CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2011; **23**:372–389
Published online 4 November 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1647

Efficient mapping and acceleration of AES on custom multi-core architectures

Amit Pande*, † and Joseph Zambreno

Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, U.S.A.

SUMMARY

Multi-core processors can deliver significant performance benefits for multi-threaded software by adding processing power with minimal latency, given the proximity of the processors. Cryptographic applications are inherently complex and involve large computations. Most cryptographic operations can be translated into logical operations, shift operations, and table look-ups. In this paper we design a novel processor (called μ-core) with a reconfigurable Arithmetic Logic Unit, and design custom two-dimensional multi-core architectures on top of it to accelerate cryptographic kernels. We propose an efficient mapping of instructions from the multi-core grid to the individual processor cores and illustrate the performance of AES-128E algorithm over custom-sized grids. The model was developed using Simulink and the performance analysis suggests a positive trend towards development of large multi-core (or multi-μ-core) architectures to achieve high throughputs in cryptographic operations. Copyright © 2010 John Wiley & Sons, Ltd.

Received 23 January 2009; Revised 22 February 2010; Accepted 8 July 2010

KEY WORDS: μ-core; micro-programming; multi-core processors; cryptography

1. INTRODUCTION

A cryptographic algorithm is an essential part of secure communication. In November 2001, the National Institute of Standards and Technology (NIST), United States chose a new encryption standard called the Advanced Encryption Standard (AES) to replace the existing Data Encryption Standard (DES) algorithm. Since then, AES has become the industry standard for many commercial cryptographic applications. The computational power required to encrypt data using AES can significantly limit the throughput of encryption algorithm, and thus security is often sacrificed for throughput in real-time applications.

Parallelism can provide high computation power and can be used in principle to accelerate encrypted transfers. Parallel architectures such as multi-core processors offer a large amount of concurrency to improve the throughput of different applications including encryption. With this motivation, in this paper we present a new architecture for efficient cryptography in this work. The architecture is essentially a two-dimensional (2D) grid of individual small processing cores—with each core communicating with the nearest neighbors in four directions (East, West, North, and South) only. The individual processor core is called as a μ-core owing to its small size and simple design. Each individual core has an Arithmetic Logic Unit (ALU) for simple logical and shift operations, a register bank, a high-speed on-chip memory, and some control logic. We observe

*Correspondence to: Amit Pande, Electrical and Computer Engineering Department, 2215 Coover Hall, Ames, IA-50011, U.S.A.
†E-mail: amit@iastate.edu

that many private-key cryptographic algorithms can be broken into shifts, logic operations, and look-up operations. Consequently, a simple ALU design is sufficient for this class of application.

In this paper we provide a study of the performance of cryptographic algorithms (e.g. AES) on parallel μ-core 2D grid architectures customized for efficient implementation in terms of throughput, hardware usage, and power consumption. The main contributions of this paper are summarized as follows:

- We introduce the implementation of cryptographic algorithms over efficient parallel architectures such as a μ-core array.
- We introduce the concept of a crypto-ALU, which is a reconfigurable block that can be configured to perform various arithmetic and logical tasks. Reconfiguration can be used to map different requirements of various cryptographic algorithms (e.g. encryption, hashing) into this ALU.
- The AES-128E algorithm is mapped into a 2D μ-core array and its performance has been reported in terms of efficiency and throughput.
- We present a scheme for microprogramming the 2D multi-core arrays. An efficient mapping of macroinstructions from 2D arrays into microinstructions for individual cores is discussed.

The paper is organized as follows: Section 2 provides a brief overview of the AES algorithm. In Section 3, we give an overview and design features of μ-core processor followed by an introduction to multi-μ-core (MMC) array (built with individual μ-cores) in Section 4. In Section 5, we present a scheme for microprogramming in MMC arrays and illustrate how macroinstructions can be mapped to individual microinstructions for individual cores. In Section 6, we examine the implementation of the AES algorithm and its building blocks efficiently over custom-sized MMC arrays, followed by simulation results in Section 7, Conclusions and future work are provided in Section 8.

2. OVERVIEW OF AES

Security, simplicity, and suitability to both hardware and software implementations. The Rijndael algorithm, which was developed by Rijmen and Daemen [1] was selected as the AES competition winner in 2000.

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits [2]. Many hardware- and software-based implementations of the AES algorithm and its variants have been reported in the research literature [3–7]. The main contribution of this work is the design of a novel multi-core processor model that can efficiently map cryptographic operations. We demonstrate an efficient mapping of macroinstructions into μ-instructions for individual μ-cores. An individual μ-core can reconfigure itself to implement the requested logic/arithmetic operation. This will become evident in the following sections. The main operations in AES-128E are shown in Figure 1 and enumerated below:

1. The *SubBytes* transformation is a nonlinear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box is invertible and constructed by composing two transformations: (1) taking the multiplicative inverse and (2) applying an affine transformation in the finite field GF(2^8).
2. *ShiftRows* cyclically shifts the bytes in the last three rows of the state. The second, third, and fourth rows are shifted by one, two, and three bytes to the left.
3. *MixColumns* transformation separately modifies each column of the state in what is essentially a matrix multiplication operation. In the 8-bit finite mathematical field (Galois field), the entire operation can be reduced to shifts and XOR operations.
4. *AddRoundKey* adds the round key to the state using a bitwise XOR operation.

More details regarding AES algorithm are available in the original AES standardization documentation by NIST [1] and the developers' own writing [2].

3. μ-CORE ARCHITECTURE

We use the term micro-core or μ-core to refer to a compact, simple processor core designed to perform specific tasks. It is a small core with few registers, small memory and a simple ALU. We refer to the sequence of code used to program the μ-core to implement different applications as μ-code. This μ-code is a set of very detailed and rudimentary lowest-level routines which controls and sequences the actions needed to execute (perform) particular instructions, sometimes also to decode (interpret) them. Designing the control as a program that implements the machine instructions in terms of simpler microinstructions is called microprogramming [8].

3.1. The requirements of a μ-core processor

We first study the essential design features of a μ-core. Register banks, an instruction and data memory, input and output ports are essential to any processor. Execution units are also essential in order to perform operation or calculations in the processor. Execution units include ALUs, floating point units (FPU), load/store units, and branch prediction. The choice of the number of execution units, their latency and throughput is a central micro-architectural design task. The size, latency, throughput, and connectivity of memories within the system are also micro-architectural designs. This includes decisions on the performance level and connectivity of these peripherals. As micro-architectural design decisions directly affect what goes into a system, attention must be paid to such issues as:

1. Chip area/cost
2. Power consumption
3. Logic complexity
4. Connectivity/interconnect
5. Performance of targeted applications.

The individual μ-cores have minimal basic functionalities and a reconfigurable ALU which reduce the chip area and keep the power consumption low. Reconfigurable ALU has a reconfigurable part which can be programmed on-the-fly to implement different operations (such as multiplication or constant division or table look-up) at different instances of time. This leads to area savings over a traditional architecture where all such functionalities are separately mapped to physical hardware. Configurable processors provided by Stretch [9] also have such a feature where a part of ALU can be re-configured to implement different complex functions. Area is a complex function of the number of μ-cores, number of buffers, interconnect pattern, and the available technology. In this paper, we have discussed different configurations of MMC array (arrays of grid size 4×4 to grid size 64×128) . The knowledge of area measurement corresponding to each configuration is outside the scope of this paper. However, the number of μ-cores can be used as an indicator of the area required by a particular MMC array. To avoid any overlapping among the interconnections, they can be laid on a plane like the planer 2D-systolic arrays [10, 11]. This motivates us to keep the design planar. We keep I/O ports at the four ends of the 2D array which allows us to keep interconnect-delays small and uniform. This, in turn, allows a higher operating frequency of the design.

