

Alternative Application-Specific Processor Architectures for Fast Arbitrary Bit Permutations

Zhijie Jerry Shi, Xiao Yang, and Ruby B. Lee

Abstract—Block ciphers are used to encrypt data and provide data confidentiality. For interoperability reasons, it is desirable to support a variety of block ciphers efficiently. Of the basic operations in block ciphers, only bit permutation is very slow on existing processors, followed by integer multiplication. Although new permutation instructions proposed recently can accelerate bit permutations in general-purpose processors, reducing the number of instructions needed to achieve an arbitrary n -bit permutation from $O(n)$ to $O(\log_2(n))$, the data dependency between permutation instructions prevents them from being executed in fewer than $\log_2(n)$ cycles, even on superscalar processors. Since Application-Specific Instruction-Set Processors (ASIPs) have fewer constraints on maintaining standard processor datapath and control conventions, six alternative ASIP approaches are proposed in this paper to achieve arbitrary 64-bit permutations in one or two cycles without increasing the cycle time. These approaches use new BFLY and IBFLY instructions. We also compare these approaches and their efficiency in performing arbitrary 64-bit permutations.

Index Terms—application specific instruction-set processor, ASIP, bit permutation, block cipher, confidentiality, cryptography acceleration, embedded system, instruction set architecture, software encryption, symmetric-key cipher.

I. INTRODUCTION

Block ciphers are symmetric-key cryptographic algorithms used to encrypt data and provide confidentiality for network transactions and stored data. Such support for confidential information is important on all networked computers and computing devices due to the ease of eavesdropping attacks on the public Internet and wireless networks. Standard security protocols, including SSL and IPSEC, support a large variety of cryptographic algorithms. The cipher suite to be used is often negotiated at the beginning of a communication session. Different classes of cryptographic algorithms are needed for achieving user

authentication, data confidentiality, and data integrity. Even within a given class of algorithms, such as symmetric-key block ciphers, it is desirable to support a wide spectrum of ciphers efficiently, for interoperability reasons. As secure computing paradigms become more pervasive, it is likely that such cryptographic computations will become a major component of every processor's workload. Understanding the new requirements of secure information processing is important for the design of all future programmable processors, whether general-purpose, application-specific, or embedded.

In this paper, we first investigate which operations are frequently used by symmetric-key ciphers but not efficiently supported by existing processors. We show that arbitrary bit permutations are the only operations not efficiently supported in existing processors. We then ask whether a new application-specific instruction-set processor (ASIP) designed to support both current and future symmetric-key ciphers should support fast bit permutations? The other candidates are data-dependent rotations (a smaller set of permutations) and multiply instructions – both of which are already supported by some processors.

Symmetric-key ciphers are composed of operations that achieve *confusion* and *diffusion* in transforming the plaintext message into the encrypted ciphertext [1]. Permutation is very effective in achieving diffusion in symmetric-key algorithms. Bit permutation, achieving many bit-level effects more efficiently than word-level operations, is potentially a much more powerful operation than data-dependent rotation or multiplication for cryptographic algorithms. It has been demonstrated that some bit permutation instructions are very effective in constructing fast and secure block ciphers [2], [3].

Arbitrary bit permutations achieve any one of $n!$ outcomes rather than just one of n outcomes achieved by rotations. When bit permutation is compared to multiplication, a permutation functional unit can be implemented with less area than a multiplier. An n -bit multiplier is typically four times the area of an n -bit ALU, and takes there to five times the latency. The question we examine in this paper is whether n -bit permutations can be achieved in less time and cost than n -bit multiplications and without much incremental time and cost compared to the much smaller set of data-dependent rotations. This implies that a permutation functional unit should be smaller than a multiplier, take less than three cycles of latency, and ideally take just one or two cycles of latency like a data-dependent rotation. Such fast permutation

Manuscript received June 11, 2004. This work was supported in part by the U.S. National Science Foundation (NSF) under grants CCR-0105677 and CCR-0326372.

Z. J. Shi was with Princeton University. He is now with the Computer Science and Engineering Department, University of Connecticut, Storrs, CT 06269 USA (phone: 860-486-0599; fax: 860-486-4817; e-mail: zshi@cse.uconn.edu).

X. Yang is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: xiaoyang@princeton.edu).

R. B. Lee is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: rblee@princeton.edu).

operations can accelerate many important symmetric-key ciphers and unleash the opportunity to design and use faster new ciphers.

In the rest of the paper, Section II describes the operations used in block ciphers. Section III describes past work on permutation instructions, including how recent past work has reduced the time taken to achieve any n -bit permutation down from $O(n)$ to $O(\log_2(n))$ instructions and cycles. Section IV investigates the latency through different permutation functional units, in order to determine which of these can be executed in one cycle on a typical processor. Section V describes alternatives for achieving arbitrary bit permutations in at most two cycles. We compare these alternatives in Section VI and conclude in Section VII.

II. OPERATIONS IN BLOCK CIPHERS

Table 1 shows the basic operations used by a range of popular block ciphers such as DES (Data Encryption Standard [4]) and AES (Advanced Encryption Standard [5]). All algorithms use the Arithmetic and Logic Unit (ALU) operations: add, subtract, or logical operations. Most use some form of table lookup, accomplished with memory load instructions. Three ciphers use integer multiplication while three others use bit permutations. Except for one cipher, all require some form of fixed rotation, variable (or data-dependent) rotation, or bit permutation.

