Efficient Translation of Algorithmic Kernels on Large-Scale Multi-Cores

Amit Pande and Joseph Zambreno
Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011, USA
Email: {*amit, zambreno*}@*iastate.edu*

***Abstract*—In this paper we present the design of a novel embedded processor architecture (which we call a μ-core) that makes use of a reconfigurable ALU. This core serves as the basis of custom 2-dimensional array architectures that can be used to accelerate algorithms such as cryptography and image processing. An efficient translation and mapping of instructions from the multi-core grid to the individual processor cores is proposed and illustrated with an implementation of the AES encryption algorithm on custom-sized grids. A simulation model was developed using Simulink and the performance analysis suggests a positive trend towards the development and utilization of such hardware.**

I. INTRODUCTION

Computational parallelism can be used in principle to accelerate various application domains, including signal processing, communication, and security. Although conventional parallel architectures (including multi-core processors) offer large amounts of concurrency with the aim of improving application throughput, scalability to large numbers of processing elements continues to be a limiting factor. With this motivation, in this paper we present a new multi-core architecture with simple yet robust individual processing processor cores, which we label as a μ-core.

Each individual μ-core has a reconfigurable Arithmetic and Logical Unit (ALU) for simple computational operations, a register bank, a high-speed on-chip memory and some control logic. The reconfigurable ALU allows for a semi-custom mapping of various application-specific tasks into the μ-core. Our proposed multi-core architecture is essentially a two-dimensional grid of individual small processing μ-cores - with each core communicating with the nearest neighbors in four directions (East, West, North, and South) only.

We present a scheme for efficient translation of macro-instructions (grid level instructions) into micro-instructions (simple instructions for the individual processor such as move, copy or Input / Output). Using this technique, we provide an implementation of the AES-128 encryption [3] algorithm and its brief performance analysis. The main contributions of this paper are summarized as follows:

- We present a scheme for microprogramming the 2-D μ-core arrays. An efficient translation of macro-instructions from 2-D arrays into micro-instructions for individual cores is discussed in this paper.
- We demonstrate the translation of the AES-128E algorithm into a 2-D array of μ-cores. Its performance has been reported in terms of core efficiency and overall throughput.
- We introduce the concept of a reconfigurable ALU that can be configured to perform various arithmetic and logical tasks. Reconfiguration can be used to efficiently map the requirements of various image processing, cryptographic (e.g. encryption, hashing) or other algorithms into this ALU.
- Experiments were performed to demonstrate the versatile performance of different 2-D grid arrays. Moreover, the translation explained in this work can be used on any 2-D array.

The paper is organized as follows. In Section 2, we provide an overview of the design and features of our μ-core processor, followed by an introduction to the multi-μ-core (MMC) array concept (built with individual μ-cores). In Section 3, we present a scheme for 'microprogramming' in MMC arrays and illustrate how macro-instructions can be mapped to individual micro-instructions for individual cores. In Section 4, we examine the implementation of the AES algorithm and its building blocks efficiently over custom-sized MMC arrays, followed by an overview of our Simulink-based simulation results in Section 5. Section 6 presents conclusions and a short summary of the work.

II. 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. As compared to traditional processor cores, a μ-core is smaller, with fewer registers, less memory, and a more simplified ALU. In principle, any generic application can be mapped into these cores - we provide a reconfigurable ALU unit which keeps the design simple and can implement a variety of instructions through static or runtime reconfiguration.

A. Overview of μ-core Processor Model

An architectural overview of our μ-core processor design is presented in Figure 1. There are four input and four output ports allowing for easy integration of the individual core into a 2-D array form. The four inputs and the output ports

TABLE I
ISA DESCRIPTION FOR OUR μ-CORE PROCESSOR

Arithmetic Instructions

bit a	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Syntax
0	0	$R_c(3)$	$R_c(2)$	$R_c(1)$	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_c] \ll AND(R[R_b], R[R_a])$
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	$R[R_c] \ll LookUp(R[R_b])$
1	0	$R_c(3)$	$R_c(2)$	$R_c(1)$	$R_b(3)$	$R_b(2)$	$R_b(1)$	0	1	1	$R[R_c] \ll ShiftLeft(R[R_b], 1)$
1	0	$R_c(3)$	$R_c(2)$	$R_c(1)$	$R_b(3)$	$R_b(2)$	$R_b(1)$	1	0	0	$R[R_c] \ll ShiftRight(R[R_b], 1)$
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

bit a	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Syntax
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	$SelectPort \ll OUTPUT(R_b)^*$