The performance of targeted applications (such as AES) is separately measured in this paper.

3.2. Overview of μ-core processor model

An architectural overview of our μ-core processor design is presented in Figure 2. There are four input and four output ports allowing the easy integration of the individual core into a 2D array form. The four inputs and output ports correspond to the East, West, North, and the South data transfer ports for the μ-core. The design consists of an input multiplexer, an output demultiplexer, a register bank, a 64 (64×1) byte memory, a small Crypto-ALU, a control word (CW) decode mechanism, and a register bank input select multiplexer. Each individual μ-core has a 64-byte high speed memory for data storage. The CW is issued by the instruction cache and read by the processor every cycle. The register bank consists of eight registers $R0$, $R1$,

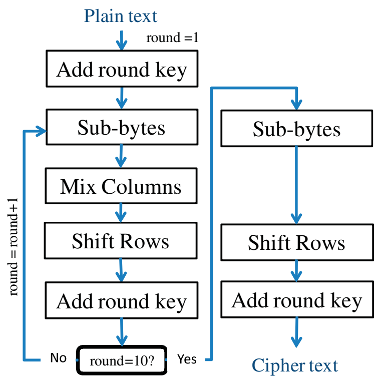

Figure 1. Overview of AES-128E encryption scheme.

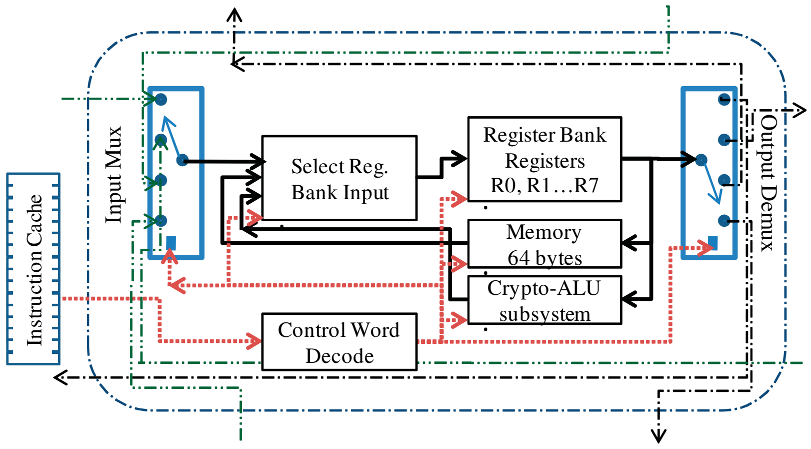

Figure 2. Architectural overview of a μ-core.

$R2 \dots R7$, each register being 8 bits wide. The input multiplexer muxes the four inputs into one selecting the input port as indicated in the CW. The output demultiplexer demuxes the output into the designated output port with the help of control logic. The Crypto-ALU is a reconfigurable ALU block giving flexibility to implement specific operations on Reconfigurable logic, such as application-specific table look-up or logical operations. As most of the cryptographic operations are done in Galois field mathematics, they are easily translated into logical and look-up operations. To allow for design flexibility, we implement ALU operations using LUT-based Reconfigurable hardware.

The CW is decoded by the decoding logic. The CW of the processor is 11 bits in size. The bits in CW are directly mapped to their specific functionality making it simple to implement the decoding logic. The following subsection will explain the CW format in more detail.

3.3. *CW description*

The CW format of our μ-core processor is given in Table I. The CW is 11 bits long (bits are numbered (0–a) from right to left). The ith bit of CW is denoted by b_i in Table I and we use the same convention in the rest of our discussion. Thus, b_8 refers to eighth bit (from right) in CW. There are eight registers in register bank R and can therefore be addressed by 3 bits. In Table I, R_a, R_b, and R_c are each three bits wide values and are used to refer to register bank registers. For example when $R_a = 3$, $R[R_a]$ denotes $R3$ of the register bank. Adding to the simple design, all the instructions execute in one cycle.

Table I. The ISA for μ-core processor.

Bit a	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Syntax
Arithmetic instructions											
0	0	$R_c(3)$	$R_c(2)$	$R_c(1)$	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_c] \ll \text{AND}(R[R_b], R[R_a)]$
0	1	$R_c(3)$	$R_c(2)$	$R_c(1)$	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_c] \ll XOR(R[R_b], R[R_a])$
1	0	$R_c(3)$	$R_c(2)$	$R_c(1)$	$R_b(3)$	$R_b(2)$	$R_b(1)$	0	0	0	$R[R_c] \ll \text{LookUp}(R[R_b])$
1	0	$R_c(3)$	$R_c(2)$	$R_c(1)$	$R_b(3)$	$R_b(2)$	$R_b(1)$	0	1	1	$R[R_c] \ll \text{ShiftLeft}(R[R_b], 1)$
1	0	$R_c(3)$	$R_c(2)$	$R_c(1)$	$R_b(3)$	$R_b(2)$	$R_b(1)$	1	0	0	$R[R_c] \ll \text{ShiftRight}(R[R_b], 1)$
1	1	1	1	1	1	1	1	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_a] \ll R[R_a]+1$
1	1	1	1	1	0	0	0	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_a] \ll R[R_a]-1$
Input/output instructions											
1	1	0	0	1	$R_b(3)$	$R_b(2)$	$R_b(1)$	0	b1	b0	$R_b \ll$INPUT(SelectPort)*
1	1	0	0	1	$R_b(3)$	$R_b(2)$	$R_b(1)$	1	b1	b0	SelectPort$\ll$OUTPUT(R_b)*
Memory read/write instructions											
1	1	1	0	0	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_a] \ll \text{Mem}(R[R_b])^{\natural}$
											$\natural$ if R_b==7 then $R[R_b]=R[R_b]-1$
1	1	0	1	0	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$\text{Mem}(R[R_a]) \ll R[R_b]^{\aleph}$
											$\aleph$ if R_a==7 then $R[R_a]=R[R_a]+1$
Register transfer instructions											
1	1	1	0	0	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_b] \ll R[R_a]$

*b1 $b0$ = 00, 01, 10, and 11 selects East, West, North, and South input/output ports, respectively.

Table II. Direction selection for the I/O instructions.

Direction	Bit 1	Bit 0
East	0	0
West	0	1
North	1	0
South	1	1

3.3.1. *Arithmetic instructions.*

The value of bit $(b_a, b_9) = (00, 01, 10)$ in the CW refers to the reconfigurable instructions in the Crypto-ALU. Thus, there is a flexibility in the design to provide custom bitwise operations. The CW is not fixed and can be customized for each μ-core to suit the application requirements.

The custom operations implemented for the present experiments (implementation of AES-128E algorithm) are given in Table I. CW $b_a = 0, b_9 = 0$ denotes bitwise AND operation whereas $b_a = 0, b_9 = 1$ denotes bitwise XOR operation. The look-up operation $(R[R_c] \ll \text{LookUp}(R[R_b]))$ is implemented using an eight bit (256 element) look-up table. There are functions for left shift, right shift, and increment/decrement operations.

Example: The CW 00100101110 XORs the inputs in registers $R5$ and $R6$ into register $R4$. CW 11111000010 decrements the value of register $R2$ by one at the beginning of the next cycle. CW 10101010100 right shifts the contents of register $R2$ and saves them to $R5$ at the beginning of the next cycle.

