirely the code of the algorithm such that the cycles lost due to misses are associated only to the first call to the AES algorithm. Moreover we assume to work with a 4-way set-associative data cache. This is motivated by the fact that AES

uses 4 groups of data: the input data block to encrypt, the various LUT's, the various round keys, and the output encrypted (or decrypted) data block. With a 4-way set associative architecture the data cache management system can avoid mutual interference, yielding improved performance. In this scenario, we can focus on *compulsory* cache misses, i.e., misses related to the initial filling of the data cache.

Under these assumptions, the execution time T_{BURST} of a burst of i calls of a version v of the AES algorithm can be decomposed into an ideal execution time associated to the instructions of each call ($E(v)$) plus the miss penalty associated to the accesses to the l-th LUT, as follows:

$$T_{BURST}(i,v) \cong i \cdot E(v) + \sum_{l \in LUTs(v)} \sum_{t=1}^{i*a(l,v)} m_{DC} \cdot p(t,l) + c \quad (1)$$

where m_{DC} is the miss penalty of a miss in the data cache, $a(l,v)$ is the number of accesses to LUT l performed by AES algorithm v (see Table 1 and 2), and $p(t,l)$ is the probability to have a miss during the t-th access to the l-th LUT within a burst. Parameter c represents the number of execution cycles of the key-schedule. Referring to Tables 1 and 2, for example AES version 4 has LUT={ $s1$, $l1$, $l2$, $l3$, $l4$ }, where $s1$ is of type SBOX and $l(1\text{-}4)$ is of type Enc_table. More in particular we have that $a(s1,4) = 16$ and $a(l(1\text{-}4),4)=36$.

4.1 Characterization of the model's coefficients

Equation 1 can be used to derive the behavior of a specific version of algorithm v for each value of i. However, some of its parameters must be characterized for the particular architecture under consideration. From Equation 1, it turns out that parameter c can be estimated by running and measuring the execution cycles of any algorithm v when i=0 ($c= T_{BURST}(0,.)$). On the other hand, parameter m_{DC} is an architectural parameter that is independent of the algorithm and can be derived from the Instruction Set Architecture description very easily.

For what concerns $p(t)$ we will now show that its asymptotical value is zero. In fact we note that the accesses to the memory are executed in a substantially random way, therefore the distribution of the accesses to the data cache blocks is assumed uniformly random. Note however, that this does not imply $p(t,l)=1/k(l)$ (where $k(l)$ is the number of cache blocks of table l) since the probability of miss during access t depends on the particular state of the cache. More in detail we assume that the probability to have a miss during access t depends on the number $\hat{x}(t,l)$ of blocks that have been already loaded during the previous accesses, and on the number $k(l)$ of cache blocks of table l:

$$p(t,l) = \frac{k(l) - \hat{x}(t,l)}{k(l)} \quad (2)$$

The problem now consists in determining the average number of blocks $\hat{x}(t,l)$ that have been already loaded into the cache. This can be done by noting that $\hat{x}(0,l) = 0$ and that:

$$\hat{x}(t+1,l) = \hat{x}(t,l)(1 - p(t,l)) + (\hat{x}(t,l)+1)p(t,l) \quad (3)$$

i.e., the average number of blocks in the cache during the access $t+1$ is equal to $x(t,l)$ with probability ($1\text{-}p(t,l)$) (i.e., the probability of a hit at time t) and $x(t,l)+1$ with probability $p(t,l)$ (i.e., the probability of a miss).

By substituting Equation 2 in Equation 3, we obtain that:

$$\hat{x}(t+1,l) = \frac{k(l)-1}{k(l)} \hat{x}(t,l) + 1$$

(4)

This recurrence equation determines completely the average number of cache blocks that are already stored in the cache at time t. Figure 1 shows an example of the resulting behavior of $p(t)$, for k=32. As can be seen, p tends to 0 as t goes to infinity.

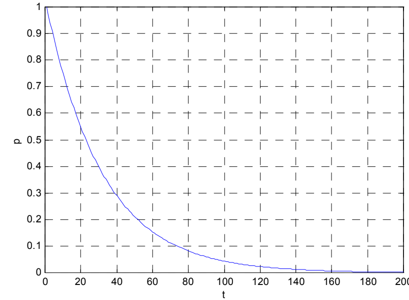

Figure 1 Behavior of $p(t)$ when k=32.

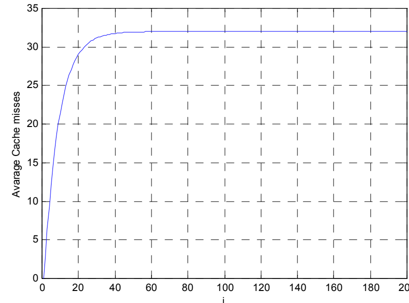

Figure 2. Number of average cache misses for varying i [a=4, k=32]

The behavior of $p(t)$ is such that the average number of cache misses

$$f(i,l,v) = \sum_{t=1}^{i*a(l,v)} p(t,l)$$

tends effectively to $k(l)$ as i grows. An example of such behavior is shown in Figure 2 (for a=4 and k=32).

This result allows us to derive also a method to estimate $E(v)$. In fact, given a small variation Δi, we have that:

$$E(v) = \lim_{i \to \infty} \frac{\Delta T_{BURST}(i,v)}{\Delta i} \quad (5)$$

since the contributions to Equation 1 given by c and the sum of $p(t)$s becomes a constant for big is. In other words, Equation 5 suggests to measure the difference in terms of execution cycles for big values of i (in our experimental results we used 512 and 1024) in order to determine $E(v)$.