Table 2 presents the typical number of instructions and cycles to perform the operations in block ciphers. All current microprocessors, embedded processor cores, and digital signal processors support ALU instructions, memory access instructions, and shift instructions. These operations normally take only one instruction and can be executed in one cycle, except for the memory access instructions which may take hundreds of cycles if cache misses occur. Not all processors support integer multiply, and on those that do, often only 16-bit multiplies are supported. Even if the multiplication is supported by hardware, it takes several cycles to execute. Furthermore, the ciphers typically use 32-bit multiplies [6], [7]. Not all processors support fixed or variable rotations. When rotations are supported, they may be done in one cycle; otherwise, they have to be emulated with other instructions, taking up to five cycles. Currently, no processor supports bit permutation operations directly, which have to be done with other instructions, either with logical instructions or table lookups [8]. This makes bit permutation the slowest of the operations in block ciphers. For example, with the table lookup method, performing arbitrary 64-bit permutations with 64-bit processors may take 23 instructions and hundreds of cycles, if cache misses occur.

III. PAST WORK IN BIT PERMUTATION INSTRUCTIONS

With existing processor instruction set architectures (ISAs), arbitrary bit permutations can be done using logical operations or table lookups [3], [8], [9]. While some simple n -bit permutations can be accomplished with a few instructions, a

generic way to achieve any arbitrary n -bit permutation with current ISAs takes $O(n)$ instructions, which is unacceptably slow. For example, each of the n bits has to be selected (with an AND instruction), shifted to its new position (with a SHIFT instruction), and then combined (with an OR instruction) with previously permuted bits. While the number of instructions can be reduced by table lookup methods, these can only achieve a small set of fixed permutations due to the memory space required for each set of tables representing a fixed permutation. Also, the memory latency and cache misses caused by table lookup can degrade performance significantly in terms of the actual execution cycles that are taken.

Recently, several different methods have been proposed to perform arbitrary n -bit permutations, reducing the number of instructions needed from $O(n)$ down to $O(\log_2(n))$ [8] – [12]. The new permutation instructions proposed each require two operands and one result, following standard processor conventions. Table 3 shows the number of instructions and cycles needed to achieve an arbitrary n -bit permutation, for $n=64$ bits, for each of these methods.

The GRP [8] instruction is based on partitioning the n bits to be permuted into two parts, based on whether the corresponding configuration bit is 0 or 1. A sequence of $\log_2(n)$ GRP instructions can perform any of the $n!$ permutations of n bits.

The CROSS [10] or OMFLIP [11] instructions build a virtual Benes network or omega-flip network, respectively, to permute n bits. A Benes network for permuting n bits consists of a butterfly network followed by an inverse butterfly network, each of which has $\log_2(n)$ stages. Figure 1 shows an 8-bit butterfly network and an 8-bit inverse butterfly network, which can be used to form a 6-stage ($2\log_2(8)$) Benes network. An omega-flip network is isomorphic to a Benes network. Each CROSS or OMFLIP instruction executes the equivalent of two stages of the network. Hence, both can achieve any one of the $n!$ permutations in at most $\log_2(n)$ instructions. The $\log_2(n)$ instructions form a data-dependent sequence, where an intermediate permutation generated by one instruction is used in the next permutation instruction. Consequently, these $\log_2(n)$ instructions cannot be executed in fewer than $\log_2(n)$ cycles, even when the processor can execute more than one instruction per cycle.

The PPERM [9] instruction and the pair of instructions, SWPERM and SIEVE [12], are based on selecting a source bit by its numeric index. They both need more than $\log_2(n)$ instructions to do an arbitrary n -bit permutation. However, these instructions have less serial data-dependencies and can be executed in fewer than $\log_2(n)$ cycles on processors which can execute more than one instruction per cycle.

In this paper, we improve upon these previous permutation methods. Our new method achieves an arbitrary 64-bit permutation in one or two cycles, rather than $\log_2(64)=6$ cycles. Specifically, our goal is to achieve arbitrary n -bit permutations with:

- Fewer than $\log_2(n)$ instructions and cycles;

- No significant increase in cycle time;
- Low cost (low datapath and control path overhead).

In [13], Lee first proposed reducing the number of cycles for arbitrary bit permutations down to one or two cycles, a level comparable to data-dependent rotations, and solved the problem for general-purpose superscalar processors. In this paper, we discuss several ASIP solutions to achieve this performance.

IV. CYCLE TIME AND LATENCY OF PERMUTATION UNITS

This section investigates the cycle time impact of different permutation functional units. Processor cycle time is often determined with respect to the latency of the ALU, because add, subtract, and logical instructions are usually the most frequently used instructions. Typically, an ALU instruction executes in a single cycle. We seek an n -bit permutation functional unit with a latency comparable to that of an n -bit ALU, so that the corresponding permutation instruction can achieve single-cycle execution in most processors.

Table 4 compares the latency of different permutation functional units in terms of the “logical effort” [14] required to implement them. Logical effort gives an estimate of the critical path of a circuit in a technology independent way. Latency is measured in terms of fan-out of four (FO4) delays, where one FO4 is the delay of an inverter that drives four identical inverters. In Table 4, we compare the latencies of a 64-bit ALU and 64-bit permutation functional units implementing the GRP, OMFLIP, and CROSS instructions in terms of FO4 delays.

