Architectural Techniques for Accelerating Subword Permutations With Repetitions

John P. McGregor and Ruby B. Lee, *Fellow, IEEE*

***Abstract*—We propose two new instructions, `swperm` and `sieve`, that can be used to efficiently complete an arbitrary bit-level permutation of an n-bit word with or without repetitions. Permutations with repetitions are rearrangements of an ordered set in which elements may replace other elements in the set; such permutations are useful in cryptographic algorithms. On a four-way superscalar processor, we can complete an arbitrary 64-bit permutation with repetitions of 1-bit subwords in 11 instructions and only four cycles using the two proposed instructions. For subwords of size 4 bits or greater, we can perform an arbitrary permutation with repetitions of a 64-bit register in a single cycle using a single `swperm` instruction. This improves upon previous results by requiring fewer instructions to permute 4-bit or larger subwords packed in a 64-bit register and fewer execution cycles for 1-bit subwords on wide superscalar processors. We also demonstrate that we can accelerate the performance of the popular DES block cipher using the proposed instructions. We obtain a DES performance improvement of at least 55% in constrained embedded environments and an improvement of 71% on a four-way superscalar processor when applying DES as a cryptographic hash function.**

***Index Terms*—Cryptography, encryption, instruction set architecture, permutation, permutation instruction, processor architecture, subword parallelism, subword permutation.**

I. Introduction

As THE POPULARITY of security applications grows, the underlying cryptographic algorithms consume an increasingly large percentage of processor workloads. These applications often include several operations involving 1-bit or multiple-bit register subwords. Many microprocessor instruction set architectures have been extended to include subword-parallel integer arithmetic instructions that improve performance by executing several operations on low-precision data in parallel. Some of these extensions include MAX [7] and MAX-2 [10] for HP PA-RISC, VIS [22] for Sun SPARC, AltiVec [5] for PowerPC, 3 DNow! by AMD [15], MMX [16] for Intel IA-32, and IA-64 multimedia instructions [6], [11].

Before performing subword arithmetic operations, it may be necessary to rearrange the subwords within a single register or between multiple registers using subword permutation instructions. In addition, we can employ subword permutations to efficiently perform transformations such as matrix transposition in multimedia applications [8]. Certain cryptographic algorithms use permutations to achieve diffusion [21], a critical characteristic of a secure cipher, in symmetric-key encryption algorithms such as DES [14], Twofish [19], and Serpent [2]. Some permutations in cryptographic algorithms are not bijections. For instance, the expansion permutation in DES maps some bits in the source datum to multiple destinations in the result datum. We define such rearrangements of an ordered set in which elements can be replicated and possibly replace other elements in the set to be permutations with repetitions. If no information is lost in a permutation with repetitions, the permutation is invertible and therefore can be used in any cryptographic algorithm. Even if information is lost, cryptographic hash functions and encryption algorithms based upon Feistel networks can still employ permutations with repetitions [18].

A. Past Work

Several methods exist for performing permutations in software. In one method, individual bits of the source datum are selected and shifted to their destination locations using a series of bitwise AND, bitwise OR, and shift instructions [12]. For an arbitrary permutation of the bits in an n-bit word, this procedure requires as many as $4n$ instructions. If the architecture includes instructions such as `extract` and `deposit` [9], one can reduce the instruction count of this procedure to $2n$, yet this method is still unacceptably slow.

Alternatively, we can employ lookup tables to perform permutations with repetitions in software [12]. First, we divide the n-bit source datum into x groups of bits; we use each group to index a unique lookup table. The output of a lookup table represents the input group of bits permuted per the desired permutation. The bits of the table output that do not represent any of the input bits are set to zeroes. Therefore, we can combine the outputs of the x lookup tables using $(x-1)$ bitwise OR or bitwise XOR operations to generate the desired permuted n-bit result.

In general, assuming the `extract` instruction is available, we require $(3x-1)$ instructions to complete an n-bit permutation using x lookup tables. Each of the x lookup tables consists of $2^{(n/x)}$ entries, and each entry is n bits in size, so the total size of the tables is $(nx) \cdot 2^{(n/x)}$ bits. This technique is commonly used but is unattractive because the permutations must be statically encoded in the tables at compile-time. Furthermore, the space required to store the lookup tables is large for acceptably small permutation instruction sequences. For example, we need 2 MB of storage to permute a 64-bit datum in 11 instructions using four lookup tables. With eight lookup tables, we require 16 kB of storage and 23 instructions to permute a 64-bit value.