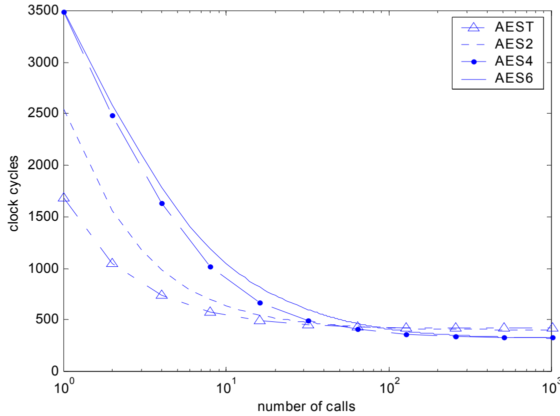

Figure 3. Behavior of $T_{BURST}(i)/i$ for the presented AES implementations (coefficients derived by the Lx architecture)

Figure 3 shows, for the AES versions considered in this paper, the predicted (theoretical) average execution time for each call (namely $T_{BURST}(i)/i$). The coefficients have been characterized from the Lx architecture, (also used in the Experimental Results). As can be seen, for a low number of calls, the AES version with less LUT accesses is theoretically preferable. In this case, in fact, the higher number of computation steps $E(v)$ is counterbalanced by a lower execution time spent in managing misses.
On the other hand, when the number of calls grows, the version with more LUT accesses becomes preferable, since misses are well amortized over the total number of calls. This result is of fundamental importance when dealing with the choice of an optimal AES implementation. The model shown in Equation 1 can thus be used to assess the best implementation suitable for a particular architecture without performing all the simulations.
In the next section, we will show that the theoretical results derived in this section are confirmed by experimental data.

5. Experimental results

We derived experimental results by using a VLIW CPU called Lx (ST200, courtesy by STMicroelectronics) [5]. First of all, we have characterized the coefficients of the model for each algorithm. The following values have been obtained:

- E(AEST)=416, E(AES2)=400, E(AES4)=319 and E(AES6)=313
- c=1000

Exhaustive simulations for all the AES versions have been performed, by using a data cache line of 16 bytes and by varying the number of calls from 1 to 1024. Figure 4 shows a scatter plot of the average estimated and measured time for call ($T_{BURST}(i)/i$)), As can be seen, Equation 1 is able to predict with a significant accuracy the real execution time of the algorithm. Experimental results have confirmed that AEST is the most performant version for up to 45 calls per burst; between 45 and 680 calls per burst AES6 is the best choice, while for more than 680 calls per burst AES6 is better.

Once the model has been characterized and validated, we used it to determine the optimal configurations for other values of cache line sizes, without performing any simulation. For a data cache line of 32 bytes, AEST has been shown superior to other algorithms up to 28 calls, AES4 for up 420 calls and AES6 for more than 420 calls per burst. As expected, when cache blocks are increased in size, cache misses finish earlier and thus the trade-off point between AES4 and AEST is decreased (from 45 to 28 calls).

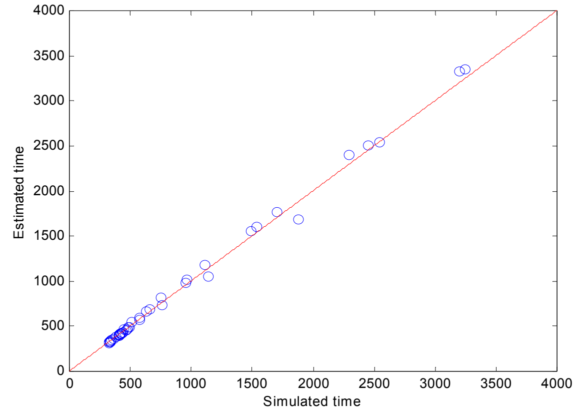

Figure 4. Scatter Plot between estimated time (Equation 1) and Measured (simulated) time.

6. Conclusions

In this paper, a model supporting the fast design space exploration of the cache line parameters impacting the performance of AES implementations has been presented. The model has been validated on a VLIW platform and the prediction error is near to zero. To the knowledge of the authors no other study of the impact of caches on AES has been reported before. The methodology presented can be used to estimate which implementation of the AES is the best choice on the basis of the amount of data to be encrypted by using a negligible number of simulations (to characterize the model). We are currently working on adapting the model to small cache sizes and decreasing the set associativity of the cache.

7. References

1. *"Announcing the ADVANCED ENCRYPTION STANDARD (AES)"* – Federal Information Processing Standards Publication, http://csrc.nist.gov/encryption/aes/, 2001
2. B. Gladman, *"A Specification for Rijndael, the AES Algorithm"*, http://fp.gladman.plus.com/, 2001
3. B. Gladman, http://fp.gladman.plus.com/cryptography_technology/rijndael/
4. J. Daemen, V. Rijmen, "The Block Cipher Rijndael", Smart-Card Research and Applications, LNCS 1820, J.-J. Quisquater and B. Schneier, Eds., Springer-Verlag, 2000, pp. 288-296
5. P. Faraboschi, G. Brown, J. Fisher, G. Desoli, and F. Homewood, ``Lx: a technology platform for customizable VLIW embedded processing," Proceedings of the International Symposium on Computer Architecture, June 2000, pp. 203--213.
6. G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti and S. Marchesin "Efficient Software Implementation of AES on 32-bits Platforms", Proceedings of CHES 2002 pp159-171