* b1 b0 = 00, 01, 10 and 11 selects East, West, North and South Input/Output Ports respectively

Memory Read/Write Instructions

bit a	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Syntax
1	1	1	0	0	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_a] \ll Mem(R[R_b])^\natural$ $\natural$ if $R_b == 7$ then $R[R_b] = R[R_b] - 1$
1	1	0	1	0	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$Mem(R[R_a]) \ll R[R_b]^\aleph$ $\aleph$ if $R_a == 7$ then $R[R_a] = R[R_a] + 1$

Register Transfer Instructions

bit a	bit 9	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Syntax
1	1	1	0	0	$R_b(3)$	$R_b(2)$	$R_b(1)$	$R_a(3)$	$R_a(2)$	$R_a(1)$	$R[R_b] \ll R[R_a]$

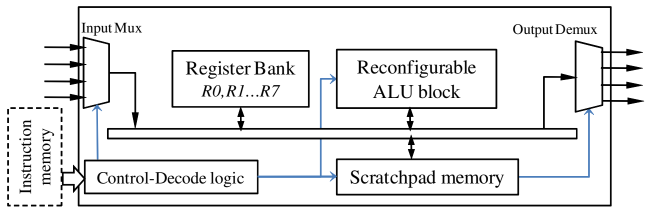


Fig. 1. Architectural overview of a μ-core

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 small scratchpad memory (64 bytes large for our initial investigation), a small reconfigurable ALU, a control word decode mechanism, and a register bank input select multiplexer. The control word is issued by the instruction cache and read by the processor every cycle. The register bank consists of eight registers *R0, R1, R2 ... R7*, each register being 8 bits wide. The input multiplexer selects from one of the input ports as indicated in the control word. The output demultiplexer routes the output into the designated output port with the help of the control logic. Given the simplicity of this design, all of the instructions are able to execute in one cycle.

The reconfigurable ALU provides the flexibility to implement application-specific operations such as table lookup, or traditional logical operations. This can be very convenient when looking at some application domains - for example, in private-key cryptography, many of the operations are performed in Galois field mathematics, and consequently these operations can be translated to simple logical, arithmetic, and lookup table operations. Reconfigurability is achieved through the use of Lookup Table (LUT) based hardware, similar to what is found in conventional FPGAs.

The bits in the Control Word (CW) are directly mapped to their specific core functionality, greatly simplifying the required decoding logic. For our initial experiments, we set the CW to be 11 bits in length. In practice, the length of the CW would be scaled to the number of ALU operations as well as the amount of memory resources. The following subsection explains the Control Word format in more detail.

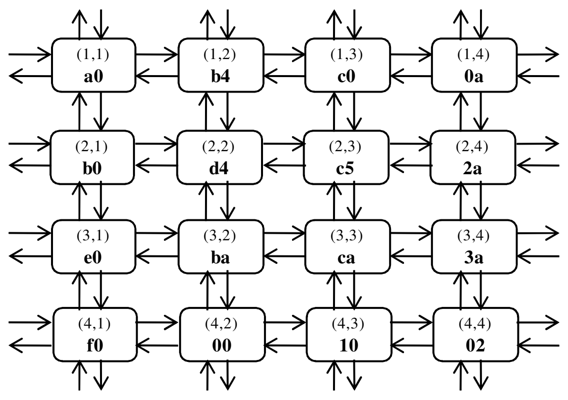
B. Control Word Description

As previously mentioned, the Control Word is 11 bits long (bits are numbered (0 - a) right to left). The overall Instruction Set Architecture (ISA) of our μ-core processor is given in Table I. In the following discussion, we denote the i^{th} bit of the CW as b_i. Thus, b_8 refers to 8^{th} bit (from the right) in the control word. There are eight registers in register bank R and therefore R can be addressed by 3 bits. The R_a, R_b, and R_c fields in the CW are each three bit wide values that address registers in R. For example, when $R_a = 3$, $R[R_a]$ denotes $R3$ of the register bank.