Multiple instruction set architectures have been amended to include instructions for permutations of 8-bit or larger

Manuscript received May 22, 2002; revised October 3, 2002. This work was supported in part by the National Science Foundation under Grant CCR-0105677 and by Hewlett Packard Laboratories.

The authors are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08540 USA (email: mcgregor@ee.princeton.edu; rblee@ee.princeton.edu).

Digital Object Identifier 10.1109/TVLSI.2003.812318

subwords. The `permute` instruction in the MAX-2 extension to PA-RISC supports permutations with and without repetitions of 16-bit subwords in a 64-bit word by statically encoding the permutation function in the instruction [10]. In IA-64, the `mux` instruction supports a small set of permutations of 8-bit subwords in a 64-bit word and supports all permutations of 16-bit subwords in a 64-bit word [6]. Similar to `permute` in MAX-2, the permutation function is statically encoded in the `mux` instruction at compile-time. The `vperm` instruction in the AltiVec extension to the PowerPC instruction set architecture permutes the 8-bit subwords of a 128-bit vector register [5]. This instruction requires three 128-bit register reads and one 128-bit register write, and the permutation function is encoded in one of the vector source registers. None of the permutation instructions in popular popular instruction set architectures (ISAs) efficiently support arbitrary permutations of 4-bit or smaller subwords.

Recently, researchers have proposed several instructions for performing arbitrary, dynamically specified permutations of 1-bit or larger subwords. Using the `pperm` instruction, we can complete an arbitrary permutation of n bits with or without repetitions in $O(\log\ n)$ instructions [12], [20]. The `xbox` instruction performs n-bit permutations in a similar fashion [4]. We can conduct a 64-bit permutation by executing 8 `pperm` or `xbox` or instructions followed by 7 bitwise XOR or OR instructions to combine the results. The `xbox` and `pperm` instructions essentially dynamically configure and invoke an n-by-n crossbar without requiring the processor to maintain any additional state information. Amending an ISA by requiring additional state variables would be undesirable: such changes require explicit operating system (OS) support and increase the complexity of context switches and interrupts. Also, the number of `pperm` or `xbox` instructions that we need to complete an arbitrary permutation does not decrease as subword size increases (and the total number of subwords to permute decreases).

Using the `grp` instruction, we can complete an arbitrary permutation without repetitions of b-bit subwords packed in an n-bit word in $\log_2(n/b)$ instructions [12], [20]. The hardware needed to support the `grp` instruction is expensive, however. The `cross` instruction employs a Benes network to complete an arbitrary permutation without repetitions using $\log_2(n/b)$ instructions [12], [26]. The `omflip` instruction improves upon the `cross` instruction by using more efficient hardware to complete arbitrary permutations without repetitions in the same number of instructions [12], [25]. Although `grp`, `cross`, and `omflip` can perform an arbitrary permutation without repetitions of 1-bit subwords quickly, these instructions cannot efficiently perform permutations with repetitions.

B. Outline

In this paper, we describe two instructions that accelerate the performance of subword permutations with repetitions. Since the development of DES, cryptographers have often avoided permutations of 1-bit subwords because general-purpose microprocessors cannot complete these operations quickly. By adding our proposed instructions to general-purpose ISAs, cryptographers can employ bit permutations with and without repetitions to rapidly achieve a desired level of diffusion in future ciphers. As a result, the proposed instructions could greatly improve the overall throughput of cryptographic algorithms.

In Section II, we discuss the mathematics of permutations and define two new instructions. We demonstrate how to apply these instructions to achieve arbitrary subword permutations with repetitions in Section III. In Section IV, we present the hardware required to implement the two new instructions. In Section V, we analyze the performance of permutations for differently sized subwords, and we evaluate the performance improvement effected by the proposed permutation instructions for a highly popular symmetric-key encryption algorithm. We summarize in Section VI.

II. Permutation Instructions

We propose two new instructions to efficiently support permutations with repetitions of 1-bit or multiple-bit subwords: `swperm` and `sieve` [13]. Using these instructions, we can dynamically specify permutations with repetitions during program execution rather than force the permutations to be statically encoded at compile-time.

