Efficient Permutation Instructions for Fast Software Cryptography

Performing permutations in software can facilitate more widespread use of secure information processing and faster multimedia processing. But current instruction set architectures, even when augmented with subword-parallel multimedia instructions, do not provide efficient, bit-level software permutations. Four new instructions each offer a solution.

Ruby B. Lee
Zhijie Shi
Xiao Yang
Princeton University

Pervasive secure information processing over the public wired and wireless Internet could benefit from rapid and convenient cryptographic transformations. But the performance of software-implemented cryptographic functions is hampered by certain operations that have not been optimized in a processor's instruction set architecture because they occurred infrequently in earlier programming workloads. One such operation is the permutation of bits within a block to be encrypted, which is particularly difficult in word-oriented processors. Permutations play a prominent role in the secure processing of information and in multimedia processing.

Secure information processing

Secure information processing includes authentication of users and host machines, confidentiality of messages sent over public networks, and assurances that messages, programs, and data have not changed in transit. It also includes access control and provisions to ensure privacy, anonymity, and the availability of essential services.

Systems can provide some security functions by using security protocols that employ different cryptographic algorithms, such as those based on public keys, symmetric keys, and hashing. To facilitate pervasive and secure information processing, such security functions must be fast and painless to implement. One way to achieve this goal is to perform the cryptographic functions rapidly in software, rather than requiring users to buy special-purpose hardware cryptography boards. Here, we assume that a programmable processor is available in an information appliance or server machine to perform regular (unsecure) transactions over the network. The question becomes "What general-purpose operations should this programmable processor incorporate so that it can execute cryptographic functions without significant performance degradation?"

Our work focuses on one set of operations, permutations, which are useful in symmetric-key cryptographic algorithms for encrypting large amounts of data for confidentiality. Sym-

0272-1732/01/$10.00 © 2001 IEEE

metric-key algorithms break a message into blocks and use confusion and diffusion operations synergistically to transform each block of plaintext (original message) into ciphertext (encrypted message) bits.[1] Confusion obscures the relationship between the plaintext and ciphertext by, for example, substituting arbitrary bits for plaintext bits. Diffusion spreads the redundancy of the plaintext over the ciphertext—for example, through permutation of the bits of the plaintext block.

DES (Data Encryption Standard) and triple DES[2] use bit-level permutations extensively to encrypt a block of 64 plaintext bits into ciphertext. But bit-level permutations are slow when implemented with the current instructions available in microprocessors and other programmable processors. Because permutations are particularly difficult for existing processors, new cryptographic algorithms, such as the Advanced Encryption Standard,[3] tend to avoid permutations to ensure fast software implementations.

In contrast, we show how programmable processors can achieve efficient bit permutations, thus enabling future cryptographic algorithms to use the superior diffusion capabilities of permutations.

Quick multimedia processing

Permutations are also an important new requirement for fast processing of digital multimedia information with subword-parallel instructions,[4,5] more commonly known as multimedia instructions. Every major microprocessor instruction set architecture (ISA) now has these subword-parallel instructions for fast multimedia information processing.[4-11] With subwords packed into 64- or 128-bit words, it is often necessary to rearrange the subwords within the word. However, many of these multimedia ISA extensions do not provide such subword permutation instructions, except for a few initial attempts in MAX-2,[5] IA-64,[8] and AltiVec.[9]

New permutation instructions

To serve the growing need for efficient software permutation, we developed four permutation instructions and an underlying methodology for efficiently performing arbitrary n-bit permutations in programmable processors. Although our work targets the more difficult problem of permuting n 1-bit elements, it also addresses the issue of permuting fewer multibit subwords packed into an n-bit word, a feature needed to accelerate multimedia processing in software.

By providing the ability to do fast permutations in software, we open the field for new cryptography and multimedia algorithms using these powerful yet simple permutation primitives. This capability results in much faster cryptography and multimedia processing, while retaining the flexibility of software implementations, for secure multimedia information appliances and servers.

Past work

Because current microprocessors are word oriented, performing bit-level permutations is painful. Every bit must be extracted from the source register, moved to its new location in the destination register, and combined with the bits that have already been moved. This process can take four instructions per bit (mask generation, AND, SHIFT, and OR). In certain microprocessors such as PA-RISC,[12] more-powerful bit-manipulation instructions—such as EXTRACT and DEPOSIT—can essentially move a bit with two instructions. As a result, any arbitrary permutation of n bits requires $4n$ or $2n$ instructions. Processors can perform certain predefined permutations with regular patterns in fewer instructions. But, in general, an arbitrary 64-bit permutation could take up to 128 or 256 instructions on current microprocessors.

Because this is unacceptably slow, programmers have used table lookup methods to implement fixed permutations. Achieving a fixed permutation of n input bits is possible with one or m table lookups. Doing so requires a table with 2^n entries or m tables with $2^{n/m}$ entries; each entry contains n bits.

For example, implementing a 64-bit permutation can be done with eight tables, each with 256, 64-bit entries. Each table represents the permutation of eight consecutive bits of the source. Each entry would have 0s in all positions, except the eight bit positions in the result to which the table lookup permutes the selected 8 bits in the source. Performing a 64-bit permutation would take 23 instructions: eight to extract LOAD indices, eight LOAD instructions for table lookups, and seven OR

instructions to combine table lookup results. The memory requirement is 16 Kbytes for eight tables. Although 23 instructions sounds considerably better than the previous methods' 128 or 256 instructions, the actual execution time could be much longer because of cache miss penalties of around 50 cycles per cache miss, on the eight LOAD instructions. Hence, the time taken to do an arbitrary n-bit permutation in software is $O(n)$ cycles.

PA-RISC's MAX-2 multimedia instruction set was the first to define a general-purpose PERMUTE instruction. PERMUTE can perform any permutation, with and without repetitions, of the four, 16-bit subwords packed in a register.[5] IA-64 has the MUX instruction, which is just like PERMUTE for 16-bit subwords except that it also has five new permute variants for 8-bit subwords. These byte permute options are reverse, mix, shuffle, alternate, and broadcast.[8] AltiVec has the VPERM instruction, which extends the permutation capabilities of MAX-2's PERMUTE instruction to 8-bit subwords selected from two 128-bit source registers.[9] However, VPERM is a very expensive instruction with three source registers and one destination register, all 128 bits wide. Lee proposes a small set of subword permutation primitives useful for 2D multimedia processing.[13] While these subword permutation instructions can accelerate permutations of 8-bit or larger subwords, none defined so far can perform arbitrary bit-level permutations efficiently.

