Processor Accelerator for AES

Ruby B. Lee and Yu-Yuan Chen
Department of Electrical Engineering
Princeton University
rblee, yctwo@princeton.edu

***Abstract*—Software AES cipher performance is not fast enough for encryption to be incorporated ubiquitously for all computing needs. Furthermore, fast software implementations of AES that use table lookups are susceptible to software cache-based side channel attacks, leaking the secret encryption key. To bridge the gap between software and hardware AES implementations, several Instruction Set Architecture (ISA) extensions have been proposed to provide speedup for software AES programs, most notably the recent introduction of six AES-specific instructions for Intel microprocessors. However, algorithm-specific instructions are less desirable than general-purpose ones for microprocessors. In this paper, we propose an enhanced parallel table lookup instruction that can achieve the fastest reported software AES encryption and decryption of 1.38 cycles/byte for general-purpose microprocessors, a 1.45X speedup from the fastest prior work reported. Also, security is improved where cache-based side-channel attacks are thwarted, since all table lookups take the same amount of time. Furthermore, the new instructions can also be used to accelerate any functions that can be accelerated through table lookup operations of one or multiple small tables.**

I. Introduction

Cryptography is essential for protecting secret or sensitive information. However, encryption and decryption impose serious performance overhead with current microprocessor architectures. As the performance of microprocessors improve, the performance of software AES encryption and decryption also improves. However, the performance is not good enough for encryption to be incorporated ubiquitously for all computing needs. Furthermore, fast software implementations of AES use tables, and the difference in timing between cache hits and misses in looking up table entries can be used to launch fast and dangerous software cache-based side channel attacks. This can easily leak the secret encryption key to the attacker, who does not even need physical possession of the victim computer.

To bridge the gap between a pure software implementation and a hardware AES implementation, several Instruction Set Architecture (ISA) extensions have been proposed [1], [2], [3] to provide speedup for software AES programs, most notably the 6 AES-specific instructions added to the latest SSE multimedia instruction subset of the Intel microprocessors [2]. Although these proposals improve software AES performance, most of the added functionalities are limited to speeding up AES only; general-purpose instructions are more desirable for inclusion in microprocessors. The contributions of this paper are:

- An analysis of current software AES optimization techniques [4], [5] and identification of their performance and security issues.
- Enhancing a general-purpose parallel table-lookup instruction [1] to further speedup AES.
- Achieving the fastest reported software AES encryption performance of 1.38 cycles/byte for a general-purpose processor.
- Thwarting cache-based side-channel attacks with constant-time table lookups, and
- Achieving increased parallelism and performance without increasing memory overhead.

The remainder of the paper is organized as follows. Section II gives an overview of AES and describes its table transformation optimization. Section III describes software AES optimization techniques, and identifies their performance and security issues. Section IV describes our proposal for enhanced general-purpose instructions that can further accelerate AES and other algorithms. Section V describes AES encryption and decryption using the enhanced new instructions. Section VI compares our solution with other AES implementations and ISA extensions. Section VII is acknowledgment and section VIII concludes our paper.

II. Overview of AES

Rijndael was proposed by Daemen and Rijmen and accepted as the Advanced Encryption Standard in 2001 [6]. It is designed to encrypt 128-bit data blocks, with a key size of 128, 192 or 256 bits. Depending on the block size, it is called AES-128, AES-192 or AES-256. All these AES versions have the same underlying structure, hence we describe only AES-128 below.

The state matrix, M, is initially composed of the input block (called the plaintext P), arranged as a 4x4 matrix of bytes for AES-128. After 10 rounds of transformation, the final state matrix is the ciphertext.

At a high level, the AES implementation can be divided into two parts: Expand the initial key K into eleven 128-bit round keys, and update the state M and `xor` it with the corresponding round key.

The update of the state matrix consists of ten rounds, and in each round the state undergoes four steps:

1) SubBytes: A non-linear byte substitution in which each byte in the state matrix M is substituted by another byte according to a predefined substitution table, the S-box.
2) ShiftRows: The rows of the state matrix M are cyclically shifted over different offsets such that row j is shifted leftward by j byte positions.