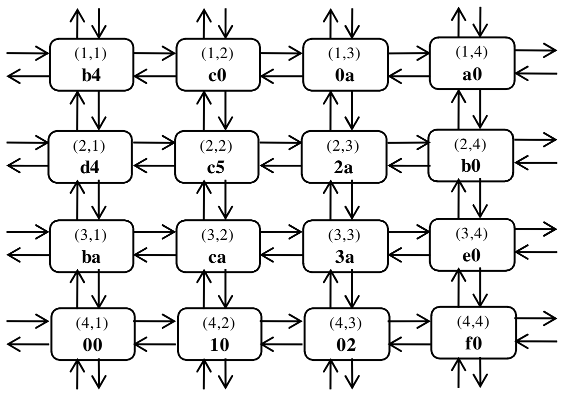
1) Arithmetic Instructions: The value of bits b_a and b_9 (e.g. 00, 01, 10) in the CW refers to the reconfigurable instructions in the ALU. Thus, there is a flexibility in the design to provide custom bitwise operations. This part of the CW is not fixed and can be customized for each μ-core to suit the application requirements.

There are a few custom operations implemented in our initial experiments (an implementation of the AES-128E algorithm). CW $b_a = 0$, $b_9 = 0$ denotes bitwise AND operation while $b_a = 0$, $b_9 = 1$ denotes bitwise XOR operation. The lookup operation ($R[R_c] = LookUp(R[R_b])$) is implemented using an eight bit (256 element) lookup table. There are also functions for left shift, right shift and increment/decrement operations.

2) Input/Output Instructions: The μ-core processor has four input and four output ports corresponding to the East, West,



(a)



(b)

Fig. 2. A left shift operation for MMC (a) initial state and (b) final state

North and South directions respectively. Instruction bits b_1 and b_0 decide the input/output ports with the value (0,0), (0,1), (1,0), and (1,1) indicating East, West, North and South port respectively.

3) Memory Access Instructions: The μ-core processor has a small 64 byte high speed memory. Control words are used for MEMory Read (MEMR) and MEMory Write (MEMW) instructions. If the register $R7$ is selected it is auto-incremented during MEMW and auto-decremented during MEMR to facilitate sequential accesses to memory.

C. The Multi-μ-Core 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 2-D array implementation. In the following section, 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.

III. EFFICIENT MAPPING OF MACRO-INSTRUCTIONS

The ISA for our μ-cores can be looked at as a horizontal microcode [5], where there is a fairly direct correspondence

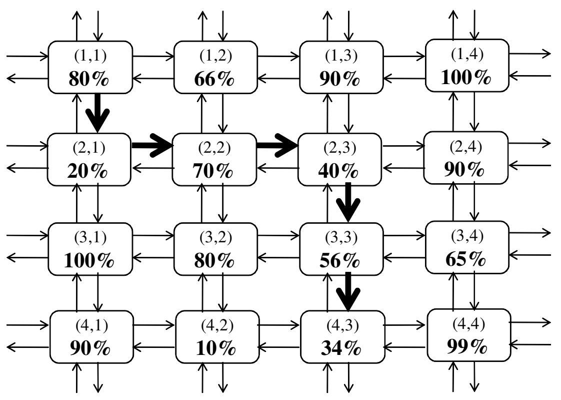


Fig. 3. Routing data from μ-core (1,1) to (4,3)

between the bit fields in a micro-instruction and the control signals sent to the various parts of the CPU, allowing for a simple design for the control-decoding logic. A translation of macro-instructions for the MMC array to micro-instructions for individual cores is required to facilitate mapping of algorithms to the MMC array. An automatic mapping of macro-instructions into micro-instructions is illustrated in the following subsections. We consider some simple MMC commands and their translation for an $M \times N$ MMC array, where for this sample implementation M (the number of rows) = N (the number of columns) = 4.