GRP uses a hierarchical network to partition data bits [3], [15], [16]. When performing a GRP operation, all the stages in a GRP functional unit are used. Performing two GRP operations requires two GRP units that are connected in series. If only one GRP functional unit is available, two GRP operations require data bits to go through the GRP unit twice. To permute n bits with $\log_2(n)$ GRP instructions, the data bits need to go through the GRP unit $\log_2(n)$ times. With existing GRP implementations, the delay of a GRP functional unit is longer than an ALU latency, e.g., when $n = 64$. Therefore, two GRP operations can not be done in one cycle on most processors.

An OMFLIP instruction can be implemented by a permutation functional unit with four stages, two omega stages and two flip stages, and some pass-through connections [11], [16]. Although only two of these four stages are used by one OMFLIP instruction, the data bits have to go through all the stages. This makes the delay of a 4-stage omega-flip network comparable to the delay of an ALU.

To permute 64 bits, a CROSS implementation needs a 12-stage Benes network, consisting of a 6-stage butterfly network followed by a 6-stage inverse butterfly network [9]. Although only two stages are used by a CROSS instruction, all 12 stages must be implemented because any two stages may be used. The 12-stage network yields a long latency of 27.2 FO4 units,

much greater than an ALU latency.

We observe, however, that a 6-stage butterfly network has a delay shorter than one ALU latency. Both an n -input butterfly network and an n -bit ALU need almost the same number of $\log_2(n)$ stages; but the stages of the butterfly network are simpler since each stage is just a row of 2:1 multiplexers (see Figure 1). A 6-stage omega network has a longer latency because each stage has long wires, whereas some stages of a 6-stage butterfly network have very short wires.

This latency analysis suggests that the data bits can go through a full 64-bit butterfly network or inverse butterfly network in a single cycle and that an n -bit permutation can be achieved in two cycles, using a butterfly functional unit in the first cycle and an inverse butterfly functional unit in the second cycle. We propose two new instructions, BFLY and IBFLY, to achieve this. BFLY uses a butterfly network to permute bits and IBFLY uses an inverse butterfly network.

The remaining difficulty is: how can we get the many configuration bits to the butterfly network (or the inverse butterfly network)? For $n=64$, a 6-stage butterfly network requires $3n$ configuration bits, $n/2 = 32$ bits for each stage. Together with the n data bits to be permuted, $4n$ bits, or four 64-bit source operands, need to be supplied to each BFLY or IBFLY instruction. In a typical microprocessor, only two source operands are supplied to each instruction. We solve this problem in the rest of this paper.

V. ALTERNATIVE ASIP ARCHITECTURES

We now examine how an operation with four source operands can be performed on a variety of ASIP architectures, where both the processor’s instruction set architecture and micro-architecture have more flexibilities [17], [18].

We illustrate with the following example. Assume a 64-bit processor. The bits to be permuted are placed in R1. Both the butterfly functional unit and the inverse butterfly functional unit have $\log_2(64)=6$ stages. The configuration bits for the 6-stage butterfly network are in R11, R12, and R13, and those for the 6-stage inverse butterfly network are in R14, R15, and R16.

Six CROSS instructions can perform any permutation of the 64 bits in R1, as shown below. The subop encodings, 5, 4, ..., 0, indicate which of the six different stages of the butterfly or inverse butterfly network are used in a CROSS instruction. Because of data-dependencies between consecutive CROSS instructions, these six instructions take six cycles to perform.

I1: CROSS.5.4	R1, R1, R11
I2: CROSS.3.2	R1, R1, R12
I3: CROSS.1.0	R1, R1, R13
I4: CROSS.0.1	R1, R1, R14
I5: CROSS.2.3	R1, R1, R15
I6: CROSS.4.5	R1, R1, R16

We will show how this permutation can be done in two cycles. We use BFLY to represent an instruction using a 6-stage butterfly network and use IBFLY to represent an instruction using a 6-stage inverse butterfly network. We define these instructions, with minor variations, for each of six

alternative solutions.

A. *LdState*

First, suppose the permutation functional units have internal storage to keep configuration bits, as shown in Figure 2a. We load configuration bits into the permutation units first, and then send the data bits (and another n control bits) to perform the permutation. Each network has two 64-bit internal registers to configure four stages.

The permutation instructions are defined in Figure 3a. *LdState.bfly* and *LdState.ibfly* load the value of two source registers into the internal registers, *C1* and *C2* in the butterfly network, and *C4* and *C5* in the inverse butterfly network, respectively. *BFLY* specifies the data to be permuted (R_s) and the configuration bits for the last two stages of the butterfly network (R_{c3}). The bits in R_s are permuted with the butterfly network configured by *C1*, *C2*, and R_{c3} and then stored in the destination register R_d . Similarly, *IBFLY* permutes the bits in R_s with an inverse butterfly network configured with *C4*, *C5*, and R_{c3} , and saves the permuted bits in R_d . Four instructions can achieve any of the $64!$ permutations of the 64 bits in $R1$ in four cycles:

<i>LdState.bfly</i>	$R11, R12$
<i>BFLY</i>	$R1, R1, R13$
<i>LdState.ibfly</i>	$R14, R15$
<i>IBFLY</i>	$R1, R1, R16$

Since the configuration bits for the first four stages are still in the permutation units, if the same permutation needs to be performed again, only two instructions, *BFLY* and *IBFLY*, are needed, taking only two cycles. Since *BFLY* and *IBFLY* use different functional units, permutations on different data registers may be pipelined on multi-issue processors to achieve a throughput of one permutation per cycle.