3) MixColumns: The state matrix M is multiplied with a predefined $4{\times}4$ matrix C.
4) AddRoundKey: Each byte of the round key is exclusive-or'ed (`xor`) with the corresponding byte of the state matrix.

In [7], the designers of AES suggested a method to implement each round of AES with only table lookups and `xor` operations on a 32-bit processor. Essentially the different steps of the round transformation can be combined into a single set of table lookups, which we describe for AES-128.

The idea is to define a set of tables which combine the computation of the S-box and the MixColumns matrix C. Therefore, we can combine SubBytes and MixColumns and transform a round of AES-128 into three steps:

1) Permute the byte indices of the input state matrix according to ShiftRows.
2) For each column of the state matrix, perform four table lookups using each byte in the column as the index.
3) `xor` all four 32-bit table lookup results and the round key to produce the 4x4 byte output of a round.

Steps 2 and 3 are repeated for all four columns of the state matrix, resulting in sixteen table lookups and sixteen `xor`s.

III. Software AES

Pure software implementations of AES often exploit the table lookup technique described above to increase the performance. A round of AES encryption in OpenSSL [8] consists of 12 shift instructions, 12 mask instructions, 16 `xor` instructions, 16 loads for table lookups and 4 loads for the round keys, totaling 60 instructions for one round. The tenth round needs additional attention: since it does not have a MixColumns step, a separate table is typically used to store the original S-box with 8-bit entries rather than the 32-bit entries in T0, T1, T2 and T3. Alternatively, rather than requiring a fifth table, we can mask out part of the lookup results from the existing tables, to mimic the behavior of looking up only the original S-box table. This requires 16 additional mask instructions. The zeroth round essentially consists of 4 loads for round key and 4 `xor` instructions. Taking into account the 24 additional instructions in the tenth round and the zeroth round, the total instruction count for AES encryption is 624, excluding the loading for input plaintext and storing for output ciphertext. This achieves a theoretical performance of 624/16 = 39 cycles/byte, assuming one cycle per instruction.

In practice, each cache miss incurred by the load instructions will add a few tens of cycles for a Level-1 cache miss, or hundreds of cycles for a Level-2 cache miss, thus increasing the average CPI (cycles per instruction) and hence the average cycles/byte, resulting in decreased performance. Furthermore, due to the cache miss information given by the load instructions when looking up the table entries, software cache-based side-channel attacks [9], [10], [11], [12] are feasible, enabling direct acquisition of the secret key by an attacker, without the need for sophisticated cryptanalysis.

Several instructions for speeding up software AES for various CPU architectures are described in [4]. For example, the x86 architecture allows for scaled indices in load instructions, thereby combining the table load instruction and the shift instruction. Other code optimization techniques, including a combined `load` and `xor` instruction and a combined `shift` and `mask` instruction, are extensively explored in [4]. Note the baseline mode of operation in [4] is counter mode, AES-CTR; therefore no decryption of AES is required or considered. Further, the authors assume that key generation is not in the critical path, since the same key is re-used for multiple blocks of counter encryption.

We make the following observations about the software solutions:

- Architectural differences between processors have a significant impact on the optimized performance.
- Optimization for both performance and security (from side channel attacks) is hard to achieve simultaneously.
- No general architectural support currently exists for speeding up software AES. Code optimized for one architecture may not be optimal for another architecture.
- Word-level parallelism is not fully exploited. The operations done for each 32-bit word are the same and can be done in parallel. They are sequentially executed in current architectures, leading to sub-optimal performance.

To solve the above issues, we suggest that a general-purpose instruction can be added to any microprocessor's ISA (Instruction Set Architecture), which can significantly accelerate software AES while defeating cache-based side-channel attacks. The instruction also enables the exploitation of word-level parallelism.