A. Row/Col Shift Left/Right/Top/Bottom

Consider the macro-instruction *CycleMxN(Left, a, B, i)* for a $M \times N$ 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. For example, *Cycle4x4(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 2. The ID of each processor is listed along with the register $R3$ value which is boldfaced. To map this macro-instruction into micro-instructions, we make the following observations:

1) The operand i needs to be coded in the register address of each micro-instruction.
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 micro-instructions for row 3.
3) To efficiently map the instruction, we note that i left shifts are equal to $(M - i)$ right shifts. Since both left and right operations take equal cycles, we need to choose the smaller of i and $(M - i)$.
4) One left shift (and equivalently 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-numbered μ-cores transfer their value to some temporary registers in even numbered μ-cores (except the first μ-core).

c) All μ-cores (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 on one column at a time.

B. Add GF(2^8)

We consider addition of large words represented in $M \times N$ 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. The Galois field arithmetic used in most cryptographic operations makes it easy to implement each addition as a bitwise XOR between registers. Therefore, the macro-instruction *Add4x4(R2,R4)* can be translated into the logical instruction *XOR(R2,R4)* for each individual μ-core.

C. Routing Data

Consider the macro-instruction *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 macro-instruction into micro-instructions, we proceed as follows:

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

   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 *RouteMxN(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 3 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.

D. Logical Shift for $M \times N$ Bytes Word

Next, we consider *WordShiftMxN(R_a, i)*. This macro-instruction implements a logical shift of i bits for the $M \times N$ byte data stored in the individual R_a registers. We consider the translation of the macro-instruction *WordShift4x4(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,$j-2$).

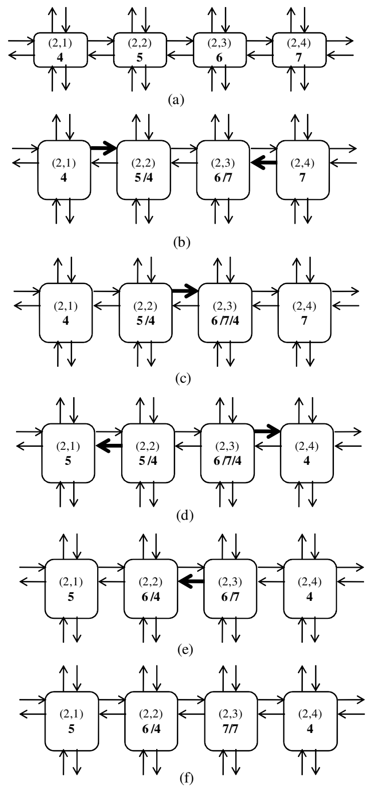


Fig. 4. Illustration of ShiftRow operation for the second row (a) initial state, (b)-(e) intermediate stages, and (f) final state.

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 minimum i position.
4) Shift left the R_a contents and XOR them with the output of 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.

IV. AES Implementation on a 4×4 MMC Processor Core

Many hardware and software-based implementations of cryptographic algorithms such as AES have been reported in the research literature [8], [9]. The authors in [7] discuss multi-core cryptographic support on the Niagara CMT processor. The authors in [6] provide an AES implementation on commodity hardware while the authors in [2] implement Montgomery-multiplication operations on multi-core systems (using MAC operations). One of the main contributions of this work is to illustrate the efficient mapping of AES into our proposed multi-core architecture.

The 4×4 MMC processor has 16 input ports: 4 each to the East, West, North and South. In this section, we demonstrate the scheme mentioned above to implement the AES-128E algorithm. Firstly, 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 bytes of plaintext were loaded in parts into the register $R0$ of each of the 16 μ-cores. The eleven keys from the KeyExpansion stage were loaded into the 64 byte scratchpad memory and subsequently read into register $R2$.

The following subsections explain the mapping of AES macro-instructions into micro-instructions for individual μ-cores. Much of the AES-specific terminology is further detailed in [3].

A. AddRoundKey

The AddRoundKey operation is similar to an *Add GF(2^8)* instruction except that it first involves reading the value from memory to register $R2$ before the subsequent XOR operation. Thus, two micro-instructions are issued to each core for AddRoundKey, except for the first AES iteration in which the key is directly XORed into the register $R0$ content.