A. Permutations With Repetitions

A permutation is a rearrangement of the elements in an ordered set, i.e., a bijective map from a set S to itself [1]. In the context of this paper, such a set is not a collection of all possible n-bit words; we are concerned with ordered sets that are column vectors of b-bit subwords. We define a surjective (i.e., one-to-many) map from an ordered set S to another ordered set D—where the cardinality of S equals that of D—to be a *permutation with repetitions*. In other words, a permutation with repetitions can map an element in the source set S to multiple elements in the destination set D, whereas a permutation without repetitions cannot map an element in S to more than one element in D. For example, if S is the two-subword column vector $(a,b)^T$, there exist two possible permutations of S, $(a,b)^T$ and $(b,a)^T$, but there exist four possible permutations with repetitions of $S: (a,b)^T, (b,a)^T, (a,a)^T, (b,b)^T$. We can encode a permutation with repetitions by specifying the source element in S that is mapped to a particular destination element in D for all of the elements in D. If the permutation is arbitrary, the following expression describes the minimum number of bits needed to encode a permutation with repetitions

$$\sum_{i=1}^{||D||} \log_2 ||S||. \tag{1}$$

In this paper, we examine permutations with repetitions of b-bit subwords packed in an n-bit source register that we write to b-bit subwords of an n-bit destination register. Hence, $||S||$ is equivalent to the number of bits in the source register, n, divided by the subword size, b, and $||D||$ is the number of bits in the destination register, n, divided by the subword size, b. We can rewrite the expression as follows:

$$\sum_{i=1}^{||D||} \log_2 ||S|| = \sum_{i=1}^{\frac{n}{b}} \log_2\left(\frac{n}{b}\right) = \frac{n}{b}\log_2\left(\frac{n}{b}\right). \tag{2}$$

TABLE I
MINIMUM NUMBER OF BITS NEEDED TO SPECIFY AN ARBITRARY PERMUTATION WITH REPETITIONS

Subword size	Number of subwords per 64-bit register	Number of bits to encode a 64-bit permutation with repetitions
32 bits	2 subwords	2 bits
16 bits	4 subwords	8 bits
8 bits	8 subwords	24 bits
4 bits	16 subwords	64 bits
2 bits	32 subwords	160 bits
1 bit	64 subwords	384 bits

We assume that all registers are 64–bits wide; therefore n equals 64. Table I summarizes the minimum number of bits needed to specify an arbitrary 64-bit permutation with repetitions when using subword sizes ranging from 1 32 bits.

RISC instructions typically allow two register reads and one register write per instruction. We wish to design instructions that allow us to dynamically specify permutations at run-time, so we use one of the 64-bit source registers, `rs`, to store the information to be permuted, and we use the other 64-bit source register, `rp`, to store information concerning the permutation function. For subwords of size greater than or equal to 4 bits, we require at most 64 bits of information to specify the entire permutation. Hence, we can describe the entire permutation in a single instruction. Since we can specify 64 more configuration bits with each additional permutation instruction, permutations of 32 2-bit subwords require at least three RISC instructions, and permutations of 64 1-bit subwords require at least six RISC instructions.

In the rest of this paper, we use the term "permutation" to mean a permutation with repetitions.

B. The `swperm` Instruction

The `swperm` instruction permutes the 16 4-bit subwords of a 64-bit source register `rs` according to information stored in a 64-bit source register `rp`. The instruction writes the permuted result to the 64-bit destination register `rd`. The information stored in `rp` fully describes the desired permutation, so one can specify the permutation function dynamically. The instruction format of `swperm` is as follows:

$$\texttt{swperm rd}, \texttt{rs}, \texttt{rp}$$

We designed this instruction to permute subwords of size 4 bits or greater in a single cycle and to expedite permutations of 1-bit and 2-bit subwords.

Fig. 1 illustrates an example operation of `swperm`. In the figure, s_i is the ith 4-bit aligned subword of the source register `rs`. We express the contents of `rp` necessary to complete the example permutation in hexadecimal. The value of the ith 4-bit subword in `rp` indicates which aligned 4-bit subword in the source register should be mapped to the ith 4-bit subword in the destination register.