3.3.2. *Input/output instructions.*

The μ-core processor has four input and four output ports corresponding to the East, West, North, and South directions, respectively. The bits b_1 and b_0 decide the input/output port as explained in Table II.

Example: The CW 11001011011 inputs the data from South to register $R3$ $(R3 \ll \text{INPUT(South)})$ and CW 11001111110 outputs the contents of register $R7$ to North port $(\text{North} \ll \text{OUTPUT}(R7))$.

3.3.3. Memory access instructions. The μ-core processor has a small 64 byte high speed memory. The CWs for MEMory Read (MEMR) and MEMory Write (MEMW) instructions are given in Table I. If the register $R7$ is selected it is auto-incremented during MEMW and auto-decremented during MEMR to facilitate sequential read from/write to memory.

Example: The CW 11100011010 indicates the operation $R2 \ll \text{Mem}([R3])$, i.e. the contents to memory location indicated by the value of R3 are written to register $R2$. The value of register $R3$ are calculated modulo 64 to prevent any overflow error.

3.3.4. Register transfer instructions. The CW $11100R_bR_a$ copies the value of R_a to R_b.

Example: CW 11100010000 copies $R0$ to register $R2$.

4. INTRODUCTION TO THE MMC ARRAY

Given their simple and highly scalable design, μ-core processors can be used in clusters to achieve high throughput and efficiency. By design, they have four input and four output ports giving the possibility of an efficient 2D array implementation. This is illustrated in Figure 3 where we use individual μ-cores to achieve a 4×4 array structure. In this paper, we restrict our initial discussion to a 4×4 MMC array structure although the discussion is valid for any other array structure, such as 8×4 or 8×8.

5. EFFICIENT MAPPING OF MACROINSTRUCTIONS

The ISA for μ-core cores can be looked at as a horizontal microcode [12], where there is a fairly direct correspondence between the bit fields in a microinstruction and the control signals sent to the various parts of the CPU, allowing for a simple design of control-decoding logic. A translation of macroinstructions for the MMC array to microinstructions for individual cores is required to facilitate mapping of algorithms to MMC array. An automatic mapping of macroinstructions into microinstructions is illustrated in the following subsections. We consider some simple MMC commands and their translation logic for an $M\times N$ MMC array (here $M=N=4$) for sample implementation, where M equals the number of rows whereas N equals the number of columns.

5.1. *Row/col shift left/right/top/bottom*

Consider the macroinstruction *ShiftMxN*(Left, a, B, i) for an $M\times N$ array which (cyclically) shifts the contents of register $R[i]$ in the register bank for rows specified by vector B by a units to left.

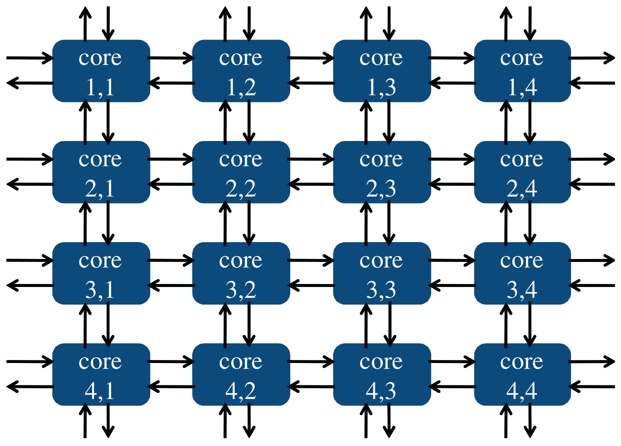

Figure 3. Overview of 4×4 MMC grid built over μ-core as individual cores.

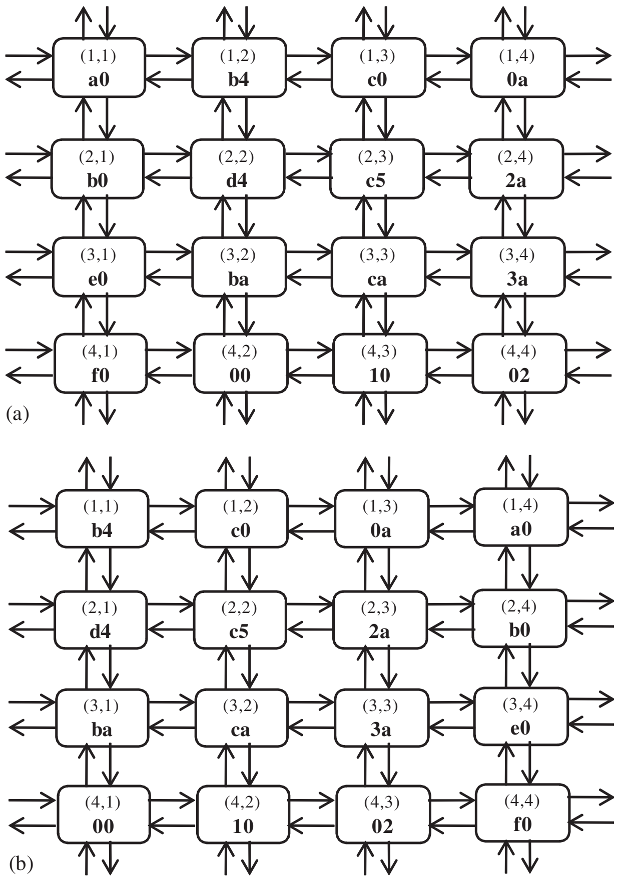

Figure 4. A left shift oper6ion for MMC: (a) initial state and (b) final state.

For example, *Shift4x4*(Left, 1, [1, 2, 3, 4], 3) implements a circular right shift for the contents of all register $R3$ in all four rows by 1 to right. This is illustrated in Figure 4. The ID of each processor is mentioned along with the register $R3$ value which is boldfaced. To map this macroinstruction into microinstructions, we make the following observations:

1. The operand i needs to be coded in the register address of each microinstruction.
2. The operand B specifies the rows to be μ-coded for this instruction. If the operand is an array with elements $[1, 2, 4]$, then we need to skip the microinstructions for row 3.
3. To efficiently map the algorithm we note that i left shift are equal to $(M-i)$ right shifts. As both left and right operations take equal cycles, we need to choose the lower of i and $(M-i)$.
4. One left shift (and essentially one right shift) operation requires a sequence of operations.
   (a) Even numbered μ-cores transfer their values to some temporary register in odd μ-cores.
   (b) All odd μ-cores transfer their value to some temporary registers in even numbered μ-cores (except the first μ-core).
   (c) All μ-core (except the first) shift the value from temporary registers to the designated registers.
   (d) The first μ-core transfers its register value to the last processor via sequential steps using temporary registers of the intermediate μ-cores.

The same procedure can be used to map right shifts. In the case of top or bottom shifts the argument proceeds similarly, except that now we operate one column at a time.

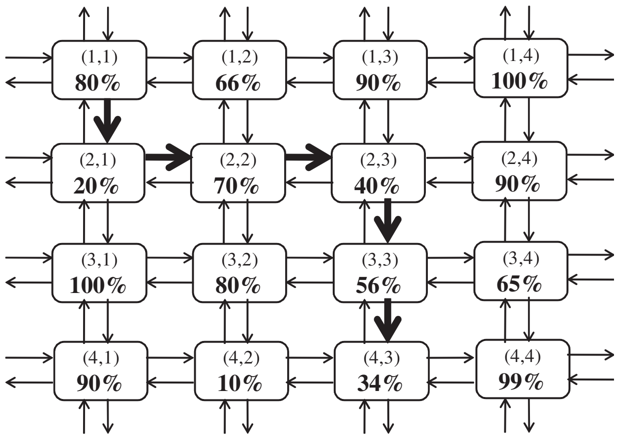

Figure 5. Routing data from μ-core (1,1) to (4,3).