In searching the mathematical literature on permutations, we could not find any work on the efficiency of achieving arbitrary permutations. The problem of performing any arbitrary n-bit permutation in the minimum number of steps using permutation primitives (encoded in instructions) is thus far an unsolved problem.

Minimal number of steps for arbitrary n-bit permutations

The number of n-bit permutations is $n!$. According to Stirling's approximation,[14,15]

$$n! = O(n^n)$$

and

$$\lg(n!) = \Theta[n \lg(n)]$$

This approximation shows that the number of bits needed to specify one permutation is on the order of $n \lg(n)$. Sometimes permutations also replicate or omit some bits; we call these permutations with repetitions, or mappings. The repetition of bits increases the number of possible permutations to n^n. Therefore, the number of bits required to specify a permutation of n bits with repetitions is

$$\lg(n^n) = n \lg(n)$$

A reasonable design goal is to use the standard data path arrangements of current microprocessors with a general-purpose register file, as Figure 1 shows. Each instruction has two source registers and one destination register; no extra state (other than the data in the register file) is stored between instructions. Then, a permutation must use one source register for the n bits to be permuted, and the destination register gives an intermediate result leading to the final desired permutation. Because the second source register can hold only n bits of permutation specification and it takes $n \lg(n)$ bits to specify an n-bit permutation, a single permutation will require $\lg(n)$ permutation instructions. Hence, the minimum number of instructions needed for performing any arbitrary n-bit permutation is $\lg(n)$, assuming each instruction has only two source registers and one destination register, and stores no extra state. Similarly, the minimum number of instructions needed to perform any $2n$-bit permutation using n-bit registers is

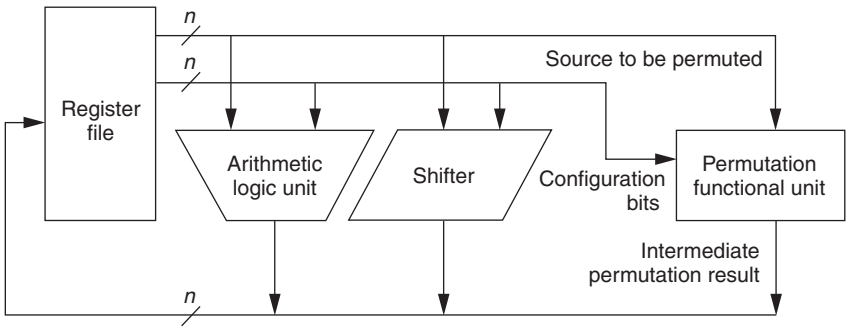

Figure 1. Standard microprocessor data path with a new permutation functional unit.

$2n \times \lg(2n) / n = 2 \lg(n) + 2$.

Alternative permutation methodologies

We start with bit-level permutations of all n bits in a register. We particularly consider n = 64, because today's microprocessors have registers and data paths that are either 32 or 64 bits wide, and common symmetric key block ciphers like DES and triple DES encrypt 64-bit blocks.[1] We describe four methods:

- PPERM, a new, lower-cost version of PPERM3R[16] that selects bits for one byte of the result;
- GRP, a permutation instruction that separates bits into left and right parts;
- CROSS, a permutation instruction using Benes interconnection network theory; and
- OMFLIP, a permutation instruction using enhanced Omega-Flip interconnection network theory.

In each case, the same solution must be applicable to different subword sizes that are powers of 2. The difficulty of permutations increases with the number of elements to be permuted, and the number of subwords in an n-bit register increases as the subword size decreases. For a fixed word size of n bits, clearly the hardest permutation problem is for 1-bit subwords, which requires permuting n subwords.

All four permutation instructions fall into the conventional two-source, one-result execution paradigm to let a permutation functional unit fit easily into the standard microprocessor data path of Figure 1.

PPERM

An intuitive way to do permutations is to explicitly specify the position in the source for each bit in the destination. This is what the PERMUTE instruction in MAX-2 does; it directs any subword in the source register to any subword position in the destination register.[5] The PERMUTE instruction currently supports only 16-bit subwords packed in a 64-bit word, so all position information can be encoded in eight bits of the instruction. To perform bit-level permutation, we generalize the PERMUTE instruction to support 1-bit subwords: Each bit is a subword, and the number of subwords is n. Therefore, supporting 1-bit subwords requires $n \lg(n)$ bits to specify the position information for n bits. So, performing a single permutation takes more than one instruction. To specify the source positions for k bits with one instruction, we define PPERM instructions as

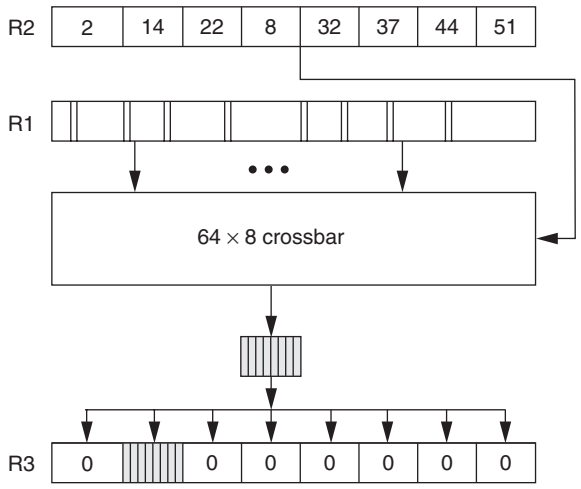

Figure 2. Diagram of flow of bits for PPERM,1 R1,R2,R3; 0x020E160820252C33. The numbers 2, 14, 22, 8, 32, 37, 44, and 51 are the bit positions in R1.

