A Comparative Study of Performance of AES Final Candidates Using FPGAs*

Andreas Dandalis[1], Viktor K. Prasanna[1], and Jose D.P. Rolim[2]

[1] University of Southern California, Los Angeles CA 90089, USA
{dandalis,prasanna}@halcyon.usc.edu
http://maarcII.usc.edu
[2] Centre Universitaire d'Informatique, Université de Genève
24 Rue General Dufour, 1211 Genève 4, Switzerland
Jose.Rolim@cui.unige.ch

Abstract. In this paper we study and compare the performance of FPGA-based implementations of the five final AES candidates (MARS, RC6, Rijndael, Serpent, and Twofish). Our goal is to evaluate the suitability of the aforementioned algorithms for FPGA-based implementations. Among the various time-space implementation tradeoffs, we focused primarily on time performance. The time performance metrics are throughput and key-setup latency. Throughput corresponds to the amount of data processed per time unit while the key-setup latency time is the minimum time required to commence encryption after providing the input key. Time performance and area requirement results are provided for all the final AES candidates. To the best of our knowledge, we are not aware of any published results that include key-setup latency results. Our results suggest that *Rijndael* and *Serpent* favor FPGA implementations *the most* since their algorithmic characteristics match extremely well with the hardware characteristics of FPGAs.

1 Introduction

The projected key role of AES in the 21st century cryptography led us to implement the AES final candidates using Field Programmable Gate Arrays (FPGAs). The goal of this study is to evaluate the performance of the AES final candidates on FPGAs and to make performance comparisons. In addition, we evaluate the suitability of reconfigurable hardware as an alternative solution for AES implementations.

In this study, we concentrate only on performance issues. We assume that all the considered algorithms are secure. Time performance and area requirements results are provided for all the final candidates. The time performance metrics are throughput and key-setup latency. Throughput corresponds to the amount of data processed per time unit while key-setup latency is the minimum time

* This research was performed as part of the MAARCII project. This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-99-1-0004 monitored by Fort Huachuca.

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 125–140, 2000.

required to commence encryption after providing the input key. Besides the throughput metric, the latency metric is the key measure for applications where a small amount of data is processed per key and key context switching occurs repeatedly.

FPGA technology is a growing area that has the potential to provide the performance benefits of ASICs and the flexibility of processors. This technology allows application-specific hardware circuits to be created on demand to meet the computing and interconnect requirements of an application. Moreover, these hardware circuits can be dynamically modified partially or completely in time and in space based on the requirements of the operations under execution [5,13].

Private-key cryptographic algorithms seem to fit extremely well with the characteristics of the FPGAs. The fine-granularity of FPGAs matches extremely well the operations required by private-key cryptographic algorithms such as bit-permutations, bit-substitutions, look-up table reads, and boolean functions. On the other hand, the constant bit-width required alleviates accuracy-related implementation problems and facilitates efficient designs. Moreover, the inherent parallelism of the algorithms can be efficiently exploited in FPGAs. Multiple operations can be executed concurrently resulting in higher throughput compared with software-based implementations. Finally, the key-setup circuit can run concurrently with the cryptographic core circuit resulting in low key-setup latency time and agile key-context switching.

In our implementations, we focused on the time performance. Our goal was to exploit, for each candidate, the inherent parallelism of the cryptographic core (at the round level) to optimize performance. Moreover, we have exploited the low-level hardware features of FPGAs to enhance the performance of individual required operations. Our throughput results are compared with the FPGA-based results in [9,11]. In [9,11], only the cryptographic core of each algorithm was implemented using FPGAs and, thus, no key-setup latency results were provided. As a result, only throughput comparisons are made with the FPGA-based results in [9,11]. Moreover, our time performance results are compared with the best software-based results in [3,4] and the NSA's ASIC-based results [17].

An overview of FPGAs and FPGA-based cryptography is given in Section 2. In Section 3, general aspects of our implementations are discussed. The implementation results for each algorithm are described in Section 4. In Section 5, a comparative analysis among the results of all the candidates is performed. In addition, comparisons with related work are made. Comparisons with software and ASIC implementations are made in Sections 6 and 7 respectively. Finally, in Section 8, concluding remarks are made.

2 FPGA Overview

Processors and ASICs are the cores of the two major computing paradigms of our days. Processors are general purpose and can virtually execute any operation. However, their performance is limited by the restricted interconnect, datapath, and instruction set provided by the architecture. Conversely, ASICs

are application-specific and can achieve superior performance compared with processors. However, the functionality of an ASIC design is restricted by the designed parameters provided during fabrication. Any update to an ASIC-based platform incurs high cost. As a result, ASIC-based approaches lack flexibility.