5.2. *Add GF*(2^8)

We consider addition of large words represented in $M \times N$ (here $4 \times 4 = 16$) bytes in the MMC processor. The instruction $AddMxN(R_a, R_b)$ adds the contents of register R_a to that of register R_b. For example, *Add4x4*($R2, R4$) adds the 16 bytes unsigned integer stored in 16 μ-cores in register $R2$ to the 16-byte integer represented by register $R4$. The Galois field arithmetic used in most cryptographic operations makes it easy to implement each addition as a bitwise XOR between registers. Therefore, the macroinstruction $Add4x4(R2, R4)$ is translated into the instruction bitwise $XOR(R2, R4)$ for each individual μ-core.

5.3. *Routing data*

Consider the macroinstruction *RouteMxN*($i1, j1, R_a, i2, j2, R_b$). Here, we need to route the data in register R_a in μ-core ($i1, j1$) to register R_b in μ-core ($i2, j2$) with ($i1, i2 < M$; $j1, j2 < N$). To map the macroinstruction into microinstructions, we proceed as follows:

1. Choose the least utilized μ-core from the following two (i^*, k^*).
   (a) μ-core ($i1 + \text{sgn}(i2 - i1)$, $j1$)
   (b) μ-core ($i1$, $j1 + \text{sgn}(j2 - j1)$).

   Here sgn(x) is the signum function giving direction to the flow of data.
2. Transfer the data to an unused register R_* for new μ-core (i^*, k^*).
3. Rewrite the routing problem as Route(i^*, j^*, R_*, $i2$, $j2$, R_b) and solve iteratively.

The mentioned routing algorithm may not always lead to an optimal solution, and better algorithms may exist to find the easiest path. However, it gives a near-optimal route workable in most scenarios. Figure 5 illustrates the implementation of *Route4x4*($1, 1, R1, 4, 3, R3$). The processor utilization is boldfaced for each μ-core. In this example, μ-core (1,1) chooses μ-core(2,1) over μ-core(1,2) to transfer data because of lesser resource utilization by μ-core(2,1). Then, the routing is iterated and the data reaches μ-core(4,3) via μ-cores (2,2), (2,3), and (3,3), respectively. Thus, the translator needs to keep track of individual μ-core utilization, and available registers.

5.4. *Logical shift for $M \times N$ bytes word*

Next, we consider *ShiftMxN*(R_a, i). This macroinstruction implements a logical shift of i bits for the $M \times N$ byte data stored in registers R_a. We consider the translation of the macroinstruction $Shift4x4(R2, 2)$. The following steps are involved:

1. For $0 < i \leq 8$, first transfer the R_a entry of μ-core(i, j) to the temporary register R_* of μ-core($i, j-2$).

2. Subsequently, μ-core $(i, 1)$ will transfer the R_a entry to μ-core$(i-1, N)$.
3. XOR R_* with a suitable mask to get the upper i bits and (logical) shift right (8-i) bits to position them in the lower i position.
4. Shift left the R_a contents and XOR them with the output of the previous step.
5. For $8*j<i\leq(8*j+8)$, we have to shift the contents of μ-core(i, j) to μ-core$(i, j-2)$ and proceed accordingly.

6. AES IMPLEMENTATION ON 4×4 MMC PROCESSOR

The 4×4 MMC processor has 16 input ports: four each to East, West, North, and South as shown in Figure 3. To implement AES or any other algorithm, we need to insert PlainText or any other data through these ports into the processors. To measure the performance for a single iteration of AES-128E we first loaded the PlainText into register $R0$ of each μ-core. The 16-byte PlainText was thus the distributed register $R0$ of each 16 μ-core. The eleven keys from key expansion stage were loaded into the 64-byte memory and subsequently read into register $R2$.

The following subsections explain the mapping of AES macroinstructions into microinstructions for individual μ-cores.

6.1. Add round key

The AddRoundKey operation is similar to *Add GF*(2^8) instruction except that it first involves reading the value from memory to register $R2$. Thus, two μ-instructions are issued to each core for AddRoundKey:

1. Load Key from memory to register $R2$
2. XOR the contents of $R0$ and $R2$ and return the result to $R0$.

The first instance of AddRoundKey has the key already loaded, therefore eleven instances of AddRoundKey require 21 cycles.

6.2. SubBytes

The bytes substitution operation is implemented using a 256 bytes look-up table. Thus, it takes one cycle for a single SubByte operation.

6.3. ShiftRows

No operation is required for the first row. The operation for the second row takes five cycles as illustrated in Figure 6 and summarized as follows:

1. μ-core (2,1) and (2,4) send R_0 contents to the two center cores which store them in register R_1.
2. μ-core (2,2) transfers R_1 value to (2,3) which stores it into register R_2.
3. μ-core (2,2) and (2,3) send their R_1 and R_2 values to external processors (2,1) and (2,4) which store these final values into R_0.
4. μ-core (2,3) transfers R_0 value to (2,2) which stores the final value to register R_0.
5. μ-core (2,3) transfers its final value from register R_1 to register R_0.

The operation for the third row takes six cycles and is explained with the help of Figure 7 and is summarized as follows:

1. In two cycles, (3,3) transfers its value to (3,1) register R_1 which is later shifted to register R_0. Simultaneously, (3,4) shifts its value to register R_0 of (3,3) in one cycle.
2. In the next two cycles, first (3,2) transfers its value to (3,3) to its register R_1 and then it gets the final value to its register R_0.
3. In two subsequent cycles, (3,1) transfers its R_0 value to (3,3). In parallel operations with these two cycles, we perform two operations.
   - In the first cycle (3,4) receives its final value to register R_0.
   - In the second cycle (3,1) transfers its final value from register R_1 to R_0.