PPERM,*x Rs, Rc, Rd*

Rs and *Rc* are the source registers, and *Rd* is the destination register. *Rs* contains the bits to be permuted, *Rc* contains the configuration bits, and *Rd* gets the permuted result. Variable *x* specifies which k contiguous bits in *Rd* will receive source bits selected from *Rs*. The other bits in *Rd* are cleared to 0, as Figure 2 shows. The $k \lg(n)$ bits in *Rc* specify where to extract the k bits. To store the position information in one register, the following inequality should hold:

$$k \lg(n) \leq n$$

Therefore,

$$k \leq n / \lg(n)$$

Each instruction can specify approximately $n/\lg(n)$ bits. In total, around $n/k = \lg(n)$ instructions are required for an n-bit permutation. If n = 64 and k = 8, a permutation

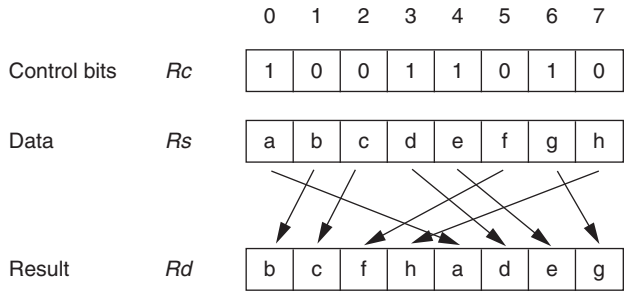

Figure 3. GRP instruction executed with 8-bit registers.

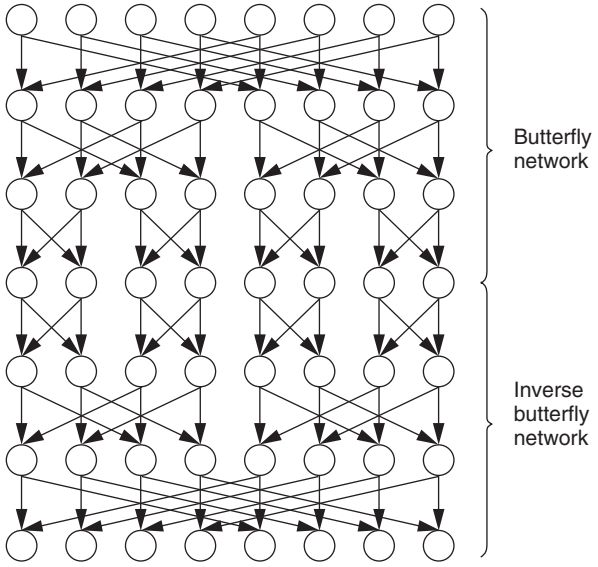

Figure 4. Eight-input Benes network.

requires eight PPERM instructions to permute bits, and seven OR instructions to merge the permuted bits into a permutation of 64 bits.

Specifying the configuration bits for the PPERM instruction is straightforward: For each of the k bits that will change in the final permutation, we use $\lg(n)$ bits to specify which bit in the source register goes into a particular position in the final permutation.

GRP

The GRP instruction also looks like any two-operand, one-result instruction for use on the typical microprocessors of Figure 1:

GRP *Rs, Rc, Rd*

Again, *Rs* and *Rc* are the source registers, and *Rd* is the destination register.

The basic idea of the GRP instruction is to sort the bits in *Rs* into left and right groups, according to the bits in *Rc*. The relative positions of bits in the left and right groups do not change. Concatenating these two groups gives the result in *Rd*. Figure 3 shows how the GRP instruction works on 8-bit registers.

For n-bit registers, we can do any n-bit permutations with no more than $\lg(n)$ GRP instructions, as has been proved by construction.[16]

CROSS

In interconnection networks such as the butterfly network, you can direct data at any input to any output by setting up the proper connections in the multistage network. A permutation of n elements (without repetitions) is like n pieces of data being routed simultaneously to n different destinations. Our goal is to find a network that can achieve this routing using edge-disjoint paths for any permutation of the inputs and in the minimum number of network stages. We will then try to simulate this network using primitive single-cycle permutation instructions.

The CROSS instruction is based on the Benes network, which results from connecting two butterfly networks of the same size, placed back-to-back. Figure 4 shows an eight-input Benes network. Eight bits placed at the eight inputs could be routed, in any permutation, to the eight outputs with edge-disjoint paths.[17] In addition, a Benes network has the following properties:

- An n-input Benes network can be broken into $2\lg(n)$ stages, with $\lg(n)$ distinct stages.
- In each stage of a Benes network, every input has two outputs to the next stage. For each input, there is another input that shares the same two outputs with it. We call these pairs of inputs conflict pairs. The connections for a conflict pair can be configured with one bit. The distances between conflict pairs are specific to stages.

Because of the existence of conflict pairs, configuring each stage of an n-input Benes network takes $n/2$ bits. If the configuration bit for a conflict pair is 0, the two inputs take

straight paths to the next stage. If it is 1, the two inputs cross paths to the next stage.

We can define a set of basic operations for a Benes network. Each basic operation corresponds to one distinct stage in a Benes network. Parameter m specifies a basic operation, where 2^m is the distance between conflict pairs for the corresponding stage. A basic operation uses $n/2$ configuration bits to set up the connections in the corresponding stage and move the n input bits to the output of that stage. Because an n-input Benes network has $\lg(n)$ distinct stages, it can be represented by $\lg(n)$ different, basic operations. With these basic operations, the CROSS instruction is defined as follows:

CROSS,*m1*,*m2 Rs, Rc, Rd*

Rs is the source register, which contains the bits to be permuted; *Rd* is the destination register for the permuted bits; and *Rc* is the configuration register. One CROSS instruction performs two basic operations on the source, according to the contents of the configuration register and values $m1$ and $m2$, which specify the two basic operations to use. Figure 5 shows the operation of a CROSS instruction.

To generate the CROSS instruction sequence for a specific permutation, we perform the following steps:

- Obtain a valid configuration for the desired permutation on a Benes network.[18]
- Break the configured Benes network into pairs of stages.
- For each pair of stages, assign a CROSS instruction. Each CROSS instruction uses the output from the previous CROSS instruction and generates input for the next CROSS instruction. The first CROSS instruction uses the original input; the last one generates the final result.

Because an n-input Benes network has $2\lg(n)$ stages, any n-bit permutation requires at most $\lg(n)$ CROSS instructions. Figure 6 shows an example permutation.