IV. Proposed enhanced-ISA Software Solution

In [1], a Parallel Table Lookup (PTLU) module is proposed to speed up block ciphers. Fiskiran and Lee describe an implementation on 64-bit processors, where AES-128 encryption takes 10 cycles per round, and a total of 126 cycles for the 10 rounds. The tenth round required many extra instructions in their implementation. Our solution is inspired by their design, but we significantly enhance the performance achieving 2 cycles per round, and a total of 22 cycles for all 10 rounds. We improve the performance significantly with a new general-purpose enhancement to the PTLU module that reduces the 10th round to only 3 cycles. While [1] alludes to performance numbers of 32 cycles for a 128-bit PTLU module, they do not describe how this is achieved. Even compared to this, our achieved performance of 22 cycles is 1.45X faster.

We propose adding an instruction to a microprocessor: an enhanced Parallel_Read (`Pread`) instruction. The instruction is general-purpose, not AES-specific and may be useful for other algorithms. From the point of view of crypto algorithms, it can be considered a generalized S-box. The Parallel_Read instruction implements a software-managed fast memory where up to 16 parallel reads of up to 8 different tables is enabled. These reads all have constant access times, since the module implementing the `Pread` instruction is designed to look like a functional unit with fixed latency, rather than cache memory with two different access times – depending on whether the data is in the cache (a cache hit) or not (a cache miss).

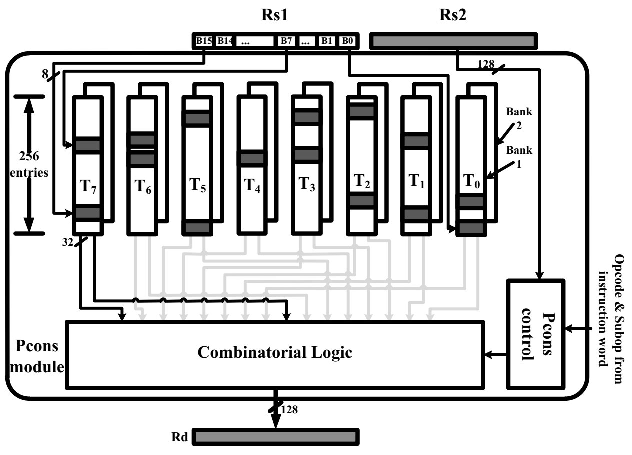

Fig. 1. Our *Pcons* unit with two banks and two read-ports per table.

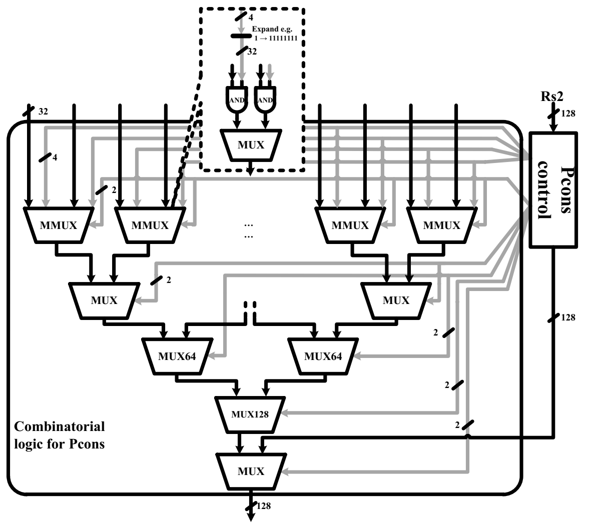

Fig. 2. The internals of the Combinatorial Logic block for the *Pcons* unit.

A. Parallel_Read instruction

We propose a *Pcons* (Parallel constant) unit, which is like an enhanced PTLU module [1], where small tables of constants can be read in parallel in constant time. This is equivalent to always hitting in the traditional hardware-managed cache. This can increase performance and mitigate side-channel attacks at the same time. Figure 1 shows the default version of our proposed *Pcons* unit. A single Parallel_Read (`Pread`) instruction can read up to 8 different tables in parallel, each with up to 256 entries. Each table thus requires one byte of index to select one of the 256 entries. We allow each table to have two separate read ports; hence one 128-bit register can be used to index (and read) 16 pieces of data out of 8 separate tables in parallel. Since modern microprocessors have 64-bit registers in their basic integer datapaths, and 128-bit registers in their multimedia datapaths, it is reasonable to implement our `Pread` instruction using the 128-bit multimedia registers, i.e., with SSE instructions in Intel x86 processors [13] or Altivec [14] instructions in the IBM Power processors. The first read-ports of the 8 tables are addressed by the rightmost 8 bytes of a source register for the `Pread` instruction, while the second read-ports of the 8 tables are addressed by the next (leftmost) 8 bytes. This is one of our enhancements over [1], where we double the parallelism achievable with a `Pread` instruction, without doubling the storage overhead for parallel tables.