FPGA technology is a growing area of research that has the potential to provide the performance benefits of ASICs and the flexibility of processors. Application specific hardware circuits can be created on demand to meet the computing and interconnect requirements of an application. Moreover, these hardware circuits can be dynamically modified partially or completely in time and in space based on the requirements of the operations under execution. As a result, superior performance can be expected compared with the performance of the equivalent software implementation executed on a processor.

FPGAs were initially an offshoot of the quest for ASIC prototyping with lower design cycle time. The evolution of the configurable system technology led to the development of configurable devices and architectures with great computational power. As a result, new application domains become suitable for FPGAs beyond the initial applications of rapid prototyping and circuit emulation. FPGA-based solutions have shown significant speedups (compared with software and DSP based approaches) for several application domains such as signal & image processing, graph algorithms, genetic algorithms, and cryptography among others.

The basic feature underlying FPGAs is the programmable logic element which is realized by either using anti-fuse technology or SRAM-controlled transistors. FPGAs [5,13] have a matrix of logic cells overlaid with a network of wires. Both the computation performed by the cells and the connections between the wires can be configured. Current devices mainly use SRAM to control the configurations of the cells and the wires. Loading a stream of bits onto the SRAM on the device can modify its configuration. Furthermore, current FPGAs can be reconfigured very quickly, allowing their functionality to be altered at runtime according to the requirements of the computation.

2.1 FPGA-Based Cryptography

FPGA devices are a highly promising alternative for implementing private-key cryptographic algorithms. Compared with software-based implementations, FPGA implementations can achieve superior performance. The fine-granularity of FPGAs matches extremely well the operations required by private-key cryptographic algorithms (e.g., bit-permutations, bit-substitutions, look-up table reads, boolean functions). As a result, such operations can be executed more efficiently in FPGAs than in a general-purpose computer.

Furthermore, the inherent parallelism of the algorithms can be efficiently exploited in FPGAs as opposed to the serial fashion of computing in an uniprocessor environment. At the cryptographic-round level, multiple operations can be executed concurrently. On the other hand, at the block-cipher level, certain operation modes allow concurrent processing of multiple blocks of data. For example, in the ECB mode of operation, multiple blocks of data can be

processed concurrently since each data block is encrypted independently. Consequently, if p rounds are implemented, a throughput speed-up of $O(p)$ can be achieved compared with a "single-round" based implementation (one round is implemented and is reused repeatedly). Moreover, by adopting deep-pipelined designs, the throughput can be increased proportionally with the clock speed. On the contrary, in feedback modes of operation (e.g., CBC, CFB), where the encryption results of each block are fed back into the encryption of the current block [14], encryption can not be parallelized among consecutive blocks of data. As a result, the maximum throughput that can be achieved depends mainly on the encryption time required by a single cryptographic round and the efficiency of the implementation of the key-setup component of an algorithm.

Besides throughput, FPGA implementations can also achieve agile key-context switching. Key-context switching includes the generation of the required key-dependent data for each cryptographic round (e.g., subkeys, key-dependent S-boxes). A cryptographic round can commence as soon as its key-dependent data is available. In software implementations, the cryptographic process can not commence before the key-setup process for all the rounds is completed. As a result, excessive latency is introduced making key-context switching inefficient. On the contrary, in FPGAs, each cryptographic round can commence as early as possible since the key-setup process can run concurrently with the cryptographic process. As a result, minimal latency can be achieved.

Security issues also make FPGA implementations more advantageous than software-based solutions. An encryption algorithm running on a generalized computer has no physical protection [14]. Hardware cryptographic devices can be securely encapsulated to prevent any modification of the implemented algorithm. In general, hardware-based solutions are the embodiment of choice for military and serious commercial applications (e.g., NSA authorizes encryption only in hardware) [14].

Finally, even if ASICs can achieve superior performance compared with FPGAs, their flexibility is restricted. Thus, the replacement of such application-specific chips becomes very costly [10] while FPGA-based implementations can be adapted to new algorithms and standards. However, if ultimate performance is essential, ASICs solutions are superior.

3 Implementation and Design Decisions

As a hardware target for the proposed implementations, we have chosen the Xilinx Virtex family of FPGAs. Virtex is a high-capacity, high-speed performance FPGA providing a superior system integration feature set [16]. For mapping onto Virtex devices, we used the Foundation Series v2.1i software development tool. The synthesis and place-and-route parameters of the tool remained the same for all the implementations. All the results were based on placed-and-routed implementations (device speed -6) that included both the key-setup component and the cryptographic core along with their control circuit.