B. SubBytes

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

C. ShiftRows

The operation for the second row takes five cycles as illustrated in the Figure 4. The operation for the third row takes six cycles. 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 stored in temporary registers. Thus, each ShiftRows operation requires six cycles.

D. MixColumns

The translation of the MixColumns step into micro-instructions is done in the following manner: For one column, transfer each of the four bytes into all the other three μ-cores. This step requires eight cycles. Thus, the registers $R0$, $R1$, $R2$ and $R3$ of each core has four inputs in the order described in [1], [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. 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 II.

E. Performance Analysis

Next, we performed an analysis to calculate the overhead in implementing AES on larger 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 multiple of 4×4 (e.g. 4×8, 16×16, 32×32). The AES algorithm will still take the same number of computational cycles (217) but the I/O cycles will change. We assume that the input is distributed across the larger dimension (length or breadth) of the array to minimize the number of cycles. The total number of cycles is calculated to be equal to $(4 \cdot min(M, N) - 1)$ cycles.

TABLE II
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**

V. SIMULATION RESULTS

We performed an analysis of the throughput of our multi-μ-core-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. The assumption of clock scalability may be critical for large MMC arrays, however, relevant research has been done in related areas [4], [10] which can be adapted to achieve clock synchronization for large MMC arrays. The implementation was accomplished using the Simulink tool which served as a great aid for efficient implementation and visual representation.

Figure 5 shows 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 an 8×4 configuration to an 8×8 grid layout accompanies an increase in number of I/O instructions. For a grid of size $M \times N$ $(M \leq N)$, we need $M - 1$ cycles to transfer data to or from the processor cores.

For a grid consisting of P processors, $M \leq \sqrt{P}$, therefore an upper bound on the total I/O overhead is $2(\sqrt{P} - 1)$ cycles. Figure 6 gives the throughput of individual μ-cores as the grid size increases. The increase in grid size implies a larger I/O time and hence a reduced throughput. Figure 7 gives the variations of throughput as we move from a flat topology ($M \times N$, $(M << N)$) to a square topology ($M \times N$, $(M \sim N)$). It can be observed that a flat topology has a larger throughput.

In Figure 8, 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 MMC array increase with the grid size. The total number of cycles in an AES 128 bit encryption is 217 (for