OMFLIP

We can achieve any permutation of n bits

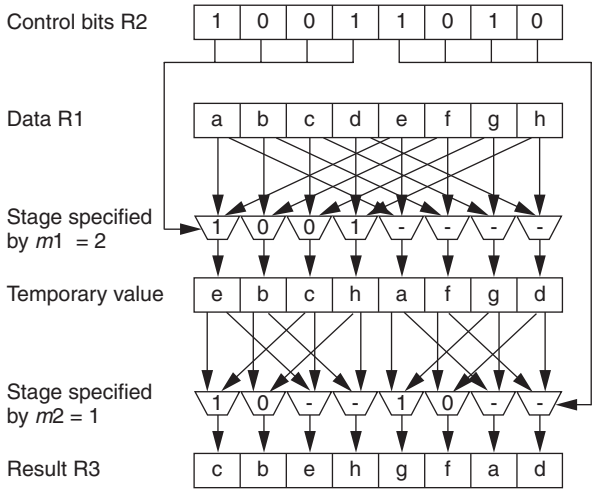

Figure 5. Operation of CROSS,2,1 R1,R2, R3. A "–" indicates a don't-care value.

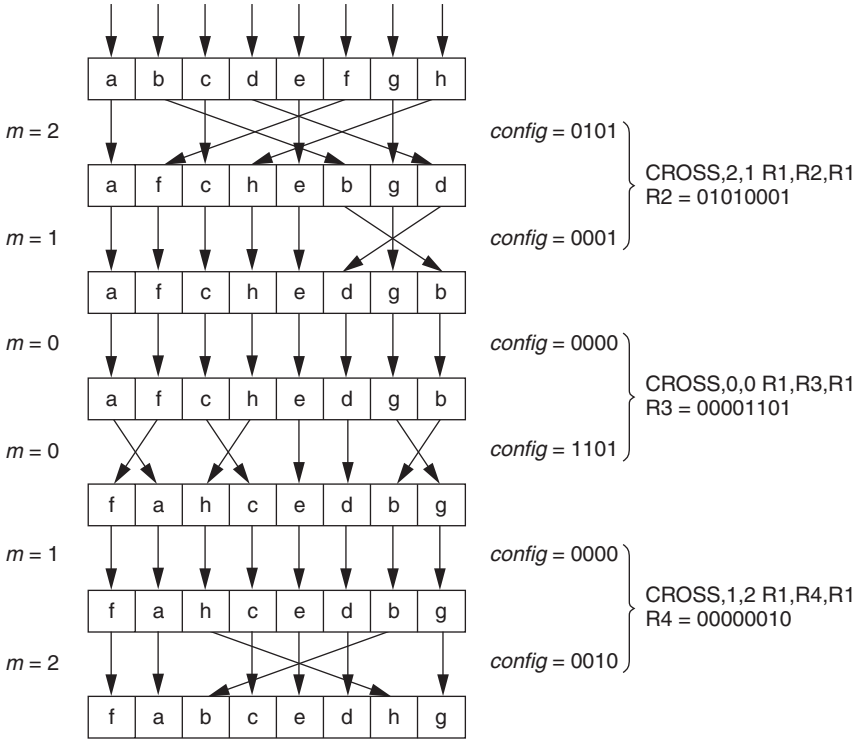

Figure 6. Example showing the permutation (abcdefgh) → (fabcedhg) on an eight-input Benes network.

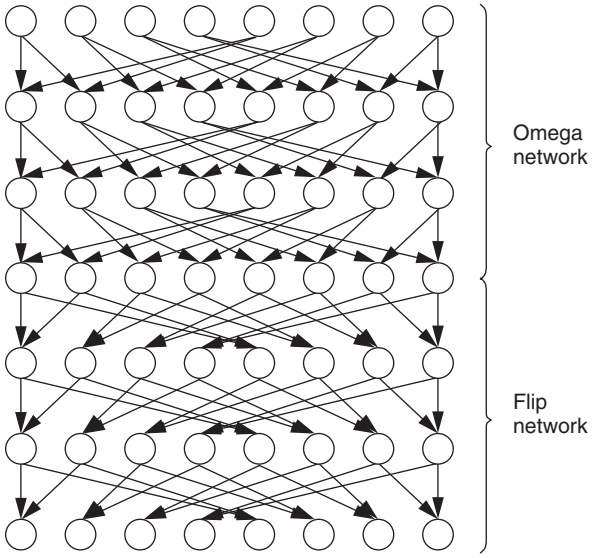

Figure 7. Eight-input omega-flip network.

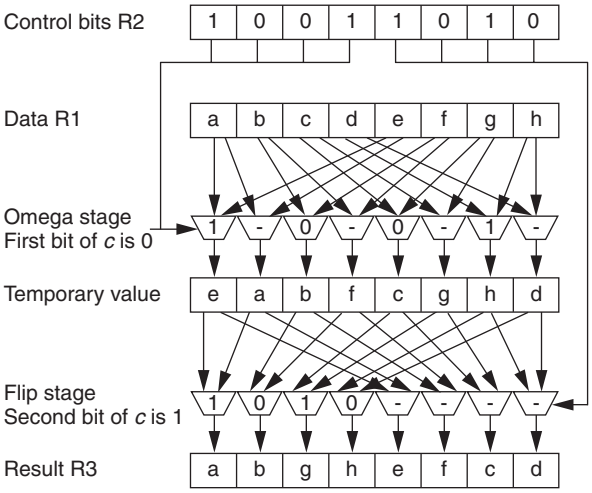

Figure 8. Operation of OMFLIP,01 R1,R2, R3.

using at most lg(*n*) CROSS instructions. However, because an *n*-input Benes network has lg(*n*) distinct stages and these stages are in a specific order, a permutation functional unit must have a whole Benes network in hardware to implement all CROSS instructions. This could consume more area than desired in a given processor.

An alternative solution is to use the omega-flip network, which can achieve the same performance as the CROSS instructions with far smaller hardware implementations. An omega-flip network consists of an omega network followed by a flip network. An *n*-input omega network has lg(*n*) identical omega stages. An *n*-input flip network is the exact mirror image of an *n*-input omega network. The flip network has lg(*n*) identical flip stages. Figure 7 shows an eight-input omega-flip network. An *n*-input omega-flip network is isomorphic to an *n*-input Benes network. As a result, they share common properties, such as conflict pairs. However, an *n*-input omega-flip network has only two distinct stages, as opposed to a Benes network's lg(*n*) distinct stages.