When multiple processes share the same functional units, the internal registers have to be saved and restored during context switches, incurring operating system overhead. To save the internal registers, another instruction, *MovePUtoGR*, has to be defined to move the configuration bits from the permutation unit's internal registers to general registers. Four instructions are needed for this state saving, since four general registers have to be written, one per instruction. The general registers are then saved in the normal manner for context switches or interrupts.

B. *Register pair*

The *LdState*'s internal registers increase the complexity of the permutation functional units, and cause context-switch overhead. Without the internal registers, four 64-bit values have to be provided to the butterfly network or the inverse butterfly network in one instruction. Normally, two source register specifiers are allowed in one instruction. In order to minimize the number of instructions, we can treat the four registers as two register pairs and define that each of the two source register specifiers in a permutation instruction specifies a register pair instead of a single register. The instructions are

defined in Figure 3b. Although only two source registers, R_{s1} and R_{s2} , are specified in the *BFLY* instruction, they actually refer to four registers: R_{s1} , $R_{(s1+1)}$, R_{s2} , and $R_{(s2+1)}$. The *BFLY* instruction permutes bits in R_{s1} with configuration bits in R_{s2} , $R_{(s2+1)}$, and $R_{(s1+1)}$. Similarly, the *IBFLY* instruction permutes bits in R_{s1} with configuration bits in R_{s2} , $R_{(s2+1)}$, and $R_{(s1+1)}$.

The functional units shown in Figure 2b can perform *BFLY* and *IBFLY* defined here. They take four inputs and generate one result. We call this kind of permutation functional unit a (4,1)-PU. One of the four inputs is the data bits to be permuted, and the other three configure the butterfly or the inverse butterfly network. Four registers specified with two register pairs are read in one cycle, and all four values are fed into the (4,1)-PUs simultaneously. Figure 4a shows a processor datapath with (4,1)-PUs. The dashed lines are register bypass paths. The shaded gray area represents the permutation functional units shown in Figure 2b.

This method requires a register file to have four read ports to read two register pairs in one cycle. Also, the decode logic has the overhead of checking the extra data availability for the *BFLY* and *IBFLY* instructions. In addition, this method causes compiler complexity due to the register pairing. Furthermore, one of the register pairs specifies both data bits and configuration bits, which may cause extra register move instructions.

C. *Two-length ISA*

Register pairing causes overhead to the compiler and decode/issue logic. To remedy this, we can explicitly specify four source registers in one instruction. The permutation instructions are defined in Figure 3c. The *BFLY* and *IBFLY* instructions permute bits in R_s with configuration bits in R_{c1} , R_{c2} , and R_{c3} and put the permuted bits in R_d . It takes at most two instructions to achieve an arbitrary permutation of the n bits in $R1$ in our example.

The functional units and processor datapath are shown in Figure 2b and Figure 4a. Each *BFLY* and *IBFLY* takes only one cycle, and any permutation needs at most two cycles.

The problem with this method is that we may not have enough bits in the instruction word to specify all five registers. Suppose there are 32 registers, and an instruction length of 32 bits. We need five bits to specify each register and 25 bits for all five registers. Only seven bits are left in the instruction for specifying the opcode and other fields, which may not be enough. Longer instructions may have to be used. Hence, this method is called the two-length ISA method; each instruction is either a short instruction with two source operands or a longer instruction with four source operands. But variable length instructions introduce overhead to the fetch and decode logic. One alternative is to write the result into the original data register, as shown in Figure 3d. This can probably be encoded in 32 bits, allowing all instructions to have a fixed length of 32 bits. However, for ISAs with instructions shorter than 32 bits, permutation instructions may

need to use a longer format, since they still have one extra register specifier.

Similar to the register pairing method, this method requires four register read ports. Because most other instructions use only two register read ports, it may be wasteful to add two register read ports just for the permutation instructions.

D. Bundled instruction

The two-length ISA complicates the fetch and decode logic. It may be desirable to use the same instruction format for permutation instructions as for other instructions. We achieve this by using a bundle of two instructions to deliver the four source registers: a BFLY.ct1 instruction with a BFLY instruction, or an IBFLY.ct1 instruction with an IBFLY instruction, as shown in Figure 3e. Each pair of instructions jointly specifies the data register R_s and the configuration registers: R_{c1} , R_{c2} , and R_{c3} . Instructions in a bundle must be executed together so that all four registers can be fed into the permutation units. Although we have four instructions, they can be executed in two cycles. The functional units and datapath are the same as shown in Figure 2b and Figure 4a.

This method incurs control overhead to detect bundles. For example, an instruction buffer is needed to store two instructions, and control overhead to detect a sequence of two bundled instructions. In addition, we still need four register read ports to accommodate a bundle of two instructions. The execution of a bundle will be fetch-bound in a single-issue processor where only one instruction can be fetched per cycle. This fetch bandwidth limitation can result in a 4-cycle latency for an n -bit permutation, even though the bundle can otherwise execute in two cycles.