C. The `sieve` Instruction

We use the `sieve` instruction to "filter" bits from `rs` and then direct the resulting bits into particular destinations in `rd`. More specifically, the instruction directs 1 (or 2 bits) from each 4-bit subword of `rs` to 4 (or 2) possible locations in the corresponding 4-bit subword of `rd`. The `sieve` instruction utilizes a third register, `rp`, to configure the bit filter, whereas the `swperm` instruction operates globally over the 4-bit subwords of `rs`, the `sieve` instruction operates locally within the 4-bit subwords of `rs`. In combination with the `swperm` instruction, we can employ the `sieve` instruction to implement arbitrary permutations of 1-bit or 2-bit subwords. The instruction format for `sieve` is as follows:

$$\texttt{sieve}, \texttt{h}, \texttt{f rd}, \texttt{rs}, \texttt{rp}.$$

The 4-bit function code of `sieve` consists of a 1-bit value, `h` (h), and a 3-bit value, `f` ($f_2 f_1 f_0$)

Fig. 2 illustrates two example operations of the `sieve` instruction on the ith 4-bit subword of `rs`. In the figure, $s_{i,j}$ represents the jth bit of the ith subword of `rs`, and $d_{i,j}$ represents the jth bit of the ith 4-bit subword of `rd`. The `sieve` instruction operates in one of two modes: "1-bit mode" enables permutations of 1-bit subwords, and "2-bit mode" facilitates permutations of 2-bit subwords. In 1-bit mode, the instruction directs one of the 4 bits in the ith subword of `rs` to one of the 4 bits of the ith subword of `rd`; the instruction sets the remaining 3 bits in the ith subword of `rd` to 0. Similarly, in 2-bit mode, `sieve` directs either the leftmost (i.e., most significant) two bits or the rightmost (i.e., least significant) 2 bits of the ith 4-bit subword of `rs` to either the left half or the right half of the ith 4-bit subword of `rd`. The instruction sets the remaining two bits of the ith subword of `rd` to 0.

Bits from `rp` and the function code bit h specify which 1-bit or 2-bit subword that the instruction selects from the ith subword of `rs`. In 1-bit mode, 1 bit from every 4-bit subword of `rs` is selected and passed to `rd`. Hence, there exist 4 possible selection operations per `rs` subword, so we need 2 bits to encode the selection operation for each subword. Since a 64-bit register consists of 16 4-bit subwords, we need a total of 32 bits to encode the selection operations for all 16 subwords. We store these 32 bits in the register `rp`. To minimize the number of memory access instructions that we potentially need to load the bit selection information into registers, we use one 64-bit register to store the 32 bits of selection information for two `sieve` instructions. The function code bit h indicates whether to use the most significant or least significant 2-bit half of each 4-bit `rp` subword to perform the `rs` bit selection. In 2-bit mode, we only require 1 bit (rather than 2 bits) of selection information per 4-bit `rs` subword. Hence, we encode `rp` and h as described above, but the even bits of `rp` are ignored.

Fig. 3 illustrates which bits of `rd` receive bits of `rs` given different values of the three function code bits $f_2 f_1 f_0$. In the figure, the boxes containing 64 blocks represent the 64-bit register `rd`. The gray blocks represent bits that receive bits from `rs`; the white blocks represent the bits of `rd` that we set to zeroes. f_2 indicates whether to use 1-bit or 2-bit mode. Bits $f_1 f_0$ of the function code indicate which bit of each 4-bit `rd` subword receives a selected bit from `rs` in 1-bit mode. For example, when $f_1 f_0 = 00$, only the zeroth bit of each 4-bit `rd` subword receives a bit from `rs`. In 2-bit mode, f_0 is ignored, and f_1 in-

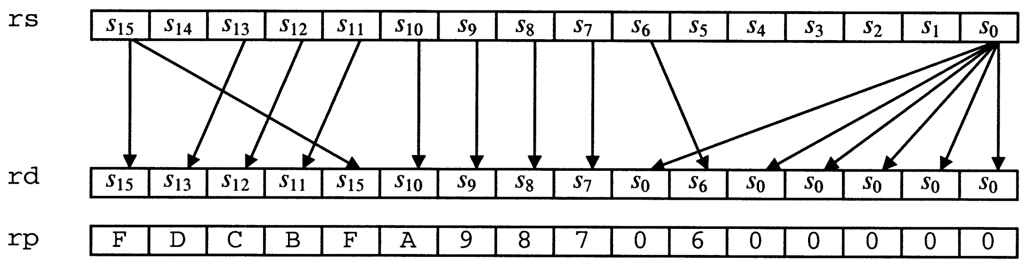
rs
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
rd
s_{15} s_{13} s_{12} s_{11} s_{15} s_{10} s_9 s_8 s_7 s_0 s_6 s_0 s_0 s_0 s_0 s_0
rp
F D C B F A 9 8 7 0 6 0 0 0 0 0

Fig. 1. Example operation of the `swperm` instruction.