For an omega-flip network, we can define two basic operations, omega and flip, which accomplish the operations of an omega stage and a flip stage. Each uses *n*/2 configuration bits to set up the corresponding stage and move the input bits to their destinations. The configuration bit definitions follow the same convention as that for the Benes network. An OMFLIP instruction carries out two basic operations on the source word; we define it as

OMFLIP,*c Rs, Rc, Rd*

Rs, Rc, and *Rd* registers hold source bits, configuration bits, and the result. Parameter *c* is a two-bit subopcode that defines which two basic operations to use. For each bit in *c*, 0 indicates the omega operation, and 1 indicates the flip operation. Figure 8 shows the operation of an OMFLIP instruction.

Because the omega-flip network is isomorphic with respect to the Benes network, we can use the following steps to generate the OMFLIP instruction sequence for a specific permutation:

- Obtain a valid configuration for the desired permutation on a Benes network.
- Map the Benes network configuration to the omega-flip network based on the isomorphism between them.
- Break the configured omega-flip network into pairs of stages.
- For each pair of stages, assign an OMFLIP instruction. Each OMFLIP instruction uses the output from the previous OMFLIP instruction and generates input for the next OMFLIP instruction. The first OMFLIP instruction takes the original input; the last one generates the final result.

Because there are 2 lg(n) stages in an n-input omega-flip network, any n-bit permutation requires at most lg(n) OMFLIP instructions.

Yang and Lee describe an implementation of the OMFLIP instruction that is only four stages deep—two omega stages followed by two flip stages.[19] Each stage is enhanced with a pass-through path, where the input is passed unchanged to the output. This is sufficient to implement the four combinations of the two basic operations, omega and flip. This number of stages does not change with n—unlike the Benes network, which requires the implementation of 2 lg(n) stages.

We can also implement the OMFLIP instruction in only two "enhanced" omega-flip stages, where each enhanced stage implements either an omega operation or a flip operation.

Subwords, scalability, and repetition

We also want to investigate whether the permutation instructions just defined have other desirable features. For example, can we use them for efficient permutation of multibit subwords rather than just 1-bit subwords? Do they scale to allow efficient permutation of $2n$ bits rather than just n bits? Can they support permutations with repetitions (mappings)?

Multibit subword permutations. When subwords greater than one bit are packed into an n-bit register, this results in fewer subwords to permute. For example, with 8-bit subwords in a 64-bit register, there are only eight elements to permute, rather than 64 elements with 1-bit subwords. The permutation should also need fewer instructions: lg(8) = 3 rather than lg(64) = 6 instructions. But can the permutation instructions we have defined for bit-level permutations also allow efficient permutation of multibit subwords?

The PPERM instruction, as currently defined, is not efficient for multibit subword permutations. It can perform permutations of larger subwords packed in a register. But it cannot take advantage of the lower number of elements that have to be permuted to reduce the number of required permutation instructions.

The GRP instruction is as efficient for permuting multibit subwords as for 1-bit subwords. The number of GRP instructions required depends only on the number of elements to be permuted, and hence requires fewer GRP instructions to permute larger subwords. For example, on 64-bit processors, if the subword size is eight, there are only eight elements. Therefore, at most lg(8) = 3 instructions are enough to do any permutation of the eight 8-bit subwords.[16]

CROSS and OMFLIP instructions are also as efficient for multibit subwords as for 1-bit subwords. One important property of a Benes network is that when you configure it for k-bit subword permutations, the middle 2 lg(k) stages act as pass-throughs.[18] The same is true of the omega-flip network.[19] But you can eliminate these pass-through stages before assigning CROSS or OMFLIP instructions. Therefore, when permuting k-bit subwords in an n-bit word, the maximum number of instructions needed is lg(n) − lg(k) = lg(n/k) = lg(r), where r is the number of subwords.

Scalability to 2n bits. In the new AES[3] cryptography algorithms, 128-bit blocks are encrypted, rather than 64-bit blocks, as in DES.[2] Hence, we want to know if the permutation instructions defined can permute subwords packed into two n-bit registers. Here, we double the number of elements to permute. As discussed earlier, achieving any arbitrary permutation of $2n$ bits should require at least 2 lg(n) + 2 instructions. We wish to see how close each of the permutation instructions we have defined can come to this optimal number of instructions in executing arbitrary $2n$-bit permutations.

We can enhance PPERM's definition to allow permuting $2n$ bits stored in two n-bit registers. Because selecting one of 64 bits in the source register for a bit in the destination register requires only six index bits, and PPERM has eight such bits available, there are two extra bits per index. By defining one of these bits as the *otherreg* bit in each index, a PPERM instruction will pick bits from the source register only if *otherreg* = 0 for that index. If *otherreg* = 1, the PPERM instruction will clear the corresponding result bit to 0.

Permuting $2n$ bits requires two source registers and produces values for two destination registers. Each destination register works with

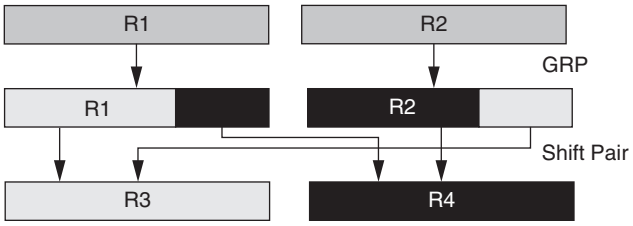

Figure 9. Separate bits with the GRP and Shift Pair instructions when doing 2*n*-bit permutations.

eight PPERM instructions for each source register. So combining the results into one destination register requires a total of 16 PPERM instructions and 15 OR instructions. Producing the result for the other destination register requires repeating the same number of instructions. Hence, permuting $2n$ bits requires a total of $2(16 + 15) = 62$ instructions. This is not efficient.

The GRP instruction scales efficiently to $2n$-bit permutations with the help of an instruction like the Shift Pair instruction, available in the PA-RISC[12] and IA-64[8] architectures. The Shift Pair instruction concatenates two source registers to form a doubleword value, and then extracts any contiguous single-word value. Suppose registers R1 and R2 store the bits to be permuted, and the results go to R3 and R4. Figure 9 shows how two GRP instructions followed by two Shift Pair instructions can separate the $2n$ bits into the appropriate left and right registers (R3 and R4) of the result. Finally, two separate n-bit permutations of R3 and R4 produce the desired $2n$-bit permutation.

In total, excluding the instructions needed for loading control bits, we need $4 + 2 \lg(n)$ instructions to perform a $2n$-bit permutation. This number is only two instructions more than the minimum number of instructions required. With 64-bit registers, a 128-bit permutation uses at most 16 instructions.