Among the various time-space tradeoffs, our focus was primarily time performance. For each algorithm we have implemented the key-setup component, the control circuitry, and the encryption block cipher for 128-bit data blocks using 128-bit keys. A "single-round" based design was chosen for each implementation. Since one round was implemented, it was reused repeatedly. The key-setup component was processing data in parallel with the cryptographic core. While the cryptographic core was processing the data of the ith round, the key-setup component was calculating the key-dependent data for the $(i + 1)th$ round. As a result, even if an algorithm does not support on-the-fly key generation in the software domain, the key setup can be executed on the fly in FPGAs.

Our goal was to maximize throughput for each candidate algorithm. We have exploited the inherent parallelism of each cryptographic core and the low-level hardware features of FPGAs to enhance the performance. The performance metrics are throughput and key-setup latency. The throughput metric indicates the amount of data encrypted per time unit after the initialization of the algorithm. The key-setup latency denotes the minimum time required to commence encryption after providing the input key. While throughput indicates the bulk-encryption capability of the implementation, key-setup latency indicates the capability of agile key-context switching.

Since one round was implemented and was reused repeatedly, the throughput results correspond to $\frac{128}{n * t_{round}}$, where n and t_{round} are the the number of required rounds and the encryption time per round respectively. Similar performance analysis can be performed for larger sizes of data blocks and keys as well as for implementations that process multiple blocks of data concurrently.

The key-setup latency issue was of primary interest, that is, the cryptographic core had to commence as early as possible. Based on the achieved throughput, we designed the key-setup component to sustain the processing rate of the cryptographic core and to achieve minimal latency. The key-setup latency metric is the key metric for applications where a small amount of data is processed per key and key-context switching occurs repeatedly. In software implementations, the cryptographic process cannot commence before the key-setup process for all the rounds is completed. As a result, the key-setup latency time equals the key-setup time.

To implement efficient key-setup circuits, we took advantage of the embedded memory modules (Block SelectRAM) of the Virtex FPGAs [16]. The Virtex FPGA Series provides dedicated on-chip blocks of true dual-read/write port synchronous RAM, with 4096 memory cells each. Depending on the size of the device, 32-132 Kbits of data can be stored using the Block SelectRAM memory modules. The key-setup circuit utilized these memory modules to pass its results to the cryptographic core. As a result, the cryptographic core could commence as soon as the key-dependent data for the first encryption round is available in the memory modules. Then, during each encryption round, the cryptographic core reads the corresponding data from the memory modules.

For each algorithm, we have also implemented the key-setup circuit and the cryptographic core separately. For all the implementations, the maximum clock

speed of the key-setup circuit was higher than the maximum clock speed of the cryptographic core. Based on the results of these individual implementations, we also provide latency estimates for implementations that clock each circuit at its maximum speed.

Regarding the cryptographic cores, the majority of the required operations fit extremely well in Virtex FPGAs. The permutations and substitutions can be hard-wired while distributed memory can be used as look-up tables. In addition, boolean functions, data-dependent rotations, and addition can be mapped very efficiently onto Virtex FPGAs. Wherever a multiplication with a constant was required, constant coefficient multipliers were utilized to enhance the performance compared with "regular" multipliers. Regular multiplication is required only by the $MARS$ and $RC6$ block ciphers. In both cases, two 32-bit numbers are multiplied and the lower 32-bit of the output are used in the encryption process. We tried the multiplier-macros provided for Virtex FPGAs but we found that they were a performance bottleneck. Besides the excessive latency that was introduced due to the numerous pipeline stages, excessive area was also required since the full multiplier was mapped onto the FPGA. Instead of using these macros, a multiplier that computes partial results in parallel and outputs only the required 32-bits was used. As a result, the latency was reduced by more than 50% and the area requirements were also reduced significantly.

4 Implementation Results

In the following, implementation results as well as relevant performance issues specific to each algorithm are provided. The key-setup latency results are represented both as absolute time and as the fraction of the corresponding encryption time over one 128-bit block of data. In addition, the throughput results are represented both as encryption rate and as encryption rate elaborated on area. Finally, area requirements results are provided for both the key-setup and the cryptographic core circuits. In the following, the order of presenting the algorithms is alphabetic. Detailed algorithmic information for each candidate can be found in [6,12,7,2,15].