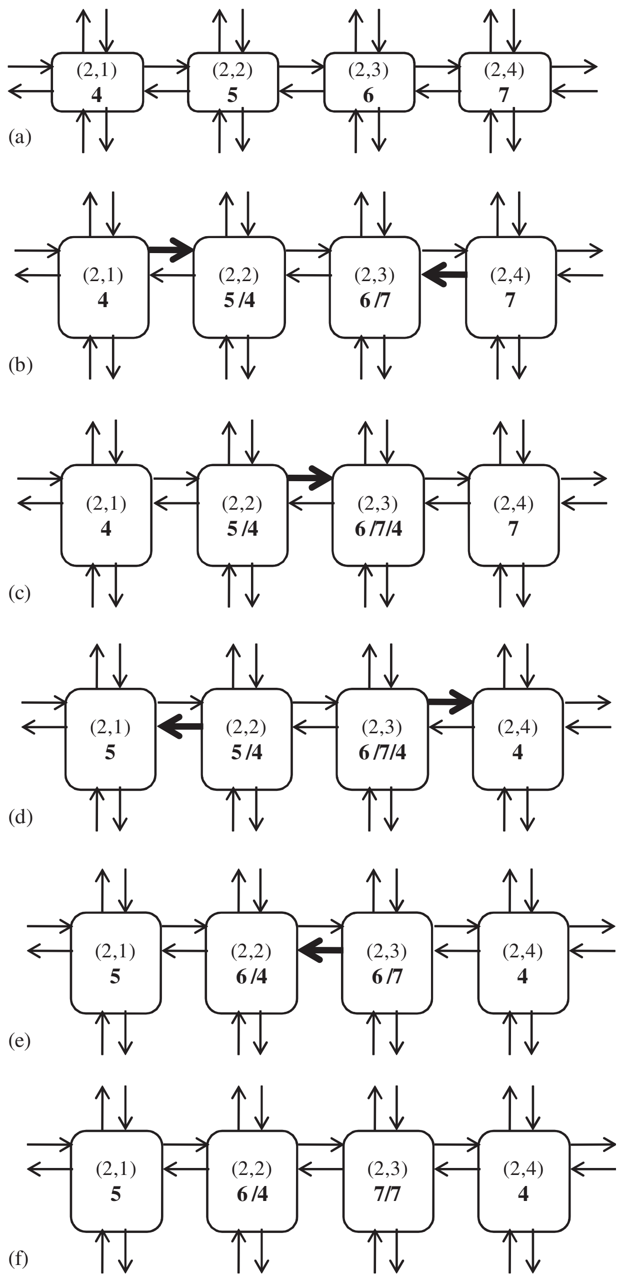

Figure 6. Illustration of ShiftRow operation for second row: (a) initial state; (b)-(e) intermediate stages; and (f) final state.

The operation for the fourth row is similar to the second row except that it implements a right shift. The first boldfaced value is the value in register R0, the subsequent values are values in temporary registers. Thus, each ShiftRows operation requires six cycles.

6.4. MixColumns

The translation of the MixColumns step into microinstructions is done in the following manner:

1. For one column, transfer each of the four bytes into all the other three μ-cores. This step requires eight cycles.

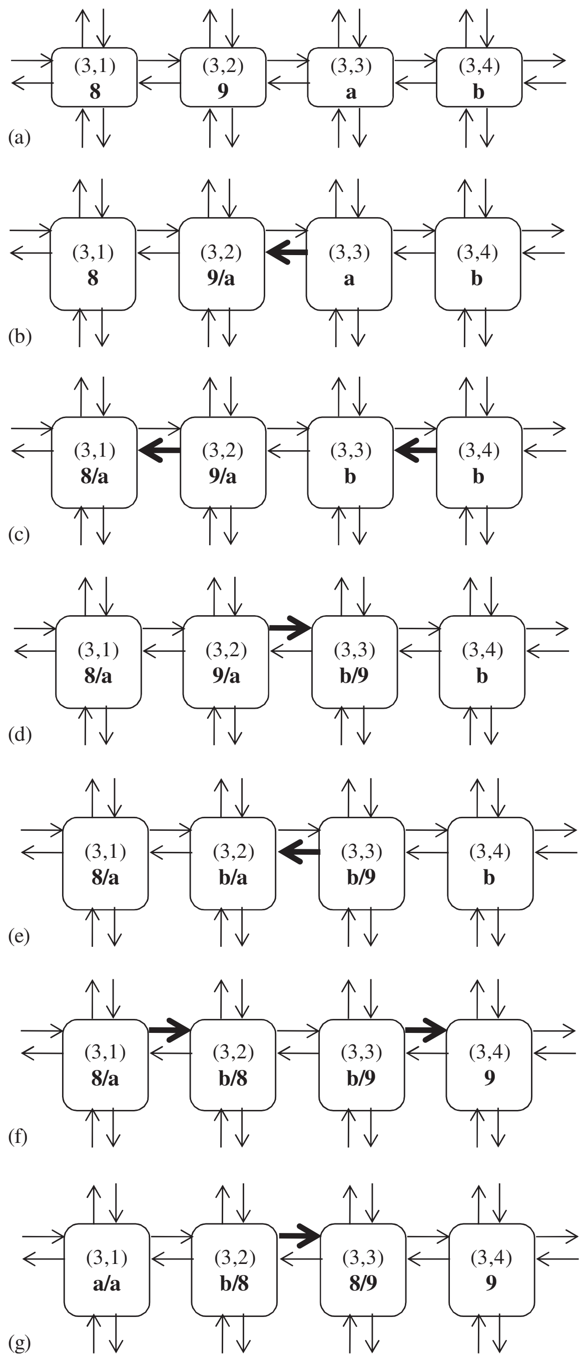

Figure 7. Illustration of ShiftRow operation for third row: (a) initial state; (b)–(f) intermediate stages; and (g) final state.

2. Thus, each processor has all four operands in its register bank in the following manner. R0 contains its original entry. $R1$ contains the entry from $((i+1)\text{mod}(4), j)$th μ-core, $R2$ contains the entry from $((i+2)\text{mod}(4), j)$th μ-core, and $R3$ contains the entry from $((i+3)\text{mod}(4), j)$th μ-core.
3. The GF(2^8) multiplication by 2 can be resolved by one logical shift and a subsequent XOR operation. This has been included in the ISA of the μ-core. Thus, we use two cycles to store the product of $R0$ and $R1$ with 2 into $R4$ and $R5$, respectively.

Table III. Break-up of cycles for AES.

Operation	No. of iterations	Total no. of cycles
AddRoundKey	11	$10\times 2+1=21$
ShiftRows	10	$10\times 6=60$
SubBytes	10	$10\times 1=10$
MixColumns	9	$14\times 9=126$
Total		$21+10+60+126=217$

4. Finally, we XOR the contents of $R0$, $R2$, $R3$, $R4$, and $R5$ to get the MixColumn output. This step takes four cycles (one cycle for each XOR operation).

The MixColumn operation is implemented in fourteen cycles in our implementation. The total number of cycles taken by the AES-128E operation in our 4×4 MMC processor is 217 as illustrated in Table III.

6.5. *Performance analysis*

Next, we performed an analysis to calculate the overhead in implementing AES on large array of μ-core processors. We observe that a 4×4 grid is well suited to integrate and utilize the algorithmic parallelism in AES. Therefore, we implemented grids of sizes in multiples of 4×4 (e.g. 4×8, 16×16, 32×32). The AES algorithm will still take the same number of cycles (217) but the I/O cycles will change. We propose a Selective Row/Columnwise Input/Output scheme to efficiently perform I/O operations.

6.5.1. Selective row/columnwise input/output scheme. Let us consider a grid of size $4m\times 4n$ and proceed as follows:

- If $m\geq n$ we input data columnwise from East and West directions, else we input rowwise from North and South directions. We present the algorithm for $m\geq n$ and similarly the algorithm can be constructed for $m<n$.
- Input the values to the nodes $(1,2n),(2,2n)\ldots(4m,2n)$ and $(1,2n+1),(2,2n+1)\ldots(4m,2n+1)$ from the East and West, respectively.
- The plain-text input to innermost nodes will arrive in $2n$ cycles.
- We pipeline the plain-text inputs to subsequent nodes starting from inside. $(1,2n-1)$, $(2,2n-1)\ldots(4m,2n-1)$ and $(1,2n+2),(2,2n+2)\ldots(4m,2n+2)$ receive plain-text in $(2n+1)$th cycle.
- The total number of I/O cycles is $(4m-1)$ or $(4.\min(m,n)-1)$ cycles.

7. SIMULATION PLATFORM

The μ-core processor, its instruction set and the multi-core architecture was simulated in Simulink, a graphical visualization and simulation tool from the MATLAB team.