We can also use CROSS or OMFLIP instructions to permute $2n$ bits stored in two registers. We use a similar approach to that used with GRP instructions. We first permute each of the two n-bit words using CROSS or OMFLIP. This step separates the bits going to the left and right n-bit words in the result. Then we use two Shift Pair instructions to move the bits belonging to the left and right words into two registers. Finally, we permute the bits in these left and right registers to obtain the final result. The first step uses a maximum of $2 \lg(n)$ CROSS or OMFLIP instructions; the last step also uses at most $2 \lg(n)$ instructions. Therefore, it takes at most $4 \lg(n) + 2$ instructions to permute $2n$ bits using n-bit words. Compared to the ideal instruction count of $2 \lg(n)+2$, this method uses $2 \lg(n)$ more instructions. Hence, CROSS and OMFLIP instructions are not as efficient as the GRP instruction for $2n$-bit permutations.

Permutations with repetitions (mappings). So far, we have discussed permutations in which each bit in the source goes to a different bit in the destination. No bit in the source is repeated or omitted in the destination. We now consider the case in which bits in the destination replicate bits in the source.

PPERM can easily handle mappings, because it can select any of the n bits any number of times, a capability useful for some cryptographic algorithms. DES, for example, has an expansion permutation that replicates some bits. Currently, none of the other permutation instructions defined—GRP, CROSS, and OMFLIP—support such permutations with repetitions, or mappings.

To summarize, GRP, CROSS and OMFLIP instructions are all efficient for permuting multibit subwords. GRP is the most efficient for scaling up to bit permutations of $2n$ bits. CROSS and OMFLIP can perform $2n$-bit permutations but require $2 \lg(n) - 2$ more instructions than GRP. PPERM is not efficient for multibit subwords and can scale to permuting $2n$ bits, but not efficiently. However, PPERM is the only permutation alternative defined so far that can perform mappings.

Hardware requirements for new permutation instructions

We now consider the hardware complexity of the permutation functional unit corresponding to each of the new permutation instructions proposed. On a 64-bit processor architecture, the PPERM instruction requires a 64×8 crossbar network and a specialized shifter in hardware. These requirements arise because each PPERM instruction generates eight consecutive bits in the result. The eight bits can come from any of the 64 bits in the source and can go to a different but contiguous eight bits in the destination.

```
1     LDO              LDO              LDO     EXTRACT
2     LD               LD               LDO     EXTRACT
3     LD    GRP        LD      PPERM    LD      LDO
4     LD    GRP        LD      PPERM    LD      EXTRACT
5     LD    GRP        LD      PPERM    LDO     EXTRACT
6     LD    GRP        LD      PPERM    LD      LDO
7     LD    GRP        LD      PPERM    LD      EXTRACT
8           GRP        LD      PPERM    LDO     EXTRACT
9                      LD      PPERM    LD      LDO
10                     OR      PPERM    LD      EXTRACT
11                     OR      OR       LDO     EXTRACT
12                     OR      OR       LD      OR
13                     OR               LD      OR
14                     OR               OR      OR
15                                      OR      OR
16                                              OR
      (a)              (b)              (c)
```

```
1     LDO       TABLE(R27), R2 ; Load the base address of the table into R2
2     EXTRACT   R9, 7, 8, R3   ; Extract the index into R3
3     LD        R3(R2), R4     ; Load the kth 64-bit table entry into R4, where R3 contains index k
(d)
```

Figure 10. Instruction sequences for permutations using GRP, CROSS, and OMFLIP (a); PPERM (b); and table lookup (c) in a two-way superscalar machine. LD is a load instruction and LDO is a load offset instruction. A single table lookup uses three instructions (d).

Table 1. Maximum number of instructions or cycles for any permutation of 64 bits.

Characteristic	PPERM	GRP	CROSS or OMFLIP	Table lookup
No. of instructions without counting LDs of configuration registers	15	6	6	31
counting LDs of configuration registers	24	13	13	
No. of cycles on two-way superscalar machine with LDs of configuration registers	14	8	8	16

The GRP instruction requires implementing a hierarchical gathering network with lg(n) stage bits in hardware. A GRP instruction must separate bits in the source into two groups based on the corresponding configuration bits. The separation process is recursive.

To implement the CROSS instruction, a processor needs a full Benes network in hardware. For each CROSS instruction, we use two of the 2 lg(n) stages and bypass the rest.

The hardware required for the OMFLIP instruction is much simpler. An omega-flip network has only two distinct stages and each OMFLIP instruction can perform the operation of two stages. Thus, the hardware must implement at most four stages: two omega and two flip stages. For the execution of each OMFLIP instruction, we use two of the four stages and bypass the other two. Hence, implementing OMFLIP requires the smallest area.

Performance

Figure 10a shows the minimum instruction sequence for permuting 64 bits using either GRP, CROSS, or OMFLIP instructions, including set-up instructions for generating the address of the memory locations containing the six configuration words and loading these words into six registers. This process takes lg(n) + 2 = 8 cycles in a two-way superscalar machine. The PPERM instruction sequence in Figure 10b takes (n/8) + 6 = 14 cycles, and the table lookup method in Figure 10c takes 16 steps. Figure 10d shows the instructions required for one table lookup.

Table 1 summarizes the number of instructions and cycles required for 64-bit permutations. GRP, CROSS, and OMFLIP have the same, highest performance in all cases. Although PPERM and table lookup have similar execution times (in cycles) on a two-way

Table 2. Maximum number of instructions or cycles for any permutation of eight 8-bit subwords.

Characteristic	PPERM	GRP	CROSS or OMFLIP	Table lookup	EXTRACT and DEPOSIT
No. of instructions without counting LDs	15	3	3	31	16
counting LDs	24	7	7		
No. of cycles on two-way superscalar machines with LDs	14	5	5	16	9

Table 3. Memory storage requirement in 64-bit processors.