4.1 MARS

The $MARS$ block cipher is the IBM submission to AES [6]. The time performance and area requirements results for our $MARS$ implementation are shown in Table 1.

Key Setup. The $MARS$ key expansion procedure expands the input 128-bit key into a 1280-bit key. First a linear-key expansion occurs following by stirring the key-words based on an S-box. Both processes involves simple operations performed repeatedly. However, the final stage of modifying the multiplication key-words involves string-matching operations that are relatively expensive functions. String-matching is an expensive operation compared with the rest of the

Table 1. Implementation Results for MARS

Key-Setup Latency		Throughput		Area Requirements		
μs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	# of slices Key-Setup	Cryptographic Core
1.96	3.12	101.88	29.55	6896	2275 (33%)	4621 (67%)

operations required by *MARS*. A compact implementation of string-matching introduces high latency while a high-performance implementation increases the area requirements dramatically. In our implementation, the last stage of the key-expansion process (i.e., string-matching) was not implemented. In spite of this, the introduced key-setup latency was still relatively high (the worst among all the implementations considered in this paper).

Cryptographic Core. The cryptographic core of *MARS* consists of a 16-round cryptographic layer wrapped with two layers of 8-round "forward" and "backward mixing" [6]. The achieved throughput depended mainly on the efficiency of the multiplier (please see Section 3). In our implementation only one round of each layer was implemented that was used repeatedly. The encryption time for one block of data was 32 clock cycles. An interesting feature of our design is that by increasing the utilization factor of the processing stages (i.e. all the three processing stages execute in parallel), the average encryption time for one block of data can be reduced to 16 clock cycles for operation modes that allow concurrent processing of multiple blocks of data (e.g., non-feedback, interleaved).

4.2 RC6

The *RC*6 block cipher is the AES proposal of the RSA Laboratories and R. L. Rivest from the MIT Laboratory for Computer Science [12]. The implemented block cipher corresponds to $w = 32$-bit round keys, $r = 20$ rounds, and $b = 14$-byte input key. The time performance and area requirements results for our *RC*6 implementation are shown in Table 2.

Table 2. Implementation Results for RC6

Key-Setup Latency		Throughput		Area Requirements		
μs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	# of slices Key-Setup	Cryptographic Core
0.17	0.15	112.87	42.59	2650	901 (34%)	1749 (66%)

Key Setup. The *RC*6 key setup expands the input 128-bit key into 42 round keys. The key for each round corresponds to a 32-bit word. The key scheduling is fairly simple. The round-keys are initialized based on two constants. We have implemented the initialization procedure using a look-up table since it is the same for any input key. Then, the contents of the look-up table were used to generate the round-keys with respect to the input key. As a result, remarkably low key-setup latency was achieved that was equal to the 15% of the time for encrypting a block of data.

Cryptographic Core. The cryptographic core of *RC*6 consists of 20 rounds. The symmetry and regularity found in the *RC*6 block cipher resulted in a compact implementation. The entire data-block was processed at the same time by using two identical circuits. The achieved throughput depended mainly on the efficiency of the multiplier (please see Section 3).

4.3 Rijndael

The *Rijndael* block cipher is the AES proposal of J. Daemen and V. Rijmen from the Katholieke Universiteit Leuven [7]. The implemented block cipher corresponds to $N_b = 4$, $N_k = 4$, and $N_r = 10$ (i.e., 4×32-bit block data, 4×32-bit key, 10 rounds). The time performance and the area requirements results of our implementation are shown in Table 3.

Table 3. Implementation Results for Rijndael

Key-Setup Latency		Throughput		Area Requirements		
µs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	# of slices Key-Setup	Cryptographic Core
0.07	0.20	353.00	62.22	5673	1361 (24%)	4312 (76%)

Key Setup. The *Rijndael* key setup expands the input 128-bit key into a 1408-bit key. Simple operations are used that resulted in extremely low key-setup latency latency. ROM-based look-up tables were utilized to perform the *SubByte* transformation. The achieved latency was the lowest among all the implementations considered in this paper.

Cryptographic Core. The cryptographic core of *Rijndael* consists of 10 rounds. The cryptographic core is ideal for implementations on FPGAs. It combines fine-grain parallelism with look-up table operations. The round transformation can be represented as a look-up table resulting in extremely high speed. We have implemented a ROM-based fully-parallel version of the look-up table. By combining common references to the look-up table, we have achieved a 25% savings

in ROM compared with the straightforward implementation suggested in the AES proposal [7]. The simplicity of the operations and the inherent fine-grain parallelism resulted in the highest throughput among all the implementations. Furthermore, the *Rijndael* implementation had the highest area utilization factor (i.e., throughput per area unit).