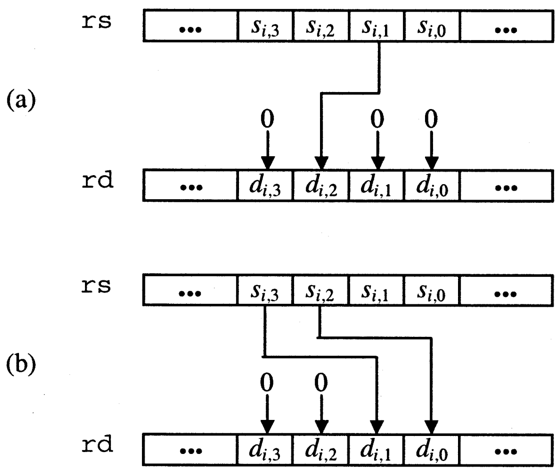
(a)
rs
$s_{i,3}$ $s_{i,2}$ $s_{i,1}$ $s_{i,0}$
0 0 0
rd
$d_{i,3}$ $d_{i,2}$ $d_{i,1}$ $d_{i,0}$
(b)
rs
$s_{i,3}$ $s_{i,2}$ $s_{i,1}$ $s_{i,0}$
0 0
rd
$d_{i,3}$ $d_{i,2}$ $d_{i,1}$ $d_{i,0}$

Fig. 2. Example operation of the `sieve` instruction. (a) In 1-bit mode. (b) In 2-bit mode.

dicates which 2-bit half of each 4-bit `rd` subword receives selected bits from `rs`.

To summarize, the `sieve` instruction selects a single bit or an aligned pair of bits from each of the 16 4-bit subwords of the source register `rs`, but `sieve` only maps these selected bits to the destination register `rd` in one of six possible ways, as shown in Fig. 3. Fig. 4 illustrates a complete example operation of the `sieve` instruction in 1-bit mode. For each of the registers, the least significant bit is located on the right end of the box representing the register. The gray blocks in the `rs` and `rd` boxes indicate which bits are selected and the locations where the selected bits are placed, respectively. The 64-bit values in the `rp` box specify the contents of the configuration register `rp` required to complete the example `sieve` operation. The right 2-bit halves of each 4-bit subword of `rp` possess values of xx, i.e., "don't care", because h equals 1.

III. Applying the Instructions

A. Permuting 1-Bit and 2-Bit Subwords

Using `swperm` and `sieve`, we can complete an arbitrary permutation of 64 1-bit subwords with 11 instructions as shown in Fig. 5(a). We can perform an arbitrary permutation of 32 2-bit subwords with five instructions as shown in Fig. 5(b). In both cases, we initially store the 64-bit value to be permuted in `r1`; upon completion, `r1` will contain the desired permuted result. For 1-bit subwords, `r5` through `r10` store configuration information for the `swperm` and `sieve` instructions, and we use `r1` through `r4` to store intermediate values. For 2-bit subwords, `r1` and `r2` store intermediate values, and `r3` through `r5` store configuration information.

To complete a permutation of 1-bit subwords, we first perform four permutations of 4-bit subwords using `swperm`. Upon completion of these four instructions, the subwords in registers `r1`, `r2`, `r3`, and `r4` will contain the zeroth, first, second, and third bits of the corresponding subwords of the desired permuted result, respectively. For example, after execution of the first `swperm` instruction, 1 of the 4 bits contained in the ith subword of `r2` will ultimately be placed in bit position 1 of the ith subword of the desired permuted result. Likewise, following the execution of the second `swperm` instruction, 1 of the 4 bits stored in the ith subword of `r3` will eventually be placed in bit position 2 of the ith subword of the desired permuted result.