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e. have different parts that are sampled or updated at different rates. For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. Simulink includes a comprehensive block library of sinks, sources, connectors and allows customized blocks [13]. Simulink workflow supports a hierarchical design allowing a top-down or a down-top design procedure. A single core model was first built using Simulink basic blocks and custom blocks generated using MATLAB M files. It was then used to create a custom grid of 4×4 processors each connected

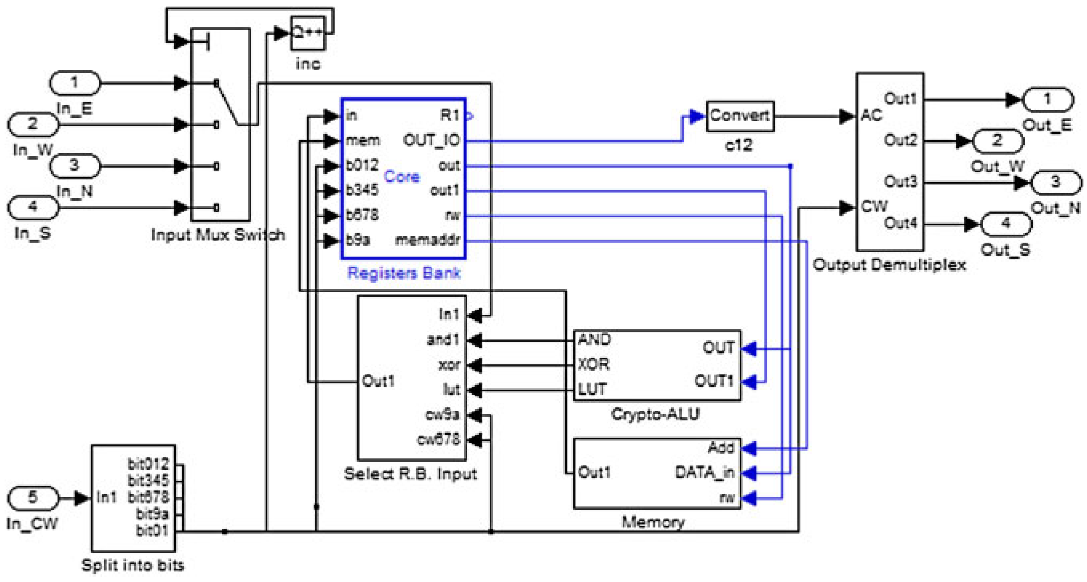

Figure 8. The single core model built on Simulink.

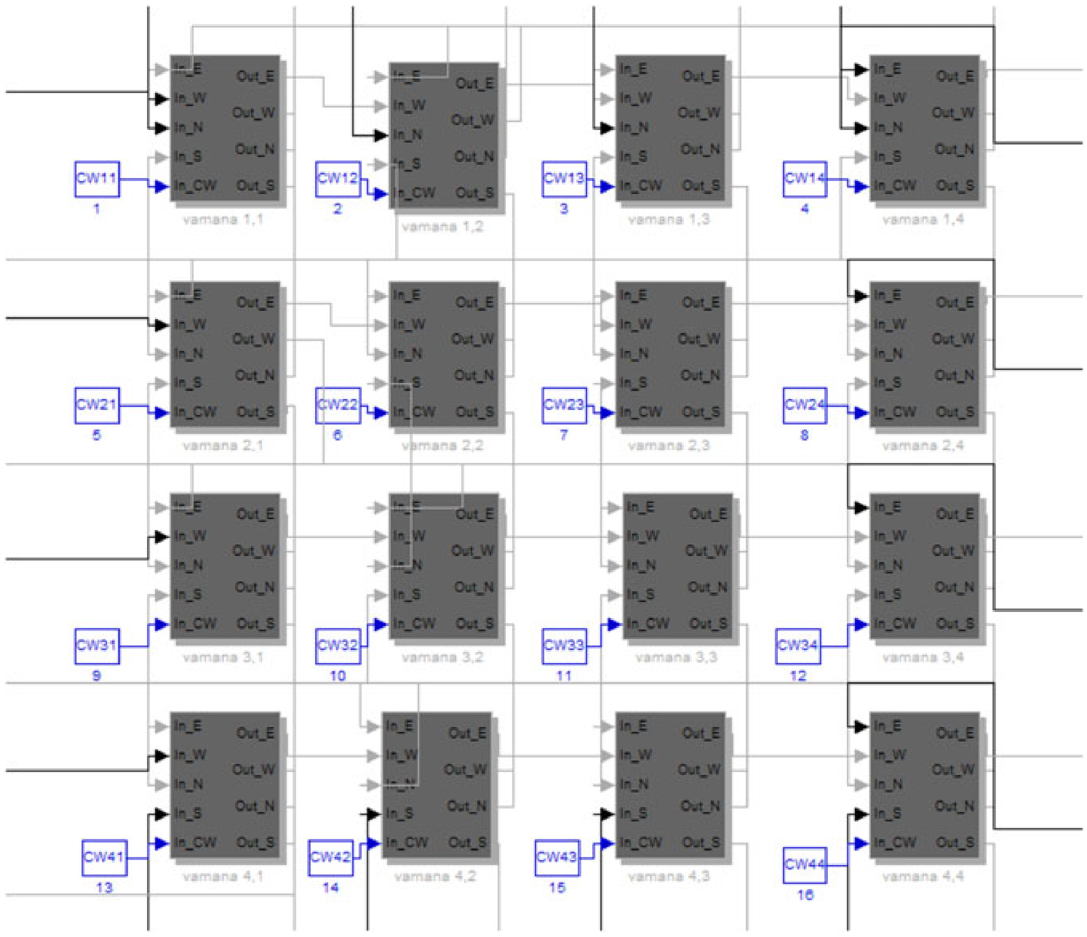

Figure 9. 4×4 Multi-core model built on Simulink.

to its neighbors in four directions (East, West, North, and South) through two ports: one each for input and output operations.

Simulink provides scopes and other display blocks to view the simulation results while the simulation is running. This assisted us in the evaluation and debugging of our design at different stages. The simulation inputs can be ported from MATLAB workspace and the results can be put back into it for postprocessing and visualization. We thus input the data inputs to the multi-core model and individual CWs to each μ-core with the help of MATLAB workspace variables.

Figure 8 gives a screenshot of the simulink model for μ-core. The inputs from four directions (labeled as *in_E*, *in_W*, *in_N*, and *in_S*, respectively) are multiplexed by the control logic to the block that selects the Register Bank input for the given cycle depending on CW. The Registers Bank outputs its values to the crypto-ALU, the memory block and to the output ports depending on the CW. The crypto-ALU can be implemented directly in Simulink or by using the Xilinx System Generator. Xilinx System Generator is a software tool (by the FPGA manufacturer company Xilinx) for modeling and designing FPGA-based DSP systems in Simulink. The tool presents a high level abstract view of a DSP system, yet nevertheless automatically maps the system to a faithful hardware implementation [14].

Figure 9 gives a screenshot of the Simulink model of the 4×4 multi-core grid built over individual μ-cores.

8. SIMULATION RESULTS

We performed an analysis of the throughput of our MMC grids for various grid layouts and sizes. We worked from the assumption that large size grid structures were feasible to implement in hardware. We also assumed that each of the individual μ-cores for the MMC array operated at the same fixed clock frequency.

The first assumption is justified because the individual μ-cores are small and compact in size, and the only interconnection required is with their nearest neighbors. Generally, in multi-core architectures, off-chip bandwidth is limited and off-chip accesses are expensive. Our architecture is more computation intensive, with only the side processors communicating with off-chip memory. Most of the communication is confined to nearest neighbor processors which help us achieve a high speed communication between μ-cores. The assumption of clock scalability may be critical for large MMC arrays, however, relevant research has been done in related areas [15, 16] which can be adapted to achieve clock synchronization for large MMC arrays.