4.4 Serpent

The *Serpent* block cipher is the AES proposal of R. Anderson, E. Biham, and L. Knudsen from Technion, Cambridge University, and University of Bergen respectively [2]. The time performance and area requirements results for our *Serpent* implementation are shown in Table 4.

Table 4. Implementation Results for Serpent

Key-Setup Latency		Throughput		Area Requirements		
µs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	# of slices Key-Setup	Cryptographic Core
0.08	0.09	148.95	58.41	2550	1300 (51%)	1250 (49%)

Key Setup. The *Serpent* key setup expands the input 128-bit key into a 4224-bit key. First, the input key is padded to 256 bits and then it is expanded to an intermediate key by iterative mixing of the key data. Finally, by using look-up tables, the keys for all the rounds are calculated. The simplicity of the required operations resulted in extremely low key-setup latency (the second lowest among all the implementations considered in this paper).

Cryptographic Core. The cryptographic core of *Serpent* consists of 32 rounds. The round transformation is a linear transform consisting of rotations, shifts, and XOR operations. Neither multiplication nor addition is required. As a result, the lowest encryption time per round and the most compact implementation were achieved among all the implementations. Furthermore, the *Serpent* implementation had the second higher area utilization factor (i.e. throughput per area unit).

4.5 Twofish

The $Twofish$ block cipher is the AES proposal of the Counterpane Systems, Hi/fn, Inc., and D. Wagner from the University of California Berkeley [15]. The time performance and area requirements results of our implementation are shown in Table 5.

Table 5. Implementation Results for Twofish

Key-Setup Latency		Throughput		Area Requirements		
					# of slices	
µs	key-setup latency time / block encryption time	MBits / sec	KBits / (sec*slice)	Total	Key-Setup	Cryptographic Core
0.18	0.25	173.06	18.48	9363	6554 (70%)	2809 (30%)

Key Setup. The $Twofish$ key setup expands the input 128-bit key into a 1280-bit key. Moreover, it generates the key-dependent S-boxes used in the cryptographic core. Four 128-bit S-boxes are generated. Since our goal was to minimize latency, we have implemented a parallel version of the key setup consisting of 24 q_0/q_1 permutation boxes and 2 MDS matrices [15]. Moreover, the RS matrix was implemented for the S-box generation. The matrices are used for "constant matrix"-to-matrix multiplication over $GF(2^8)$. The best known implementation of a constant coefficient multiplier in Virtex FPGAs is by using a look-up table [16]. As a result, low latency was achieved but excessive area was required. The area requirements corresponded to the 70% of the total area. However, by implementing a more compact design (e.g., reusing processing elements), the key-setup latency would increase.

Cryptographic Core. The cryptographic core of $Twofish$ consists of 16 rounds. The structure of the round transformation is similar to the structure of the key-expansion circuit. The only major difference is the S-boxes that the cryptographic core uses.

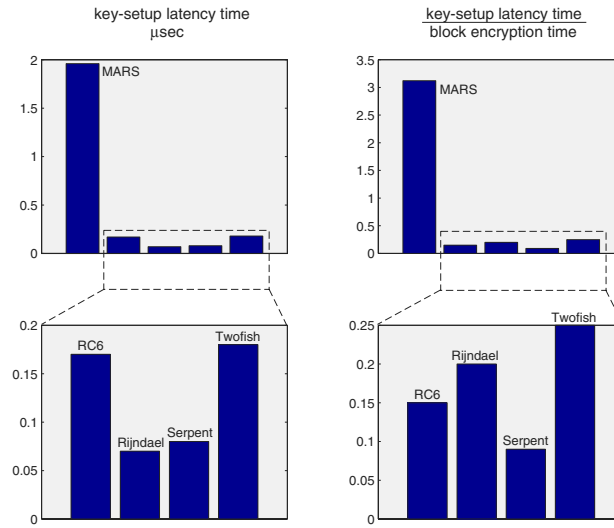
4.6 Key-Setup Latency Improvements

For each algorithm, we have also implemented the key-setup circuit and the cryptographic core separately. For each algorithm, the maximum clock speed of the key-setup circuit was higher than the maximum clock speed of the cryptographic core. Thus, by clocking each circuit at its maximum clock speed, improvement in key-setup latency can be achieved. No additional synchronization hardware is required since we can configure the read/write ports of the Block SelectRAMs having different clock speeds. Compared with implementations using one clock, the key-setup latency time can be reduced by a factor of 1.35, 2.96, 1.43, 1.00, and 1.15 for $MARS$, $RC6$, $Rijndael$, $Serpent$, and $Twofish$ respectively. Clearly, the $RC6$ block cipher can achieve the best key-setup latency improvement by clocking the key-setup and the cryptographic core circuits at their maximum clock speeds. For the $MARS$ block cipher, the result is based on an implementation that does not include the circuit for modifying the multiplication key-words.