E. Superscalar execution

The register pairing, two-length ISA, and bundled instruction methods use the datapath shown in Figure 4a. There are four register read ports and four source data buses with bypasses. But the ALU uses only two of them. The other two ports and buses are for permutation operations only. We may add another ALU into the datapath as shown in Figure 4b. This makes it possible to do 2-way superscalar execution; processors execute two ALU instructions simultaneously, utilizing all four input buses each cycle. For the permutation operations, we use the instructions defined in Figure 3e. In this case, the two instructions are just issued in the same cycle. Each instruction reads two registers. Four register values are fed into the (4,1)-PUs. Thus, an arbitrary n -bit permutation can be done in two cycles.

By increasing the issue width to 4-way superscalar execution, the butterfly functional unit and the inverse butterfly functional unit can perform different permutations in the same cycle. When performing multiple permutations, the process can be pipelined so that one permutation can complete every cycle. For example, when we perform the same permutation on R_1 , R_2 , R_3 , and R_4 , one permutation is completed every cycle starting from cycle 2.

Cycle 1:	BFLY.ct1	R11,R12	BFLY	R1,R1,R13
Cycle 2:	IBFLY.ct1	R14,R15	IBFLY	R1,R1,R16
	BFLY.ct1	R11,R12	BFLY	R2,R2,R13
Cycle 3:	IBFLY.ct1	R14,R15	IBFLY	R2,R2,R16
	BFLY.ct1	R11,R12	BFLY	R3,R3,R13
Cycle 4:	IBFLY.ct1	R14,R15	IBFLY	R3,R3,R16
	BFLY.ct1	R11,R12	BFLY	R4,R4,R13
Cycle 5:	IBFLY.ct1	R14,R15	IBFLY	R4,R4,R16

This method incurs the control overhead for superscalar execution, which is significant if compared to single-issue execution. In addition, a pair of permutation instructions, e.g., BFLY.ct1 and BFLY, needs to be detected and issued in the same cycle. However, compared to a conventional superscalar processor, the overhead of detecting and issuing a pair of instructions is relatively minor.

F. VLIW execution

Due to the complexity of control logic in superscalar execution, we can use VLIW (Very Long Instruction Word) to utilize the multiple ALUs in Figure 4b. The advantage of the VLIW execution is that the issue logic is much simpler than for the superscalar execution because the instructions to be executed together are already packed in one long instruction word during compile time. The issue logic does not need to perform complicated dependency checks. That complexity is done once at compile time. We use the same instructions defined in Figure 3e and require that an instruction bundle be put in the same VLIW instruction. Thus the proper instruction pair will always be executed simultaneously. To permute R_1 in our example, four instructions are placed in two long instruction words, and they can be executed together with other instructions packed in the same long instruction words.

This method does not need to combine instructions or modify the issue logic. In addition, there are no internal registers in the permutation functional unit. VLIW execution achieves the same performance as superscalar execution with the same degree of instruction level parallelism, but with less control complexity.

VI. COMPARISON

Table 5 compares the six alternative implementations of the BFLY and IBFLY permutation instructions. The last column shows the prior methods, GRP, OMFLIP, or CROSS permutation instructions, which all need $\log_2(n)$ instructions and cycles. The first two rows of Table 5 show that all the six new architectural alternatives for BFLY/IBFLY can do an arbitrary n -bit permutation in less than $\log_2(n)$ instructions, taking at most two cycles. This 2-cycle latency is achieved with simple single-issue processors, except for the bundled instruction and superscalar methods. Although the VLIW method issues only one instruction per cycle, this is a long instruction consisting of two regular instructions equivalent to the superscalar method with an ILP (Instruction Level Parallelism) of two. However, the VLIW method is simpler; it does not have to check at runtime whether the two

instructions can issue together, since this checking has been done at compile time.

With multi-issue processors which can issue either two or four instructions each cycle, a maximum throughput of one permutation per cycle can be achieved, at the cost of more register ports, more operand buses, and increased control complexity.

Only the superscalar and VLIW methods enable additional performance by allowing more than one ALU instruction to execute in one cycle. This results in the highest speedup of 1.68 times for a symmetric-key cryptographic algorithm like DES, compared to best prior work (last column), which already uses permutation instructions like GRP, CROSS, or OMFLIP.

The speedup of DES in Table 5 compares the ASIP alternatives in this paper with the best prior work using the permutation instructions like GRP, OMFLIP, and CROSS, which perform arbitrary n -bit permutations within $\log_2(n)$ steps. When compared with the table lookup method that is used on existing processors, the ASIP methods can achieve an even higher speedup, about twice as fast as the table lookup method. In addition to the better performance, the ASIP methods provide more flexibility as well; they can perform arbitrary permutations efficiently and the permutations can be dynamically specified, while the table lookup method can only perform a few fixed permutations known at compile time. Moreover, the table lookup method requires large tables for each permutation.

We also compare the implementation complexity in terms of operating system, compiler, ISA, datapath, and control overhead. The entries in boldface indicate key reasons that make a method less desirable. The operating system overhead for context switches in the LdState method and the high compiler complexity in the register pair method are undesirable. The two-length ISA method and the bundled instruction method are less desirable than the 2-way superscalar method because the first two incur control complexity without the performance advantage of being able to execute ALU operations in parallel. Finally, the VLIW method has the same performance advantages as the superscalar method with significantly less control complexity, and hence may be the preferred method. This study allows an ASIP designer to choose a method that comes closest to meeting all his design goals.