Another enhancement we propose over [1] is the provision of multiple banks to allow concurrent encryption and decryption or concurrent execution of different algorithms using different sets of tables. In [1], context switching incurred significant performance overhead for reloading the contents of the different sets of tables. In our solution, each `Pread` instruction specifies which bank of 8 tables it is referring to. The storage required for one bank of 8 tables, with 256 entries/table and 4 bytes/entry, is 8 Kbytes. Hence, the storage requirement for 2 banks (16 Kbytes) is only a small fraction of the 64 or 128 Kbyte size of typical Level-1 data caches, while providing much higher performance per storage byte than a typical cache.

Since we want this *Pcons* unit to look more like a functional unit than a cache memory, it should conform with the processor's datapaths between its multimedia register-set and the multimedia functional units. This implies that each `Pread` instruction can have two source register operands (three for Altivec [14]) and one register result, each register being 128 bits. Since our `Pread` instruction reads 16 pieces of data each 4 bytes long, we must condense the 64 bytes of data into a register width of 128 bits (16 bytes). This is done by the Combinatorial Logic block shown in Figure 1. For general-purpose use of this software-managed fast memory, we define a few generic ways to combine the data (up to 16 pieces) read in parallel from the 8 tables: `xor` the results together, `or` them together, select the entry from one of the 8 tables, or concatenate the 4-byte results into 8-byte or 16-byte results. This can be achieved by a tree of multiplexor blocks, as shown in Figure 2. Detailed decoding of the controls for this block, from the `op` field defined below, are given in Table I.

A key novel aspect of our `Pread` instruction, compared with [1], is the new masking ability that can optionally be applied to the outputs read from the parallel tables, as illustrated by the dotted-box call-out in Fig. 4 and row 2 of Table I. The table outputs are first masked by bits from the second operand. Each bit of the second operand is expanded into 8 bits, to mask a byte of the 4-byte output of a table. Hence, 4 mask bits are sufficient for a 4-byte table output, and a total of only $4 \times 16 = 64$ bits are needed for 16 table outputs. This is a useful, general-purpose masking of table outputs. It is also the core novel idea that allows the last round of AES encryption to be done as efficiently as the other 9 rounds (as explained in section 6 and Figure 6).

Putting all this together, our enhanced new instruction, Pread, is specified as follows with several variants specified by the `op` encodings:

```
Pread.m.op.b    Rd,  Rs1,  Rs2
```

When `m` is specified as in `Pread.m`, the table outputs are

TABLE I
CONTROL SIGNAL DEFINITION OF THE MUX TREE IN THE COMBINATORIAL LOGIC. AT EACH LEVEL OF THE MUXES, ONLY TWO CONTROL SIGNALS, (C0, C1), ARE NEEDED. NOTE THAT THE FIRST LEVEL OF MUXES, THE MMUXES, IS THE ONLY LEVEL AFFECTED BY THE MASKING. THE MUX64 AND MUX128 JUST CONCATENATE MULTIPLE 32-BIT RESULTS. THE LAST MUX LEVEL CAN BE USED TO `xor` THE SECOND OPERAND, RS2.

		(C0, C1) Value			
		(0, 0)	(0, 1)	(1, 0)	(1, 1)
MMUX	`Pread.*`	L	L `xor` R	L `or` R	R
	`Pread.m.*`	L & M1	(L & M1) `xor` (R & M0)	(L & M1) `or` (R & M0)	R & M1
MUX		L	L `xor` R	L `or` R	R
MUX64		L `||` R	0 `||` L `xor` R	0 `||` L `or` R	0 `||` R
MUX128		L `||` R	0 `||` L `xor` R	0 `||` L `or` R	0 `||` R

first masked by bits from the second operand, `Rs2`. If `m` is not specified, as in a typical `Pread` instruction, `Rs2` is `xor`'ed with the result at the last stage of the MUX-tree.