5 Comparative Analysis of Our FPGA Implementations

In Table 6, key-setup latency comparisons are made among our FPGA implementations. The comparisons are made in terms of absolute time and the ratio of the key-setup latency time to the time required to encrypt one block of data. The latter metric represents the capability of agile key-context switching with respect to the encryption rate.

Table 6. Key-Setup Latency Comparisons of Our FPGA Implementations

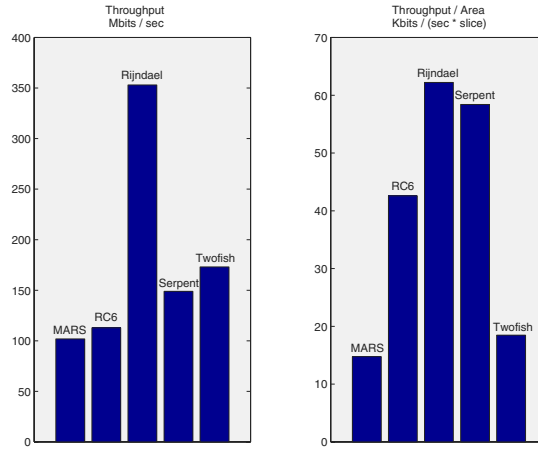


Clearly, *Rijndael* and *Serpent* achieve the lowest key-setup latency times while the latency times for *RC*6 and *Twofish* are higher by a factor of 2.5. As we have mentioned in Section 4, the key-setup latency introduced by *MARS* is the highest. All the algorithms (except *MARS*) achieve key-setup latency time that is equal to the 7-25 % of the time for encrypting one block of data.

In Table 7, throughput comparisons are made among our FPGA implementations. The comparisons are made in terms of the encryption rate and the ratio of the encryption rate to the area requirements. The latter metric reveals the hardware utilization efficiency of each implementation.

Rijndael achieves the highest encryption rate due to the ideal match of its algorithmic characteristics with the hardware characteristics of FPGAs. In addition, the encryption rate of *Rijndael* is higher than the ones achieved by the other algorithms by a factor of $1.7 - 3.12$. Moreover, *Rijndael* also achieves the best hardware utilization. The latter metric combines, for each algorithm, the computational demands in terms of an FPGA implementation with the inherent parallelism of the cryptographic round.

Table 7. Throughput Comparisons of Our FPGA Implementations



Serpent achieves the second best hardware utilization while having the lowest encryption time per round. The latter suggests that, under the same area constraints, *Serpent* can achieve throughput equivalent to *Rijndael* for operation modes that allow concurrent processing of multiple blocks of data. Similar to *Rijndael*, the algorithmic characteristics of *Serpent* matches extremely well with the hardware characteristics of FPGAs.

Finally, in Table 8, area comparisons are made among our FPGA implementations. The comparisons are made in terms of the total area as well as the area required by each of the key-setup and the cryptographic core circuits. *Serpent* and *RC*6 have the most compact implementations. *Serpent* also has the most compact cryptographic core circuit while *RC*6 has the most compact key-setup circuit. For the *MARS* block cipher, the result shown is based on an implementation that does not include the circuit for modifying the multiplication key-words [6].

5.1 Related Work

In [9,11], FPGA implementations of the AES candidate algorithms were described using Virtex devices. However, only the cryptographic core for each algorithm was implemented. No results regarding key-setup were provided. In Table 9, throughput results for [9,11] and our work are shown for encrypting 128-bit data blocks using 128-bit keys. To make a fair comparison, the results shown for [9] correspond to the performance evaluation for feedback modes. In [9], results for non-feedback modes were also provided, which corresponded to implementations that process multiple blocks of data concurrently.