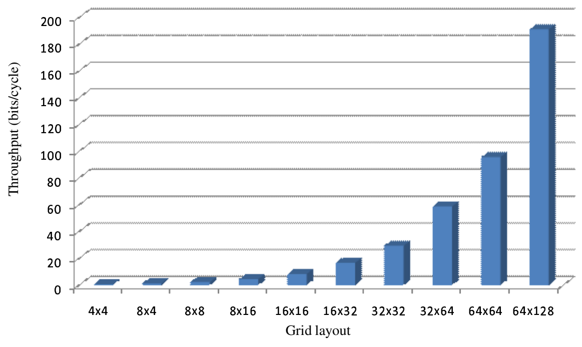


Fig. 5. System throughput for AES implementation over various multicore arrays

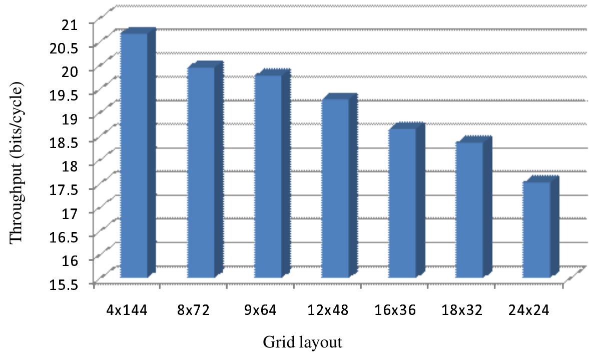


Fig. 7. Variation of throughput with variation of grid topology

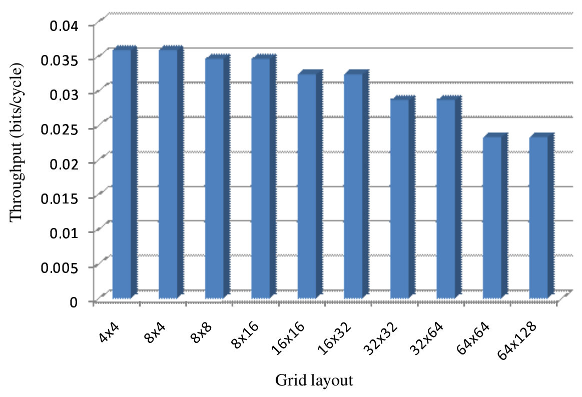


Fig. 6. Throughput of individual μ-core

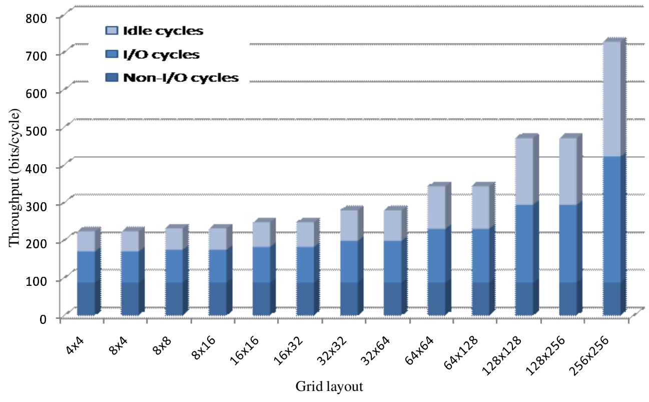


Fig. 8. The break-up of μ-core utilization with the increase in grid size

AES 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 $N \cdot M^2$ 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, while 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.

VI. CONCLUSIONS

In this paper, we introduced a novel multi-core architecture with simple, light-weight μ-core processors as building blocks. The proposed efficient mapping of the macro-instructions for grid-level operations into micro-instructions or μ-code and the reconfiguration feature of the ALU makes our proposed architecture versatile.

We demonstrated the mapping of some macro-instructions into μ-code and also mapped AES-128E encryption algorithm to various grid structures to measure its performance.

REFERENCES

[1] J. Daemen and V. Rijmen. The Block Cipher Rijndael. In *CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*, pages 277–284, London, UK, 2000. Springer-Verlag.

[2] J. Fan, K. Sakiyama, and I. Verbauwhede. Montgomery Modular Multiplication Algorithm on Multi-Core Systems. In *IEEE Workshop on Signal Processing Systems*, pages 261–266, 2007.

[3] FIPS 197. Announcing the Advanced Encryption Standard, Nov 2001.

[4] A. Fisher and H. Kung. Synchronizing large VLSI processor arrays. *IEEE Transactions on Computers*, C-34(8):734–740, Aug. 1985.

[5] J. A. Fisher. *The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources*. PhD thesis, New York University, New York, NY, USA, 1979.

[6] O. Harrison and J. Waldron. Aes encryption implementation and analysis on commodity graphics processing units. In *CHES*, pages 209–226, 2007.

[7] J. Hughes, G. Morton, J. Pechanec, C. Schuba, L. Spracklen, and B. Yenduri. Transparent Multi-core Cryptographic Support on Niagara CMT Processors. In *Proceedings of the Second International Workshop on Multicore Software Engineering (IWMSE09)*, Vancouver, Canada, May 2009.

[8] H. Kuo, I. Verbauwhede, and P. Schaumont. A 2.29 Gbits/sec, 56 mW non-pipelined Rijndael AES encryption IC in a 1.8 V, 0.18 m CMOS technology. In *Proc. IEEE Custom Integrated Circuits Conference*, pages 147–150, 2002.

[9] J. Zambreno, D. Nguyen, and A. N. Choudhary. Exploring area/delay tradeoffs in an AES FPGA implementation. In *FPL*, pages 575–585, 2004.

[10] D. Zhou and T. H. Lai. An accurate and scalable clock synchronization protocol for IEEE 802.11-based multihop ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(12):1797–1808, 2007.