Characteristic	PPERM	GRP	CROSS or OMFLIP	Table lookup
Storage requirement (bytes)	64	48	48	16K

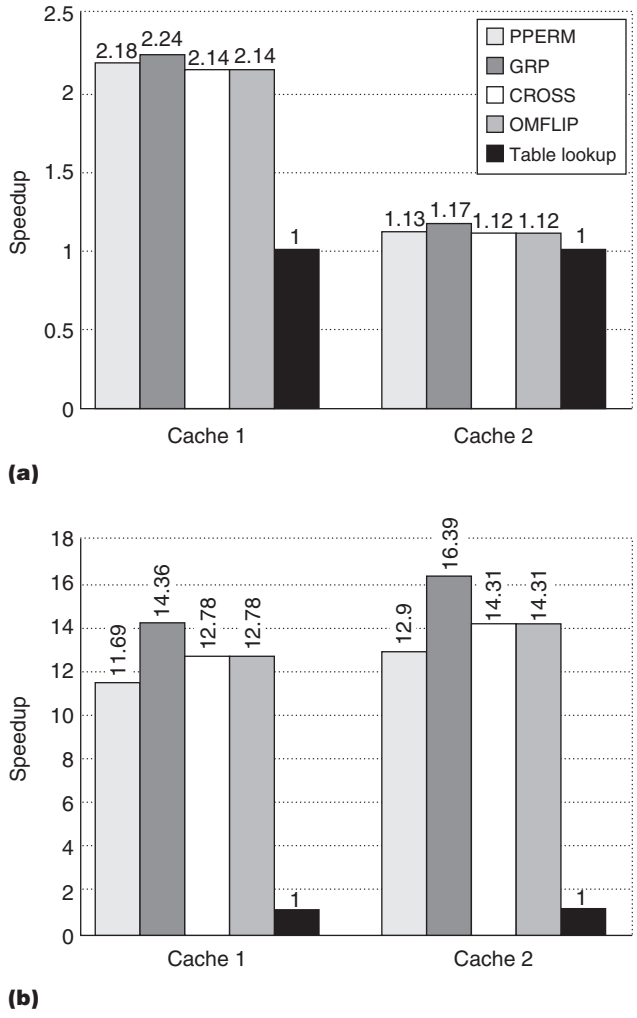

Figure 11. Speedup of DES for encryption/decryption (a) and key generation (b), based on two assumed cache configurations.

superscalar machine, PPERM requires far less memory than table lookup.

Table 2 compares the number of instructions and cycles for permuting eight 8-bit subwords. While GRP, CROSS, and OMFLIP reduce the number of instructions by half, PPERM and table lookup need the same number of instructions as for bit-level permutations. In this case, with only eight subwords to permute, using the EXTRACT and DEPOSIT instructions (in ISAs that have them[12,8]) is faster than table lookup when no permutation instruction is available in the ISA (see the last column of Table 1).

Table 3 summarizes the storage requirement for different methods. The table lookup method requires three orders of magnitude more storage, and the LD instructions in this method are more likely to cause cache misses. If a cache miss costs 50 cycles, then even if only two out of the eight LD instructions miss, this results in an additional 100 cycles to perform a 64-bit permutation.

For symmetric-key cryptography algorithms, there can be no need to load any configuration bits, because the permutation can be specified by dynamically generated data already in the general registers. Hence, no storage requirements (in Table 3) or load instructions (in Tables 1 and 2, and Figure 10) would be needed for the four permutation instructions.

Figure 11 shows the speedup of DES using the new permutation instructions (PPERM, GRP, CROSS, or OMFLIP) as compared to traditional software—without any of the new permutation instructions—that uses table lookups for permutations. We assume a 64-bit processor with two-way superscalar issue and one load/store unit. We consider two cache configurations: Cache 1 is a one-level split cache system, with 16 Kbytes of data cache memory and a cache miss penalty of 50 cycles. Cache 2 is a two-level cache system:

Table 4. Comparison of permutation alternatives.

Characteristic	PPERM	GRP	CROSS	OMFLIP	Table Lookup
Performance for arbitrary n-bit permutation	Medium $O[\lg(n)]$	Fast $\lg(n)$	Fast $\lg(n)$	Fast $\lg(n)$	Slowest
Permutations of fewer number ($r < n$) of multibit subwords	Inefficient $O[\lg(n)]$	Efficient $\lg(r)$	Efficient $\lg(r)$	Efficient $\lg(r)$	Inefficient
Scalability to $2n$ bits	Inefficient	Efficient	Less efficient	Less efficient	Inefficient
Permutations with bit repetitions	Efficient	No	No	No	Efficient
Memory requirements (for n = 64 bits)	Small (64 bytes or none)	Very small (48 bytes or none)	Very small (48 bytes or none)	Very small (48 bytes or none)	Large (16 Kbytes)
Area for permutation functional unit	Larger	Larger	Larger	Smallest	Not applicable
Single-cycle permutation primitive	Yes	Yes (complex circuitry)	Yes (with caveats)	Yes	Not applicable

The level-one cache has 16 Kbytes of data cache memory with a miss penalty of 10 cycles; the level-two cache is 256 Kbytes of unified cache with a miss penalty of 50 cycles. The cache 1 design could serve a low-cost information appliance, while cache 2 could support a desktop computer.

Figure 11a shows that the encryption speedup more than doubles for cache 1, and improves by 12 to 17 percent for cache 2. The new permutation instructions perform especially well on systems with a small cache. Figure 11b shows that the speedup of key generation increases tremendously—from 11 to 16 times—because the time taken to perform permutations dominates this computation.

Comparison of methods

Table 4 summarizes the relative advantages and disadvantages of our permutation methodologies and the table-lookup method for achieving any arbitrary permutation of n bits.

GRP, CROSS, and OMFLIP can perform any n-bit permutation in $\lg(n)$ instructions, not counting the loading of configurations into general registers. The value $\lg(n)$ is the minimum number of instructions, if each instruction has only two source registers and one destination register, and saves no other state between permutation instructions. If a program repeats a permutation many times, the cost of loading the configuration registers becomes negligible. Even counting the loading of configuration registers, it takes only $\lg(n) + 2$ cycles, if the processor is two-way superscalar. Most microprocessors today are at least two-way superscalar. PPERM needs more instructions and cycles to perform any n-bit permutation. Without one of these new permutation instructions, the fastest way to perform 64-bit permutations is with table lookup methods, which can be one to two orders of magnitude slower (in terms of cycles required), depending on the cache configuration.

All four methods can also permute any subword size that is a power of two with the same permutation functional unit. GRP, CROSS, and OMFLIP require only $\lg(r)$ instructions, where r is the number of subwords to be permuted. PPERM and the table lookup method cannot reduce the number of instructions needed to permute fewer elements.