The `op` field (5 bits) currently has $3+16$ encodings: `xor`, `or`, Concatenate, or Select one of 16 outputs read from the tables. The `b` field (2 bits) allows up to 4 banks of 8 tables each. Total encoding of these 3 subop fields can be done in only 8 bits.

For AES, each of the MUX blocks in Figure 2 reduces to a simple `xor` operation. However, a more general-purpose implementation of these MUX blocks makes it more useful for other algorithms, while incurring only a small amount of additional hardware overhead in access time and area.

The initialization of the table contents can be achieved by a *parallel table initialize* instruction, similar to that in [1], which writes across all 8 tables in a bank, taking at most 256 cycles. The overhead is typically insignificant, since it can be amortized over the encryption/decryption of many blocks.

B. Implementation Cost

We use CACTI 5.3 [15] to estimate the on-chip storage overhead of our *Pcons* unit and compare it with the on-chip level 1 cache of different sizes, both direct-mapped caches (DM) and 2-way set-associative caches. Table II shows the CACTI simulation results. Our default *Pcons* unit with 2 banks of 8 tables has a size of 16 Kbytes. When compared to a cache with the same size (16KB) with 2-way set-associativity, *Pcons*'s access time is on average 191% faster and its area is on average 55% smaller. *Pcons*'s access time is considerably smaller due to its small number of entries (256) and small line size (4 bytes). For all three storage sizes, the area of a *Pcons* unit is smaller than the equivalent-sized cache in all cases, except for the Direct-mapped 32Kbyte cache, where a 4-bank *Pcons* unit is 10% larger. Note also that for each storage size shown, the *Pcons* unit can deliver 16 pieces of data per `Pread` instruction, rather than only 1 piece of data for the DM or 2-way set-associative cache on a regular `Load` instruction.

V. AES-128 ENCRYPTION AND DECRYPTION

Using the *Pcons* unit, an AES-128 block encryption can be done in just 22 cycles in software, achieving the performance of 1.38 cycles/byte[1]. Figure 3 shows that each round for the first 9 rounds, takes just 2 instructions (cycles) each, using `byte_perm` [1] followed by a `Pread`. The `byte_perm` instruction permutes the order of bytes within a register according to the index value supplied by the second source register. The `xor` of the round key is also done by the `Pread` instruction, using the second operand `Rs2`, to supply the round key. This is done by the last MUX block in Figure 2.

The last round of AES-128 encryption needs special attention since it does not have the MixColumns step. In other words, the lookup table needed for the last round is the original S-box instead of the transformed tables. In order to extract from the transformed tables the original S-box table, the `Pread.m` instruction is used to mask out the bits not needed for the last round. Hence, the last round takes only 3 instructions: `byte_perm`, `Pread.m` and `xor`. The explicit `xor` instruction is needed to add the round key for the last round, since the second operand in the `Pread.m` instruction is used for the mask. In the `Pread` instruction in the other 9 rounds, the second operand supplies the round key that can be `xor`'ed in the Combinatorial Logic part of the `Pread` instruction. By comparison, without the `Pread.m` instruction variant, the last round of AES-128 will need at least 13 instructions [1].

We employ the same table transformation for AES-128 decryption, except that the table values are different from the values used for encryption. Therefore, for simultaneous encryption and decryption without the overhead of re-loading table values, we take advantage of the bank design of the *Pcons* unit. One bank of tables is pre-loaded with the table values for encryption and the other for decryption. Note that a third bank of tables is needed for decryption.The masking technique used in the last round of encryption *cannot* be employed in the last round of decryption, due to the non-unity values being multiplied in the inverse MixColumns (InvMixColumns) step [6]. This additional bank of tables holds the inverse S-box table values. Hence, we do not have to rearrange the byte indices in a different manner for the last round of decryption as we do in encryption. The upside is that only 21 cycles are thus needed for AES decryption. Note also that in counter-mode AES, no decryption is necessary, since

[1]Note that the performance excludes the reading of the plaintext and writing of the ciphertext and a CPI of 1 is assumed.