In this work, we skip the hardware description and VLSI implementation of such architecture and instead focus on the performance analysis with different grid layouts, taking AES-128 as a sample application for the MMC grids. The entire AES-128 encryption algorithm was simulated over the Simulink model.

Figure 10 gives the variation of system throughput with the change in grid layout. When the number of processors is doubled, the throughput approximately doubles. The increase in throughput from 8×4 architecture to 8×8 grid layout accompanies an increase in the number of I/O instructions. For a grid of size $M\times N(M\leq N)$, we need $M-1$ cycles to input or output data to the processor cores.

For a grid constituted by P processors, $M\leq\sqrt{P}$, therefore an upper bound on the total I/O overhead is $2(\sqrt{P}-1)$ cycles. Figure 11 gives the throughput of individual μ-core as the grid size increases. The increase in grid size implies a larger I/O time and hence a reduced throughput. Figure 12 gives the variations of throughput as we move from a flat topology $(M\times N,(M\ll N))$ to a square topology $(M\times N,(M\sim N))$. It can be observed that a flat topology has a larger throughput.

In Figure 13, we can observe the variation of μ-core utilization with the number of μ-cores in the grid. The non-I/O cycles refer to the cycles actually utilized for implementation of the AES-128 bit encryption. This remains constant as we vary the grid structure. However, the number of I/O cycles increases as we increase the grid size.

We also observe that the number and ratio of idle cycles for our Multi-core processor increase with the grid size. The total number of cycles in an AES 128 bit encryption is 217 (for AES

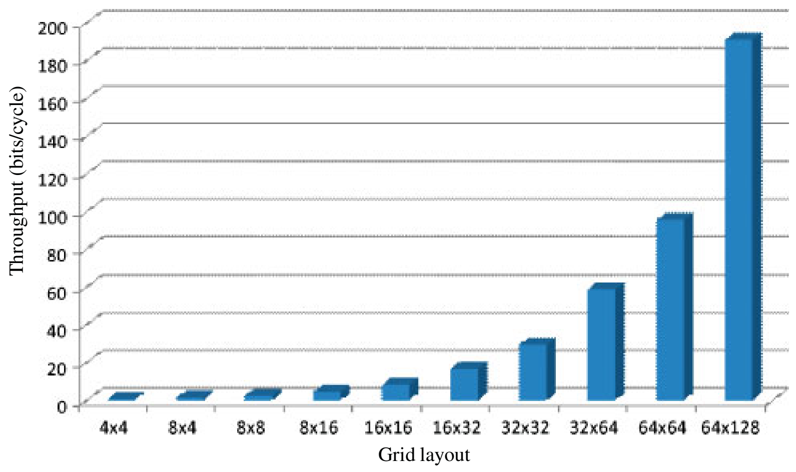

Figure 10. System throughput for AES implementation over various multi-core arrays.

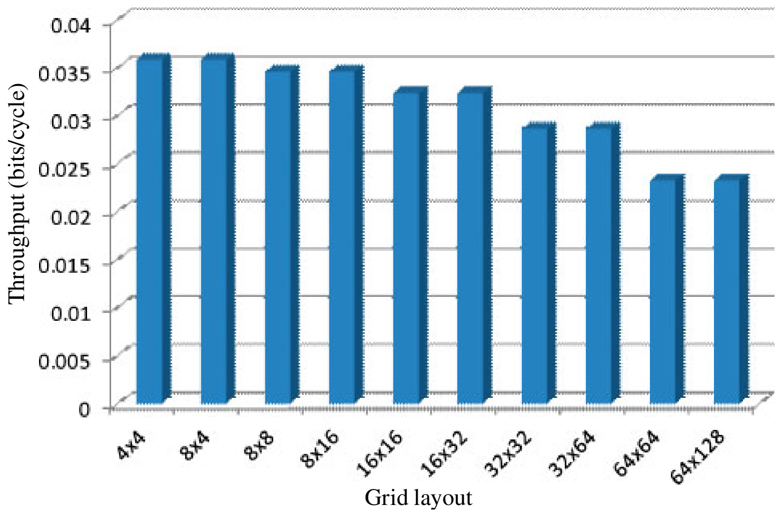

Figure 11. Throughput of individual μ-core.

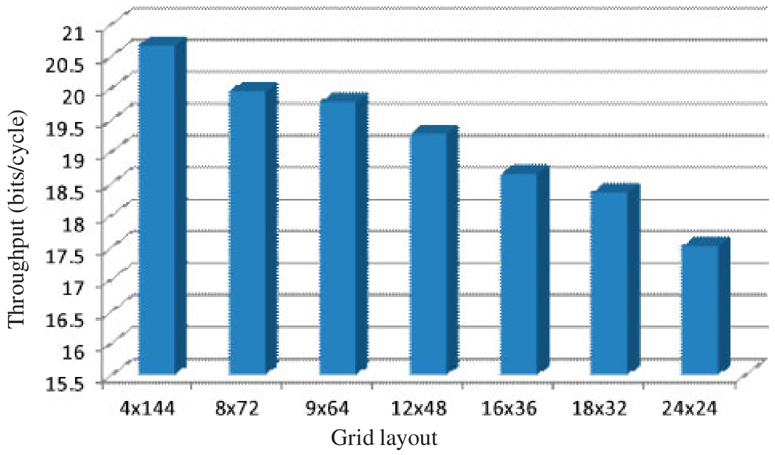

Figure 12. Variation of throughput with variation of grid topology.

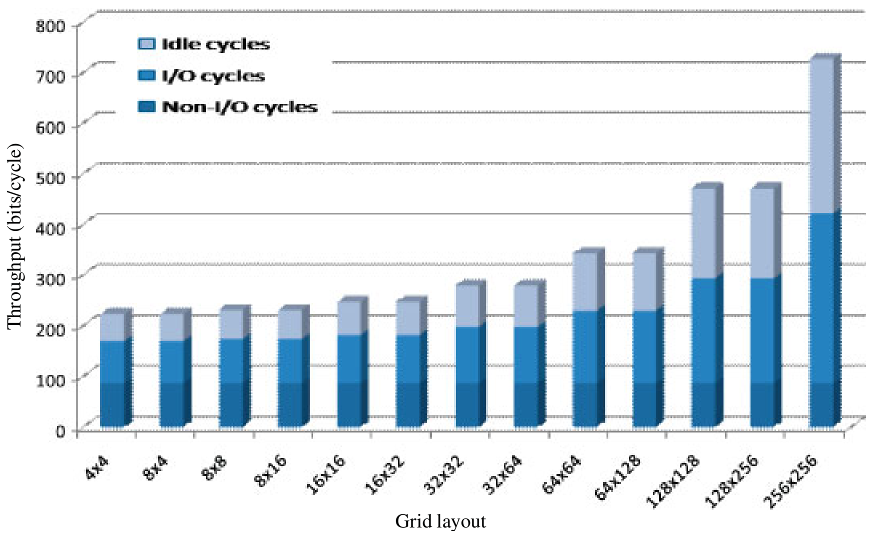

Figure 13. The breakup of μ-core utilization with the increase in grid size.

Table IV. AES encryption time for various processors.

Designer	Processor	No. of cycles
Bertoni *et al.* [4]	ARM7TDMI	2074
Bertoni *et al.* [4]	ARM9TDMI	1755
Bertoni *et al.* [17]	Extended 32-bit RISC ISA	311
Atasu *et al.* [18]	SA-1110	943
	μ-core array	217

Table V. AES throughput comparison with Software and Hardware platforms.