The four `sieve` instructions (in 1-bit mode) move 1 bit from every 4-bit subword of `r1` through `r4` to either the zeroth, first, second or third bit positions of the corresponding subwords in the destination registers. Upon completion of the `sieve` instructions, the desired permuted result is distributed across four 64-bit registers. The 16 bits in the zeroth position of each 4-bit subword in `r1` are the bits that belong in the zeroth position of each subword in the desired result. We set the remaining 48 bits of `r1` to zeroes with the first `sieve` instruction. Similarly, the bits located in the first positions of the 4-bit `r2` subwords, the second positions of the 4-bit `r3` subwords, and the third positions of the 4-bit `r4` subwords belong in the first, second, and third positions of the corresponding subwords of the desired permuted result, respectively. The last three `sieve` instructions set the bits in `r2`, `r3`, and `r4` that do not correspond to bits of the desired result to zeroes. The top four 64-block boxes in Fig. 3 illustrate this distribution of bits in `r4`, `r3`, `r2`, and `r1`. We collect the results of the four `sieve` instructions into a single register by performing 3 bitwise XOR (or bitwise OR) operations. Following the completion of the `xor` instructions, `r1` will contain the 64-bit permuted result.

To permute 32 2-bit subwords packed into a 64-bit register, we use the same method but fewer instructions, as shown in Fig. 5(b). The last two rows in Fig. 3 show how the 64-bits of the desired permuted result are distributed over the two registers `r2` and `r1` after the `sieve` instructions complete. We can combine these two registers into the final 64-bit permuted result by performing a single `xor` (or a single `or`) instruction.

We assume that the registers used to store configuration information are loaded with the appropriate data prior to the execution of these code segments. This preloading may require 6 or 3

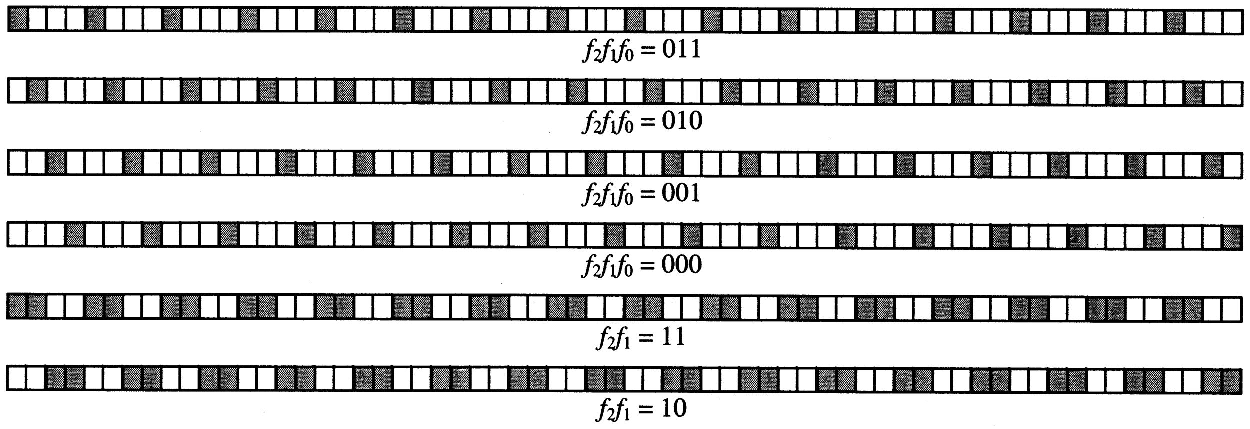

Fig. 3. Effect of `sieve` function code bits on `rd`.

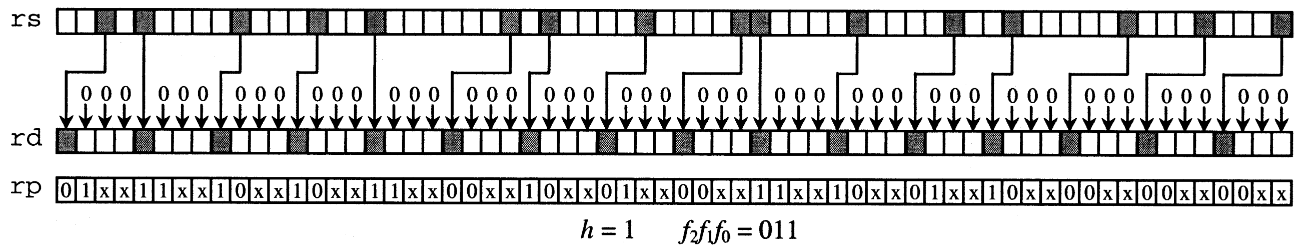

Fig. 4. Complete example operation of `sieve`.

```
swperm        r2,r1,r5      swperm        r2,r1,r3
swperm        r3,r1,r6      swperm        r1,r1,r4
swperm        r4,r1,r7      sieve,0,100   r1,r1,r5
swperm        r1,r1,r8      sieve,1,110   r2,r2,r5
sieve,0,000   r1,r1,r9      xor           r1,r1,r2
sieve,1,001   r2,r2,r9
sieve,0,010   r3,r3,r10
sieve,1,011   r4,r4,r10
xor           r1,r1,r2
xor           r3,r3,r4
xor           r1,r1,r3
```