The major difference in the throughput results is the *Serpent* algorithm. By implementing 8 rounds of the algorithm [9], the distribution of the subkeys among consecutive rounds becomes very efficient resulting in 3× speed-up

Table 8. Area Comparisons of Our FPGA Implementations

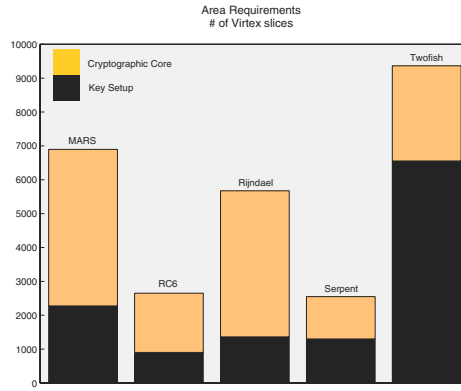


Table 9. Performance Comparisons with FPGA Implementations [9,11]

AES Algorithm	Throughput MBits/sec		
	[9]	Our	[11]
MARS	---	101.88	39.80
RC6	126.50	112.87	103.90
Rijndael	300.10	353.00	331.50
Serpent	444.20	148.95	339.40
Twofish	119.60	173.06	177.30

compared with our "single-round" implementation. For *MARS*, our implementation achieved higher throughput by a factor of 2.5 compared with [11]. The *MARS* block cipher was not implemented in [9]. For *RC*6 and *Rijndael*, all the implementations achieved similar throughput performance. For *Twofish*, the throughput achieved in [11] and in our work is higher than the one in [9] by a factor of 1.5. By combining the throughput results provided in [9,11] and the performance results provided in our work, we can verify that *Rijndael* and *Serpent* favor FPGA implementations *the most* among all the AES candidate algorithms.

6 Comparison with Software Implementations

Our performance results are compared with the best software-based results found in [3] and [4]. In [3], optimized assembly-language implementations on the Pentium II were described for *MARS*, *RC*6, *Rijndael*, and *Twofish*; only throughput results were provided. In [4], *ANSI C*-based implementations on a variety of

Table 10. Performance Comparisons with Software Implementations [3,4]

AES Algorithm	Throughput			Key-Setup Latency		
	MBits/sec		Speed-up	µs		Speed-up
	Software	Our		Software	Our	
MARS	[3] 188.00	101.88	1/1.84	[4] 8.22	1.96	4.19
RC6	[3] 258.00	112.87	1/2.28	[4] 3.79	0.17	22.29
Rijndael	[3] 243.00	353.00	1.45	[4] 2.15	0.07	30.71
Serpent	[4] 60.90	148.95	2.44	[4] 11.57	0.08	144.62
Twofish	[3] 204.00	173.06	1/1.17	[4] 15.44	0.18	85.78

platforms were described for all the AES candidate algorithms; both throughput and key-setup time results were provided.

In Table 10, throughput and key-setup latency comparisons are shown for encrypting 128-bit data blocks using 128-bit keys. Clearly, the FPGA implementations achieve significant reduction in the key-setup latency time by a factor of $4 - 144$. In software implementations, the cryptographic process can not commence before the key-setup process for all the rounds is completed. As a result, the key-setup latency time equals to the key-setup time making key-context switching inefficient. On the contrary, in FPGAs, each cryptographic round can commence as early as possible since the key-setup process can run concurrently with the cryptographic process. As a result, minimal latency can be achieved.

Regarding throughput results, the software implementations achieve higher throughput by a factor of 1.84, 2.28, and 1.17 for $MARS$, $RC6$, and $Twofish$ respectively. The latter algorithms require multiplication operations. Our intuition is that the hardware specialization and parallelism exploited in FPGAs were not enough to outperform the efficiency of the multiplication in software. On the contrary, the FPGA implementations achieved higher throughput by a factor of 1.45 and 2.44 for *Rijndael* and *Serpent* respectively. The latter reconfirms that *Rijndael* and *Serpent* favor FPGA implementations *the most* among the AES candidate algorithms. It is also worthy to mention that *Rijndael* results in one of the fastest implementations in both software and FPGAs. Finally, for operation modes that allow concurrent processing of multiple blocks of data (e.g., non-feedback, interleaved), the parallel fashion of computing in FPGAs can result in higher throughput for all the AES candidate algorithms compared with uniprocessor-based software implementations.