TABLE II
COMPARISON OF ACCESS TIME AND AREA FOR *Pcons* STORAGE VERSUS CACHES OF THE SAME SIZE. THE AREA DOES NOT INCLUDE THE COMBINATORIAL LOGIC FOR THE *Pcons* UNIT. ALL CACHES HAVE 64-BYTE LINE SIZE.

	8KB			16KB			32KB		
	Pcons 1 bank	DM cache	2-way cache	*Pcons* 2 bank	DM cache	2-way cache	*Pcons* 4 bank	DM cache	2-way cache
Access time (ns)	0.47	0.57	0.93	0.49	0.67	0.95	0.53	0.82	1.00
Area (mm^2)	0.22	0.45	0.82	0.24	0.50	0.84	0.67	0.61	0.89

```
xor            R1, R1, R10 # xor Round 0 key
byte_perm      R1, R1, R2
    # Round 1: R2 contains indices
    # (15,10,5,0,11,6,1,12,7,2,13,8,3,14,9,4)
Pread.x, b0    R1, R1, R11
    # Round 1: b0 denotes the first bank
    # op=x denotes xor in MUX blocks
byte_perm      R1, R1, R2  # Round 2
Pread.x, b0    R1, R1, R12
byte_perm      R1, R1, R2  # Round 3
Pread.x, b0    R1, R1, R13
byte_perm      R1, R1, R2  # Round 4
Pread.x, b0    R1, R1, R14
byte_perm      R1, R1, R2  # Round 5
Pread.x, b0    R1, R1, R15
byte_perm      R1, R1, R2  # Round 6
Pread.x, b0    R1, R1, R16
byte_perm      R1, R1, R2  # Round 7
Pread.x, b0    R1, R1, R17
byte_perm      R1, R1, R2  # Round 8
Pread.x, b0    R1, R1, R18
byte_perm      R1, R1, R2  # Round 9
Pread.x, b0    R1, R1, R19
byte_perm      R1, R1, R3
    # Round 10: R3 contains byte indices
    # (5,0,15,10,1,12,11,6,13,8,7,2,9,4,3,14)
Pread.m.x, b0  R1, R1, R4
    # Round 10: R4 contains masking bytes:
    # 0x0000 0000 0000 0000 2184 2184 2184 2184
xor            R1, R1, R20
    # Round 10: Since Rs2 used for mask,
    # XOR with key must be done separately
```

Fig. 3. 10 rounds of AES in 22 cycles with *Pcons* unit. R1 contains 128-bit plaintext. R2 and R3 contain the byte indices of permutation for the first 9 rounds and the tenth round, respectively. R4 contains the mask for the tenth round. R10-R20 contain the round keys.

only the counter has to be encrypted for both encryption and decryption. Hence, the extra bank for storing AES decryption tables for the last round is not required for counter-mode AES.

VI. COMPARISON WITH INTEL AND OTHER WORK

A specially hand-optimized assembly code for IA64 [5] of AES-128 is reported to achieve (18 + 13*10) = 148 cycles per block, namely 9.25 cycles/byte. However, the performance drops to 11 cycles/byte if mitigations for cache-based side-channel attacks are employed.

Käsper and Schwabe [16] presented a faster bit-sliced AES on Intel Nehalem platform, achieving 6.92 cycles/byte. Their method improves upon previous bit-sliced implementations by processing eight 16-byte AES blocks in parallel, with constant time operations. However, their implementation is limited to counter mode and is not applicable to modes with feedback, e.g., CBC or OFB mode.