Fig. 5. Assembly code for performing 64-bit permutations. (a) For 1-bit subwords. (b) For 2-bit subwords.

memory load instructions for permutations of 1-bit or 2-bit subwords, respectively. Cryptographic algorithms often employ the same fixed permutation in every encryption or hash round, however. One can usually perform a round without spilling any registers to memory, so one could load the 6 or 3 permutation configuration values into general-purpose registers once before the execution of thousands of rounds required to encipher or hash kilobytes of data. As a result, the amortized cost of the loads would be negligible. Alternatively, these configuration registers may be intermediate encryption or hash results; therefore, no memory loads would be required.

B. Permuting 4-Bit or Larger Subwords

We can perform a permutation of 4-bit or larger subwords using a single `swperm` instruction. An example of a 64-bit permutation of 4-bit subwords is illustrated in Fig. 1. Given a register `r1` that stores a 64-bit value to be permuted and a 64-bit register `r2` that contains the configuration information necessary to conduct the permutation, the following instruction completes a permutation of 4-bit subwords in a single cycle:

$$\texttt{swperm r1,r1,r2}.$$

The `swperm` instruction stores the desired permuted result in `r1`. One can also complete 64-bit permutations of 8-bit, 16-bit, and 32-bit subwords by executing a single `swperm` instruction. We can divide 8-bit or larger subwords into 4-bit subwords, and it is easy to translate a permutation encoding for 8-bit or larger subwords into a permutation encoding usable by `swperm` for 4-bit subwords.

C. Configuration Information Generation

We describe an efficient and simple algorithm that runs in $O(n)$ time, where n is the number of bits in a register, which produces the configuration information necessary to complete an arbitrary 64-bit permutation. Choosing the appropriate instructions to use, as described above, is a trivial operation that only depends on the subword size. Generating the configuration registers for these instructions is a more complicated process, however. We present source code that produces the permutation configuration information when provided with a simple description of the desired permutation.

The C function `GenPermInfo`, displayed in Fig. 6, generates the `rp` values for the `sieve` and `swperm` instructions involved in a 64-bit permutation. In Fig. 6, `i64` is a type declaration for a 64-bit unsigned integer (i.e., `unsigned long long`). The function accepts three inputs: `sigma`, `sigma_size`, and `inverse`. `sigma` is an array of integers with `sigma_size` elements; `sigma_size` must be a power

```
void GenPermInfo (i64 sigma[], i64 sigma_size,
                  i64 swperm_rp[], i64 sieve_rp[],
                  i64 inverse) {
  i64 j,k,limit,subword_size,sigma2[64];
  subword_size = 64/sigma_size;
  for (j=0;j<4;j++) swperm_rp[j]=sieve_rp[j>>1]=0;
  if (inverse==1) {
    for (j=0;j<sigma_size;j++) sigma2[sigma[j]]=j;
    sigma = sigma2; }
  if (subword_size == 1) {
    /* For permutations of 64 1-bit subwords */
    for (j=0;j<64;j++) {
      swperm_rp[j&0x3]     |= (sigma[j]>>2)
        << (j&0x3C);
      sieve_rp[(j>>1)&0x1] |= (sigma[j]&0x3)
        << ((j&0x3C)+((j&0x1)<<1)); } }
  else if (subword_size == 2) {
    /* For permutations of 32 2-bit subwords */
    for (j=0;j<32;j++) {
      swperm_rp[j&0x1] |= (sigma[j]>>1)
        << ((j&0x1E)<<1);
      sieve_rp[0]      |= (sigma[j]&1)
        << (1+(j<<1)); } }
  else /* (subword_size >= 4) */ {
    /* For permutations of 16 4-bit, 8 8-bit,
       4 16-bit, or 2 32-bit subwords  */
    limit = subword_size/4;
    for (j=0;j<sigma_size;j++)
      for (k=0;k<limit;k++)
        swperm_rp[0] |= (sigma[j]*limit+k)
          << ((j*limit+k)<<2); } }
```

Fig. 6. C source code for configuration data generation.

of 2 and less than 128. The contents of the array, which are specified by the programmer, represent the desired permutation of a 64-bit register. The array element `sigma[i]` indicates which subword from the source register should be directed to the `i`th subword in the permuted result. The number of subwords therefore equals `sigma_size`, and the size of each subword (in bits) equals 64 divided by `sigma_size`.