Designer	Processor	Clock frequency	Throughput
Pramstaller *et al.* [19]	Xilinx XCV1000EBG560 FPGA	161 MHz	215 Mbps
Zambreno *et al.* [7]	Xilinx XC2V4000 FPGA	184.16 MHz	23.57 Gbps
	μ-core array	600 MHz	4.17 Gbps
Nadehara *et al.* [20]	embedded processor	1 GHz	640 Mbps
Lipmaa [21]	Pentium4	3.2 GHz	1.538 Gbps

encryption), and $2(M-1)$ (for plaintext input and cipher text output). Out of the 217 cycles for AES encryption, 60% are idle cycles. Of the $2(M-1)$ cycles for I/O operations, there are $N \times M^2$ active processor states and the remaining $2(M-1)MN{-}NM^2{=}NM^2-2MN$ are idle. Thus, as M and N approach large values, the processor utilization drops to about 50%. Half of the cycles would be idle whereas the majority of the remaining 50% of clock cycles would be used for I/O operations and a small percentage of cycles for AES encryption.

It is difficult to make direct comparisons between hardware/ software implementations of any algorithm since the specific hardware target and the design constraints are often different. In μ-core processor design, a chief design constraint is requirement of low power and size. As it is a multi-core processor, we typically expect the throughput to be better than software implementations on single core desktop or embedded processors but less than ASIC- or FPGA-based designs which are customized to serve a specific application. Tables IV and V only give a representative performance of the μ-core architectures to demonstrate this fact. The parallelism exploited in μ-core makes it faster and compact in implementation than other software implementations. Table V compares a

16×16 MMC array calculated at a clock frequency of 600 MHz. The Tilera 64 core processor runs at frequencies of 500–700 MHz while each individual core has I/O interfaces, such as PCIe, and GbE and 64 bit instruction bundle. Large FPGA boards by leading vendors such as Xilinx (Virtex-6 SXT family) can run comfortably at 600 MHz. Our implementation runs with 11 bit Instruction word size and a simple Register bank, hence the assumption of 600 MHz clock frequency is easily justified. It can be seen that 16×16 MMC array can achieve high throughput of around 4 Gbps.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a novel multi-core architecture with simple, light-weight μ-core processors as building blocks. We presented a 2D grid architecture to build large multi-core arrays of these processors to achieve high throughput. Each individual μ-core is small, compact, and specialized for simple cryptographic operations. The ALU of each μ-core is a reconfigurable block capable of being configured for different tasks in various μ-cores depending on the actual application. In the case of AES, we implemented bitwise XOR and AND operations and a 256-byte look-up using this ALU. The proposed efficient mapping of the macroinstructions for grid-level operations into μ-instructions or μ-code and reconfiguration of ALU makes our proposed architecture versatile. We demonstrated the mapping of some macroinstructions into μ-code and also mapped AES-128E encryption algorithm to various grid structures and performed a performance analysis of various grid structures. The results favor the possibility of building large multi-core architectures for efficient cryptography on simple μ-core processors. Some directions for future work are as follows:

- Hardware implementation of μ-core and multi-core architectures. We can optimize the clock speed considering the power and throughput constraints.
- Implementation and analysis of some other interesting cryptographic operations, such as montogomery modular multiplications, can be performed.
- A more generic architecture (including arithmetic operations) for μ-cores can be developed and analyzed for performance in large grid layouts.

REFERENCES

1. Daemen J, Rijmen V. The Block Cipher Rijndael. *CARDIS '98*: *Proceedings of the International Conference on Smart Card Research and Applications*. Springer: London, U.K., 2000; 277–284.
2. FIPS 197. Announcing the Advanced Encryption Standard, 2001. Available from: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.
3. Kuo H, Verbauwhede I, Schaumont P. A 2.29 gbits/sec, 56 mW non-pipelined rijndael AES encryption IC in a 1.8 v, 0.18 μm CMOS technology. *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference*, Orlando, FL, U.S.A., 2002; 147–150. DOI: 10.1109/CICC.2002.1012785.
4. Bertoni G, Breveglieri L, Fragneto P, Macchetti M, Marchesin S. Efficient software implementation of AES on 32-bit platforms. *CHES '02*: *Revised Papers from the Fourth International Workshop on Cryptographic Hardware and Embedded Systems*. Springer: London, U.K., 2003; 159–171.
5. Chodowiec P, Gaj K. Very compact FPGA implementation of the AES algorithm. *Cryptographic Hardware and Embedded Systems Conference*, Cologne, Germany, 2003; 319–333.
6. Morioka S, Satoh A. A 10-gbps full-AES crypto design with a twisted BDD S-Box architecture. *IEEE Transactions on Very Large Scale Integration Systems* 2004; **12**(7):686–691. DOI: 10.1109/TVLSI.2004.830936.
7. Zambreno J, Nguyen D, Choudhary AN. Exploring area/delay tradeoffs in an AES FPGA implementation. Field Programmable Logic and Applications, Leuven, Belgium, 2004; 575–585.
8. http://en.wikipedia.org/wiki/microprogramming [October 2008].
9. Stretch. Available at: http://www.stretchinc.com/ [October 2008].
10. Li GJ, Wah BW. The design of optimal systolic arrays. *IEEE Transactions on Computers* 1985; **34**(1):66–77. DOI: http://dx.doi.org/10.1109/TC.1985.1676516.
11. Kung H, Leiserson CE. Systolic arrays (for VLSI). *Sparse Matrix Proceedings*, Duff IS, Stewart GW. Society for Industrial and Applied Mathematics: Philadelphia, PA; 1978.
12. Fisher JA. The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources. *Ph.D Thesis*, New York, NY, U.S.A. 1979.
13. http://www.mathworks.com/access/helpdesk-r13/help/toolbox/simulink/ug/preface2.html [October 2008].

14. http://www.mathworks.com/applications/dsp-comm/xilinx-ref-guide.pdf [October 2008].
15. Fisher A, Kung H. Synchronizing large VLSI processor arrays. *IEEE Transactions on Computers* 1985; **C-34**(8):734–740. DOI: 10.1109/TC.1985.1676619.
16. Zhou D, Lai TH. An accurate and scalable clock synchronization protocol for IEEE 802.11-based multihop ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems* 2007; **18**(12):1797–1808. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2007.1116.
17. Bertoni G, Breveglieri L, Roberto F, Regazzoni F. Speeding up AES by extending a 32 bit processor instruction set. *International Conference on Application-specific Systems*, *Architectures and Processors* (*ASAP '06*), Steamboat Springs, Colorado, September 2006; 275–282. DOI: 10.1109/ASAP.2006.62.
18. Atasu K, Breveglieri L, Macchetti M. Efficient AES implementations for ARM based platforms. *SAC '04*: *Proceedings of the 2004 ACM Symposium on Applied computing*. ACM: New York, NY, U.S.A., 2004; 841–845. DOI: http://doi.acm.org/10.1145/967900.968073.
19. Pramstaller1 NSD, Mangard S, Wolkerstorfer J. Efficient AES implementations on ASICs and FPGAs. *Lecture Notes in Computer Science*. Springer: Berlin/Heidelberg, 2005; 98–112.
20. Nadehara K, Ikekawa M, Kuroda I. Extended instructions for the AES cryptography and their efficient implementation. *IEEE Workshop on Signal Processing Systems*, *2004* (*SIPS 2004*), Austin, TX, U.S.A., 2004; 152–157. DOI: 10.1109/SIPS.2004.1363041.
21. Lipmaa H. AES implementation speed comparison, 2003. Available from: http://www.tcs.hut.fi/∼aes/rijndael.html.