Intel officially added hardware support for AES in their 2010 generation CPUs [2] by adding new instructions for speeding up AES. Six SSE instructions are added specifically for AES, including 4 instructions for AES encryption and decryption, namely `AESENC`, `AESENCLAST`, `AESDEC` and `AESDECLAST`, where `AESENC` and `AESDEC` facilitate the first nine rounds of encryption and decryption and `AESENCLAST` and `AESDECLAST` facilitate the tenth round of encryption and decryption, respectively. The other two instructions, `AESIMC` and `AESKEYGENASSIST` are designed to support AES key expansion. If we assume that round keys are generated with `AESIMC` or `AESKEYGENASSIST` and stored in appropriate registers, encrypting or decrypting one block of data (16 bytes) takes a total of 61 cycles: `pxor` takes 1 cycle and each of the AES instructions has a latency of 6 cycles [2].

Tillich et al. [3] proposed AES-specific instructions which support SubBytes and MixColumns operations, namely `sbox`, `mixcol` and their variants, for both encryption and decryption The ShiftRows operation is implicitly included in those instructions. These instructions can also be used for key expansion. While more ”general-purpose” than the AES round-specific instructions introduced by Intel, they are still algorithm-specific instructions – unlike our general-purpose `Pread` instruction, which can be used to accelerate any algorithm with many constants that can be read in parallel.

CRYPTONITE [17] is a crypto processor that supports various encryption and hashing standards, e.g. AES, DES, MD5 and SHA1. The architecture has a two-cluster design resembling a Very Long Instruction Word (VLIW) architecture, enabling side-by-side AES round encryption and key expansion for embedded systems. It also has a vector memory unit which can look up eight 8-bit tables in parallel, similar to but less general than our *Pcons* unit, and generic fold, rotate and interleave instructions for speeding up AES operations. Compared to the CRYPTONITE processor, which is a special purpose cryptoprocessor with complex vector memory and VLIW execution, our method just adds 2 instructions (`Pread` with its variants, and `byte_perm` [1]) to *any* processor.

Table III summarizes the architectural differences and compares the performance of the above ISA extensions with our proposal. The numbers are reported or calculated from the respective references.

Conservatively assuming a microprocessor frequency of

TABLE III
COMPARISON OF ISA EXTENSIONS FOR AES SUPPORT. (FOR [1], THE 7.87 IS WHAT IS DESCRIBED FOR 64-BIT PROCESSORS, WHILE THE 2 CYCLES/BYTE IS WHAT WE INFER IF THEY USE 128-BIT PROCESSORS.)

		Performance (cycles/byte)	
	Purpose	Encryption	Decryption
Intel [2]	AES-specific	3.81	3.81
Tillich [3]	AES-specific	3.19	3.19
Oliva [17]	General-purpose	4.38	5.06
Fiskiran [1]	General-purpose	7.87 (2)	7.87 (2)
This work	General-purpose	1.38	1.38

1GHz, the maximum throughput of our proposal can achieve $1/1.38 \times 8 = 5.8$ Gbps. The performance is comparable to a mode-independent ASIC implementation [18]. If we consider a faster 3 GHz processor, then it's likely that a `Pread` instruction will take two cycles (one for table lookup and one for combinatorial logic), which translates to a total of 32 cycles per AES block, thus achieving a throughput of 12 Gbps, comparable to that of highly-pipelined and loop-unrolled FPGA implementations, which ranges from 11.77 Gbps [19] to 21.56 Gbps [20]

Recent work have also proposed promising results using GPU for accelerating software AES speeds [21], [22]. The implementations utilize highly parallel processing units available in the GPU to achieve a throughput rate of more than 10 Gbps. However, the high throughput depends on the availability of independent data. Hence, both these GPU implementations and the highly-pipelined implementations described above are more suitable for encrypting independent blocks, as in counter mode for AES, and thus cannot be directly compared with our mode-independent method without caution.

VII. ACKNOWLEDGMENTS

We thank Michael Wang for implementing an initial version of the AES algorithm using the method described in this and our earlier papers on the PAX cryptoprocessor, developed by our lab.