The GRP instruction is the most efficient in terms of scaling up to perform a $2n$-bit permutation using n-bit registers. The PPERM instruction is advantageous in that it handles mappings with the same ease as permutations without repetitions. None of the other three permutation instructions currently handle permutations with repetitions.

All four instructions need to store at most $n \lg(n)$ configuration bits for a given permutation. For 64-bit permutations, this is three orders of magnitude less memory storage than the table lookup method for performing permutations in today's microprocessors, which need 16 Kbytes of table storage for one permutation. Furthermore, because cryptogra-

phy algorithms may wish to use dynamically generated data for the configuration bits, no storage may be needed for the permutation instructions, because this data will already be in the general registers.

Although the details of area and latency are beyond this article's scope, our analysis of hardware requirements shows that the OMFLIP permutation functional unit is the smallest in area. We could definitely implement OMFLIP in a single pipeline cycle, where the cycle time is between one to two times the arithmetic logic unit (ALU) latency. PPERM, CROSS, and GRP will all consume more area and likely have longer latencies than OMFLIP.

To have this fast permutation functionality, processors need only implement one of the four permutation instructions because they overlap in functionality. For the highest performance with the smallest area and shortest latency, designers would choose OMFLIP. For the highest performance with efficient scalability to permuting $2n$ bits with two registers, you would choose GRP. If permutations with repetitions are important, you would choose PPERM.

We have succeeded in reducing the number of instructions and cycles needed to perform any n-bit permutation from $O(n)$ to $O[\lg(n)]$. This is a huge speedup, especially as n increases. For $n = 64$ bits, any permutation can be done in no more than six permutation instructions or cycles, using either the GRP, CROSS, or OMFLIP instructions. Previously, this permutation would have taken up to 128 or 256 instructions using existing ISA instructions, or a similar number of cycles using table lookup methods, which incur cache miss penalties.

Future work will investigate if we can reduce this further from $O[\lg(n)]$ to $O(1)$ instructions or cycles. What are the most interesting or useful subsets of all possible permutations of n bits that can be done using just one or two instructions? Data-dependent rotations are one such subset. But are there larger subsets with better diffusion properties and other useful characteristics? Also, can we define permutation instructions that are both efficient for permutations with repetitions (mappings) and multibit subword permutations?

By providing the ability to do fast permutations in software, we not only speed up current cryptography and multimedia programs, but also open the field for new algorithms using these powerful yet simple permutation primitives. This facilitates far faster software cryptography and multimedia processing, combining high performance with flexibility and low cost. MICRO

References

1. B. Schneier, *Applied Cryptography*, 2nd ed., John Wiley & Sons, New York, 1996.
2. *Data Encryption Standard*, Nat'l Inst. Standards and Technology, FIPS PUB 46-2, Dec. 1993; http://www.itl.nist.gov/fipspubs/fip46-2.htm.
3. "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)," Nat'l Inst. Standards and Technology, 12 Sept. 1997; http://csrc.nist.gov/encryption/aes/pre-round1/aes_9709.htm.
4. R. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, vol. 15, no. 2, Mar./Apr. 1995, pp. 22-32.
5. R. Lee, "Subword Parallelism in MAX-2," *IEEE Micro*, vol. 16, no. 4, July/Aug. 1996, pp. 51-59.
6. M. Tremblay et al., "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, July/Aug. 1996, pp. 35-42.
7. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 16, no. 4, July/Aug. 1996, pp. 10-20.
8. *IA-64 Application Developers' Architecture Guide*, Intel, Santa Clara, Calif., 1996.
9. *AltiVec Technology Programming Environment Manual*, *Revision 0.1,* No. ALTIVEC-PEM/D, Motorola, Austin, Tex., Nov. 1998.
10. *AMD Extensions to the 3DNow! and MMX Instruction Set Manual,* No. 22466 D/O, AMD, Sunnyvale, Calif., Mar. 2000.
11. *Intel Architecture Software Developer's Manual*, Intel, Santa Clara, Calif.; http://developer.intel.com/design/PentiumIII.
12. R. Lee, "Precision Architecture," *Computer*, vol. 22, no. 1, Jan. 1989, pp. 78-91.
13. R. Lee, "Subword Permutation Instructions for Two-Dimensional Multimedia Processing

in MicroSIMD Architectures," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors* (ASAP 00), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 3-14.
14. M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions*, 9th printing, US Dept. of Commerce and National Bureau of Standards, Washington, D.C., 1970.
15. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1994.
16. Z. Shi and R. Lee, "Bit Permutation Instructions for Accelerating Software Cryptography," *Proc. IEEE Int'l Conf. Application-Specific Systems, Architectures, and Processors* (ASAP 00), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 138-148.
17. F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Francisco, 1992.
18. X. Yang, M. Vachharajani, and R. Lee, "Fast Subword Permutation Instructions Based on Butterfly Networks," *Proc. Media Processors 2000*, SPIE, Bellevue, Wash., 2000, pp. 80-86.
19. X. Yang and R. Lee, "Fast Subword Permutation Instructions Using Omega and Flip Network Stages," *2000 IEEE Int'l Conf. Computer Design: VLSI in Computers & Processors* (ICCD 00), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 15-22.

Ruby B. Lee is the Forrest G. Hamrick Professor in Engineering and professor of electrical engineering at Princeton University. She is also the director of the Princeton Architecture Lab for Multimedia and Security (PALMS). Her research interests include designing security and multimedia features into core processor and platform architectures for pervasive security and ubiquitous multimedia information processing. Lee has an AB from Cornell University; and an MS in computer science and computer engineering, and a PhD in electrical engineering from Stanford University. She is a senior member of the IEEE, and a member of ACM, SPIE, Phi Beta Kappa, and Alpha Lambda Delta. She holds 88 US and international patents.

Zhijie Shi is a PhD student at Princeton University. His research interests include computer architecture for fast cryptography and multimedia. Shi has a MS in computer science from Tsinghua University and an MA in electrical engineering from Princeton University.

Xiao Yang is a PhD student at Princeton University. His research interests include high-performance architecture for multimedia, especially 3D graphics. Yang has an MS in physics from Northwestern University.

Direct questions and comments about this article to Ruby B. Lee, Princeton University, Dept. of Electrical Engineering, B-218, Engineering Quadrangle, Princeton, NJ 08544; rblee@ee.princeton.edu.