7 Comparison with ASIC Implementations

Our performance results are also compared with the results of ASIC-based implementations described in the NSA's "Hardware Performance Simulations of Round 2 AES Algorithms" [17]. Our time performance results are compared

with the results provided for encrypting 128-bit data blocks using 128-bit keys using iterative architectures. In Table 11, throughput and key-setup latency comparisons are shown for encrypting 128-bit data blocks using 128-bit keys. Clearly, besides our implementations, *Rijndael* achieves the highest throughput in ASICs too. Surprisingly enough, the FPGA implementations for *MARS*, *RC*6, and *Twofish* achieve higher throughput than the ASIC-based counterparts. For one reason, since ASIC technology can provide the ultimate performance, we assume that the resulted speed-ups are due to the design techniques (e.g., inherent parallelism) and the individual components (e.g., multiplier) incorporated in our implementations. For another, the Virtex FPGAs are fabricated on a leading edge 0.18μm, six-layer metal silicon process [16], while a 0.5μm MOSIS-specific technology library was utilized in [17]. Regarding the key-setup latency time, the only major difference is the *RC*6 algorithm where an improvement by a factor of 33.76 has been achieved.

Table 11. Performance Comparisons with ASIC Implementations [17]

AES Algorithm	Throughput			Key-Setup Latency		
	MBits/sec		Speed-up	µs		Speed-up
	NSA ASIC	Our		NSA ASIC	Our	
MARS	56.71	101.88	1.79	9.55	1.96	4.87
RC6	102.83	112.87	1.09	5.74	0.17	33.76
Rijndael	605.77	353.00	1/1.71	0.00	0.07	---
Serpent	202.33	148.95	1/1.35	0.02	0.08	1/4
Twofish	105.14	173.06	1.64	0.06	0.18	1/3

8 Conclusions

In this paper we have provided time performance and area requirements results for the implementations of the five final AES candidates (*MARS*, *RC*6, *Rijndael*, *Serpent*, and *Twofish*) using FPGAs. To the best of our knowledge, we are not aware of any published results that include key-setup latency results. In our implementations, the key-setup process can be performed in parallel with the encryption process regardless of the capability of the software implementation to support on-the-fly key setup. Our implementations suggest that *Rijndael* and *Serpent* favor FPGA implementations *the most* due to the ideal match of their algorithmic characteristics with the characteristics of FPGAs. The *Rijndael* implementation achieves the lowest key-setup latency time, the highest throughput, and the highest hardware utilization. Comparing our results with software [4,3] and ASIC [17] implementations, we verified that *Rijndael*

also achieves the best time performance across different platforms (i.e., ASIC, FPGA, software).

The work reported here is part of the USC MAARCII project (`http://maarcII.usc.edu`). This project is developing novel mapping techniques to exploit dynamic reconfiguration and facilitate run-time mapping using configurable computing devices and architectures. A domain-specific mapping approach is being developed to support instance-dependent mapping [8]. Moreover, computational models and algorithmic techniques are being developed to exploit self-reconfiguration using FPGAs. Finally, the idea of "active" libraries is exploited to develop a framework for automatic dynamic reconfiguration.

References

1. Advanced Encryption Standard, `http://www.nist.gov/aes/`
2. R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", AES Proposal, June 1998.
3. K. Aoki and H. Lipmaa, "Fast Implementations of AES Candidates", Third AES Candidate Conference, April 2000.
4. L. E. Bassham III, "Efficiency Testing of ANSI C Implementations of Round 2 Candidate Algorithms for the Advanced Encryption Standard", Third AES Candidate Conference, April 2000.
5. S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial", IEEE Design & Test of Computers, Summer 1996.
6. C. Burwick et al., "MARS - a candidate cipher for AES", AES Proposal, August 1999.
7. J. Daemen, V. Rijmen, "The Rijndael Block Cipher", AES Proposal, September 1999.
8. A. Dandalis, "Dynamic Logic Synthesis for Reconfigurable Devices", PhD Thesis, Dept. of Electrical Engineering - Systems, University of Southern California. Under Preparation.
9. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists", Third AES Candidate Conference, April 2000.
10. D. Fowler, "Virtual Private Networks: Making the Right Connection", Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.
11. K. Gaj and Pawel Chodowiec, "Comparison of the hardware performance of the AES candidates using reconfigurable hardware", Third AES Candidate Conference, April 2000.
12. R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6^{TM} Block Cipher", AES Proposal, June 1998.
13. J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field Programmable Gate Arrays", Proceedings of the IEEE, July 1993.
14. B. Schneier, "Applied Cryptography", John Willey & Sons, Inc., 2nd edition, 1996.
15. B. Schneier, J. Kelsey, D. Whitingz, D. Wagnerx, and C. Hall, "Twofish: A 128-Bit Block Cipher", AES Proposal, June 1998.
16. Virtex Series FPGAs, `http://www.xilinx.com/products/virtex.htm`.
17. B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke, "Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms", Third AES Candidate Conference, April 2000.