Although this paper considers only the BFLY and IBFLY instructions, the six approaches proposed can be applied to other permutation methods, either existing methods or methods that will emerge in the future, to reduce the number of cycles for permutations as long as the cycle time of processors is not increased by the permutation units. For example, the same approaches may be applied to the omega-flip network. In this case, an OMEGA instruction uses a full omega network of six stages to permute 64 bits and a FLIP instruction uses a full flip network of six stages. The latency of a full omega or flip network, however, is longer than that of a butterfly network (or an inverse butterfly network) because

of longer wires between stages.

VII. CONCLUSIONS

In this paper, we examine the basic operations in block ciphers and identify bit permutation as a fundamental operation for the class of symmetric-key block ciphers that is not well supported in existing processor architectures. We also identify two difficulties in implementing very fast arbitrary bit permutations on ASIPs: permutation functional units may have a latency longer than a typical cycle, and a large number of configuration bits are needed to specify an arbitrary permutation. This paper solves both problems and reduces the number of cycles for performing 64-bit permutations to one or two cycles.

To solve the latency problem, we compare the critical paths through different 64-bit permutation functional units with that of a typical 64-bit ALU, and choose the BFLY and IBFLY permutation functional units because they have the shortest latency, shorter than an ALU. The cycle time of a processor will not be impacted by BFLY or IBFLY functional units; if an ALU instruction can be done in one cycle, so can a BFLY or IBFLY instruction. The BFLY permutation unit implements a $\log_2(n)$ -stage butterfly network while the IBFLY permutation unit implements a $\log_2(n)$ -stage inverse butterfly network. By performing a BFLY followed by an IBFLY instruction, any arbitrary n -bit permutation is achievable in at most two cycles on a single-issue processor. In a multi-issue processor, a throughput of one permutation per cycle can be achieved.

To use BFLY or IBFLY instructions and achieve best performance, we must supply $(\log_2(n)/2 + 1)$ operands to a BFLY or IBFLY permutation functional unit in a single cycle. For $n=64$, this is equal to four source operands per cycle. Illustrating with 64-bit processors, we present six solutions for achieving 4-source operations for ASIP architectures, which we call the LdState, register pair, two-length ISA, bundled instruction, superscalar, and VLIW methods. All these solutions take fewer than $\log_2(64)=6$ instructions for an arbitrary 64-bit permutation. All require at most two cycles to achieve any arbitrary n -bit permutation, except LdState and two-length ISA which can take four cycles.

We compare the performance potential and implementation complexity of these solutions. It is interesting that even though ASIPs offer more flexibility in terms of ISA and micro-architecture choices when compared to general-purpose microprocessors, the highest performance solutions, Superscalar and VLIW, are in fact architectural techniques used by general-purpose processors. Hence, our BFLY and IBFLY permutation instructions can be integrated into either application-specific or general-purpose processors.

With these solutions, we can achieve arbitrary bit permutations at a performance level close to data-dependent rotations and with an implementation complexity less than multiplications. Arbitrary bit permutation is potentially a much more powerful operation than data-dependent rotation

or multiplication for symmetric-key cryptographic algorithms. By showing how arbitrary bit permutations can be performed by processors in one or two cycles with relatively simple hardware, we not only enable the acceleration of important existing ciphers but also provide new opportunities for the design of faster and more effective ciphers.

REFERENCES

- [1] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell System Tech. Journal*, Vol. 28, pp. 656-715, October 1949.
- [2] R. B. Lee, R. L. Rivest, M. J. B. Robshaw, Z. J. Shi, and Y. L. Yin, "On Permutation Operations in Cipher Design," *Proceedings of the International Conference on Information Technology (ITCC)*, Vol. 2, pp. 569-577, April 2004.
- [3] Z. J. Shi, *Bit Permutation Instructions: Architecture, Implementation, and Cryptographic Properties*. Ph.D. dissertation, Princeton University, June 2004.
- [4] National Bureau of Standards (NBS), "Data Encryption Standard (DES)," *Federal Information Processing Standards Publication 46-2*, December 1993.
- [5] NIST (National Institute of Standards and Technology), "Advanced Encryption Standard (AES) - FIPS Pub. 197," November 2001.
- [6] C. Burwick et al., "MARS: a Candidate Cipher for AES," *NIST AES proposal*, June 1998.
- [7] R. L. Rivest et al., "The RC6 Block Cipher," *NIST AES proposal*, August 1998.
- [8] Z. J. Shi and R. B. Lee, "Bit Permutation Instructions for Accelerating Software Cryptography," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 138-148, July 2000.
- [9] R. B. Lee, Z. J. Shi and X. Yang, "Efficient Permutation Instructions for Fast Software Cryptography," *IEEE Micro*, Vol. 21, No. 6, pp. 56-69, December 2001.
- [10] X. Yang, M. Vachharajani and R. B. Lee, "Fast Subword Permutation Instructions Based on Butterfly Networks," *Proceedings of Media Processors 1999 IS&T/SPIE Symposium on Electric Imaging: Science and Technology*, pp. 80-86, January 2000.
- [11] X. Yang and R. B. Lee, "Fast Subword Permutation Instructions Using Omega and Flip Network Stages," *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 15-22, September 2000.
- [12] J. P. McGregor and R. B. Lee, "Architectural Enhancements for Fast Subword Permutations with Repetitions in Cryptographic Applications," *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 453-461, September 2001.
- [13] R. B. Lee, Z. J. Shi and X. Yang, "How a processor can permute n bits in $O(1)$ cycles," *Proceedings of Hot Chips 14 - A Symposium on High Performance Chips*, August 2002.
- [14] I. Sutherland, B. Sproull and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann Publishers, 1999.
- [15] Z. J. Shi and R. B. Lee, "Subword Sorting with Versatile Permutation Instructions," *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 234-241, September 2002.
- [16] Z. J. Shi and R. B. Lee, "Implementation Complexity of Bit Permutation Instructions," *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, pp. 879-886, November 2003.
- [17] R. Gonzalez, "XTENXA: a Configurable and Extensible Processor," *IEEE Micro*, Vol. 20, No. 2, pp.60-70, April 2000.
- [18] A. Mizuno et al, "Design Methodology and System for a Configurable Media Embedded Processor Extensible to VLIW Architecture," *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 2-7, September 2003.