VIII. CONCLUSION

We analyzed optimized software AES implementations and observed their inability to fully exploit available parallelism and their vulnerability to cache-based side-channel attacks. Both these issues can be overcome with our enhanced new general-purpose instructions `Pread`, which can be added to any processor ISA. Our `Pread` instruction improved upon the solution proposed in [1] by introducing parallel reads within a table, not just across tables, to increase the performance without increasing the storage needed for tables. We also introduced a new *parallel read masked* instruction variant, that significantly reduces the cycles need for the last round of AES encryption while providing a general-purpose operation to select bytes read from a table entry. Also, we added the concept of banks of tables to reduce context switching time. With these improvements, we can achieve the fastest reported software AES performance of 1.38 cycles/byte, for encrypting one block of 128-bit data. This is a 1.45 X improvement over the previous best time reported, but not discussed, in [1]. We achieve full word-level parallelism within a round, and also mitigate cache-based side-channel attacks since each `Pread` instruction takes constant time. By clearly defining the operations needed and how software AES can be accelerated with these operations, we hope to encourage the adoption of these general-purpose instructions in processors and the development of algorithms that exploit the performance potential of such instructions.

REFERENCES

[1] A. M. Fiskiran and R. B. Lee, "On-Chip Lookup Tables for Fast Symmetric-Key Encryption," in *Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors*, 2005, pp. 356–363.

[2] S. Gueron, "White Paper: Advanced Encryption Standard (AES) Instruction Set," July 2008, Intel Mobility Group, Israel Development Center.

[3] S. Tillich and J. Großschädl, "Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors," in *Cryptographic Hardware and Embedded Systems – CHES*, 2006, pp. 270–284.

[4] D. J. Bernstein and P. Schwabe, "New AES Software Speed Records," Cryptology ePrint Archive, Report 2008/381, 2008.

[5] A. Polyakov, "IA-64 ISA artwork version 1.2," http://www.openssl.org/.

[6] NIST, "FIPS-197: Advanced Encryption Standard," November 2001.

[7] J. Daemen and V. Rijmen, "The Design of Rijndael," 2002.

[8] The OpenSSL Project, http://www.openssl.org/.

[9] C. Percival, "Cache Missing for Fun and Profit," 2005, http://www.daemonology.net/hyperthreading-considered-harmful/.

[10] O. Aciiçmez, W. Schindler, and Çetin K. Koç, "Cache Based Remote Timing Attack on the AES," in *CT-RSA, The Cryptographers Track at the RSA Conference*, 2007, pp. 271–286.

[11] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks Against AES," in *Cryptographic Hardware and Embedded Systems – CHES*, 2006, pp. 201–215.

[12] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *Proceedings of the 34th annual International Symposium on Computer Architecture*, 2007, pp. 494–505.

[13] "Intel 64 and IA-32 Architectures Software Developers Manual Volume 2b: Instruction Set Reference," 2006.

[14] IBM, "Power ISA Version 2.04," April 2007, http://www.power.org/resources/downloads/PowerISA_203.Public.pdf.

[15] H. Labs, "CACTI 5.3," http://quid.hpl.hp.com:9081/cacti/index.y?new.

[16] E. Käsper and P. Schwabe, "Faster and Timing-Attack Resistant AES-GCM," in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, 2009, pp. 1–17.

[17] D. Oliva, R. Buchty, and N. Heintze, "AES and the Cryptonite Crypto Processor," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, 2003, pp. 198–209.

[18] A. Hodjat, D. D. Hwang, B. Lai, K. Tiri, and I. Verbauwhede, "A 3.84 gbits/s AES crypto coprocessor with modes of operation in a 0.18-μm CMOS technology," in *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, 2005, pp. 60–63.

[19] F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs," in *Cryptographic Hardware and Embedded Systems – CHES*, 2003, pp. 334–350.

[20] X. Zhang and K. Parhi, "An efficient 21.56 Gbps AES implementation on FPGA," in *Signals, Systems and Computers. Conference Record of the Thirty-Eighth Asilomar Conference on*, Nov. 2004, pp. 465–470.

[21] O. Harrison and J. Waldron, "Practical symmetric key cryptography on modern graphics hardware," in *Proceedings of the 17th conference on Security symposium*, 2008, pp. 195–209.

[22] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi, "Design of a parallel AES for graphics hardware using the CUDA framework," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, 2009, pp. 1–8.