Z. Jerry Shi is currently an assistant professor in the Department of Computer Science and Engineering at the University of Connecticut. This work was done while he was a Ph.D. candidate at Princeton. His general research areas are in computer architecture and cryptography. He is especially interested in high performance and secure computer systems. He received his Ph.D. in Electrical Engineering from Princeton University in June 2004.

Xiao Yang is a Ph.D. candidate in the Department of Electrical Engineering at Princeton University. His general research areas are in computer architecture and 3D graphics processing. He is especially interested in high performance and scalable architecture for 3D graphics processing.

Ruby B. Lee is the Forrest G. Hamrick Professor of Engineering and Professor of Electrical Engineering at Princeton University, with an affiliated appointment in the Computer Science Department. She is the director of the Princeton Architecture Laboratory for Multimedia and Security (PALMS). Her current research is in designing security and new media support into core computer architecture, embedded systems and global networked systems, and in architectures resistant to Distributed Denial of Service attacks and Internet-scale epidemics. She is a Fellow of the ACM and a Fellow of the IEEE. She is Associate Editor-in-Chief of *IEEE Micro* and Editorial Board member of *IEEE Security and Privacy*.

Prior to joining the Princeton faculty in 1998, Dr. Lee served as chief architect at Hewlett-Packard, responsible at different times for processor architecture, multimedia architecture and security architecture for e-commerce and extended enterprises. She was a key architect of PA-RISC used in HP workstations and servers, and of multimedia instructions for microprocessors. Dr. Lee also served as Consulting Professor of Electrical Engineering at Stanford University. She has a Ph.D. in Electrical Engineering and a M.S. in Computer Science, both from Stanford University, and an A.B. with distinction from Cornell University, where she was a College Scholar. She has been granted 115 United States and international patents.

TABLE 1: BASIC OPERATIONS IN BLOCK CIPHERS

	DES 3DES	AES	RC5	IDEA	TwoFish	Serpent	RC6	MARS	Kasumi
ALU (add,sub)			√	√	√		√	√	
ALU (logical)	√	√	√	√	√	√	√	√	√
Table lookup	√	√	√		√	√	√	√	√
Multiply				√			√	√	
Shift						√			
Fixed rotation	√	√	√		√	√	√	√	√
Variable rotation			√				√	√	√
Permutation	√				√	√			

TABLE 2: NUMBER OF INSTRUCTIONS AND CYCLES FOR OPERATIONS IN BLOCK CIPHERS

Operation	Number of instructions	Number of cycles
ALU (add, sub)	1	1
ALU (logical)	1	1
Table lookup	1	1 – hundreds
Multiply	1 or 2 ⁺	> 3
Shift	1	1
Fixed rotation	1 – 5	1 – 5
Variable rotation	1 – 5	1 – 5
Bit permutation	> 23 [*]	23 – hundreds

+ Assuming a hardware multiplier is implemented.

* Performing 64-bit permutations with the table lookup method

TABLE 3: MAXIMUM NUMBER OF INSTRUCTIONS AND CYCLES FOR 64-BIT PERMUTATIONS

	GRP	OMFLIP	CROSS	SWPERM / SIEVE	PPERM	This paper BFLY / IBFLY
Maximum number of instructions	6	6	6	11	15	<6 (2 or 4)
Maximum number of cycles	6	6	6	4 [*]	5 [*]	<4 (1 or 2)

* On 4-way superscalar processors.

TABLE 4: LATENCY OF DIFFERENT 64-BIT FUNCTIONAL UNITS

	GRP	OMFLIP	CROSS	6-stage butterfly (or inverse butterfly)	6-stage omega (or flip)	ALU
Latency (in FO4)	>22.5	15.8	27.2	12.0	23.1	16-18

TABLE 5: COMPARISON OF ALTERNATIVES FOR 64-BIT PERMUTATIONS

	LdState	Register Pair	Two-length ISA	Bundled Instr.	Super- scalar	VLIW	log ₂ (n) methods
Number of instructions per permutation	4/2 [*]	2	2	4	4	2 [#]	6
Number of latency cycles (Reg. ports, Issue width)	4/2 [*] (2,1)	2 (4,1)	2 (4,1)	4/2 ⁺ (4,2)	2 (4,2)	2 (4,1 [#])	6 (2,1)
Max throughput (Reg. ports, Issue width)	1 (4, 2)	1 (8, 2)	1 (8, 2)	1 (8, 4)	1 (8, 4)	1 (8, 2 [#])	6 (2, 1)
Parallel ALU instructions	no	no	no	no	yes	yes	no
Speedup of DES	1.09	1.10	1.10	1.10 [§]	1.68	1.68	1
OS complexity	high	none	none	none	none	none	none
Compiler complexity	low	high	low	low	low	low	none
ISA complexity	high	low	mid	low	low	mid	none
Datapath complexity	low	low	low	low	low	low	none
Control complexity	low	mid	mid	mid	mid	low	none

* When the same permutation is repeated, only two instructions (and a latency of two cycles) are required.

Each VLIW instruction is long – consisting of at least two regular instructions.

+ When the fetch/decode logic can process two instructions per cycle, the latency is two cycles.

§ Assumes two instructions fetched per cycle. If only one instruction fetched per cycle, speedup is only 1.03.

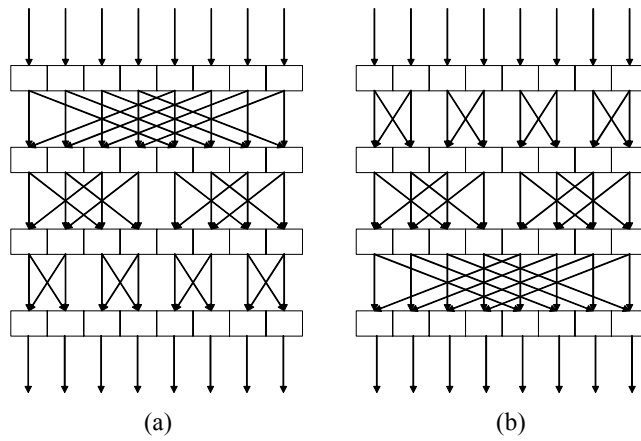
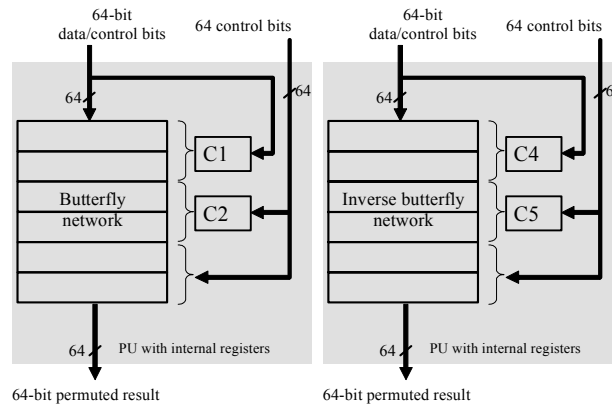
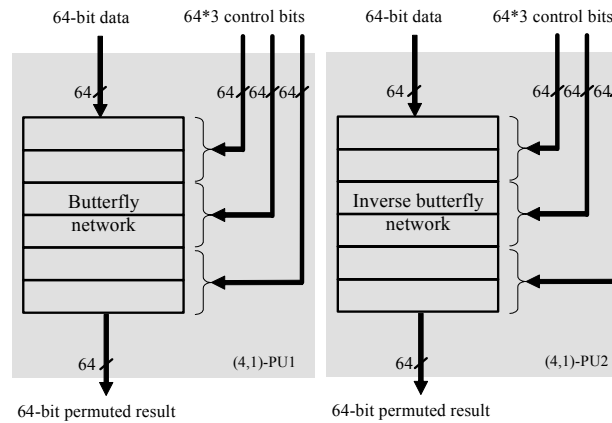


Figure 1 (a) 8-input butterfly network (b) 8-input inverse butterfly network



(a) Permutation units with internal registers



(b) 4-operand 1-result permutation units

Figure 2: Permutation units

- LdState.bfly Rc1, Rc2
- BFLY Rd, Rs, Rc3
- LdState.ibfly Rc1, Rc2
- IBFLY Rd, Rs, Rc3
- (a) LdState**

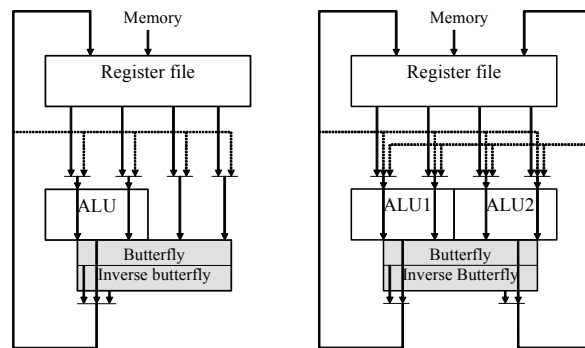
- BFLY Rd, Rs1, Rs2
- IBFLY Rd, Rs1, Rs2
- (b) Register pair**

- BFLY Rd, Rs, Rc1, Rc2, Rc3
- IBFLY Rd, Rs, Rc1, Rc2, Rc3
- (c) Two-length ISA, v1**

- BFLY Rd, Rc1, Rc2, Rc3
- IBFLY Rd, Rc1, Rc2, Rc3
- (d) Two-length ISA, v2**

- Cycle 1: BFLY.ct1 Rc1, Rc2
- BFLY Rd, Rs, Rc3
- Cycle 2: IBFLY.ct1 Rc1, Rc2
- IBFLY Rd, Rs, Rc3
- (e) Bundled instruction, superscalar, and VLIW**

Figure 3: BFLY and IBFLY instruction variants



(a) Datapath with an ALU and permutation units (b) Datapath with two ALUs and permutation units

Figure 4: Processor datapaths with permutation units