In some situations, it may be desirable to generate configuration information required by `swperm` and `sieve` to perform the inverse of a given permutation. If the value of `inverse` is 1 and the permutation specified by `sigma[]` is a bijection (and therefore the permutation is invertible), `GenPermInfo` produces the configuration information required to conduct the inverse of the permutation specified by `sigma[]`. The algorithm generates this information by first quickly computing the inverse of the provided permutation. Then, `GenPermInfo` produces configuration information for the inverse permutation by performing the same procedure used to generate configuration information for a regular (i.e., uninverted) permutation.

`GenPermInfo` outputs two integer arrays: `swperm_rp[]` and `sieve_rp[]`. Upon completion of the `GenPermInfo` routine, these two arrays contain the appropriate values of the `rp` registers required by the `sieve` and `swperm` instruction(s) to complete the desired permutation. The algorithm operates by simply extracting bits from the elements of `sigma[]` and placing them in prespecified destination locations in `swperm_rp[]` and `sieve_rp[]`. For instance, suppose we wish to generate configuration information for a permutation of 64 1-bit subwords. In this case, `sigma_size` equals 64. Each element of `sigma[]` is an integer between 0 and 63, inclusive, so we require 6 bits to encode each element. `GenPermInfo` extracts the two least significant bits from all of the elements of `sigma[]` and writes those bits to appropriate locations in `sieve_rp[0]` and `sieve_rp[1]`. The algorithm also extracts the four most significant bits from each 6-bit element of `sigma[]` and places those bits in certain locations in the elements of `swperm_rp[]`. Observe that `sigma_size` and `inverse` are the only input variables upon which the destination locations of the bits extracted from `sigma[]` depend.

Performing a bijective permutation using `swperm` and `sieve` requires the same number of instructions as completing its inverse using these instructions. Therefore, the size of the `GenPermInfo` output is independent of the value of `inverse`. For 64-bit permutations of 1-bit subwords, `GenPermInfo` outputs six 64-bit `rp` values for 4 `swperm` and 4 `sieve` instructions. For 2-bit subwords, `GenPermInfo` outputs three 64-bit `rp` values for 2 `swperm` and 2 `sieve` instructions. In addition, if the subword size is 4 bits or greater, the function generates a single 64-bit `rp` value for a single `swperm` instruction.

Inspection of the function reveals that the maximum number of steps is a constant multiplied by the number of bits in a register, n. Hence, the running time of the algorithm is $O(n)$. For the `sieve` and `swperm` instructions presented in this paper, $n = 64$.

D. Permuting Large Values

The techniques presented above involve arbitrary permutations of a 64-bit word with 1-bit or larger subwords. It may be desirable, however, to complete an arbitrary permutation of 128-bit, 256-bit, or larger words that are distributed across multiple 64-bit registers. We describe a method of applying the `swperm` and `sieve` instructions to complete such arbitrary permutations of large words. Let m be the size of the large word in bits. Let m be a multiple of 64, and $x = m/64$. Therefore, we have x 64-bit blocks in the initial large word and x 64-bit blocks in the destination (permuted) large word. Each of the x blocks of the initial word can contribute 0 to 64 bits to each of the x blocks of the destination word.

We perform a large word permutation as follows. For each block in the initial word, we perform x 64-bit permutations, one for each block in the destination word. After each 64-bit permutation, we perform a bitwise AND operation on the 64-bit permuted result and a 64-bit mask. The mask corresponding to a particular 64-bit initial block and 64-bit destination block pair contains a 1 in bit position i if and only if a bit from the initial block should be mapped to bit position i in the destination block. Since we require one mask for each 64-bit initial block and 64-bit destination block pair, at most x^2 unique masks will be required to conduct the permutation. Upon completing all of the 64-bit permutations and masking operations for a single 64-bit destination block, the x 64-bit results are collected into a 64-bit destination block by performing $(x - 1)$ bitwise XOR operations.

We present a block diagram that conceptually illustrates the operations needed to complete an arbitrary permutation with repetitions of a 128-bit word in Fig. 7. In the figure, the **64-bit Perm** objects include the instructions required to complete an

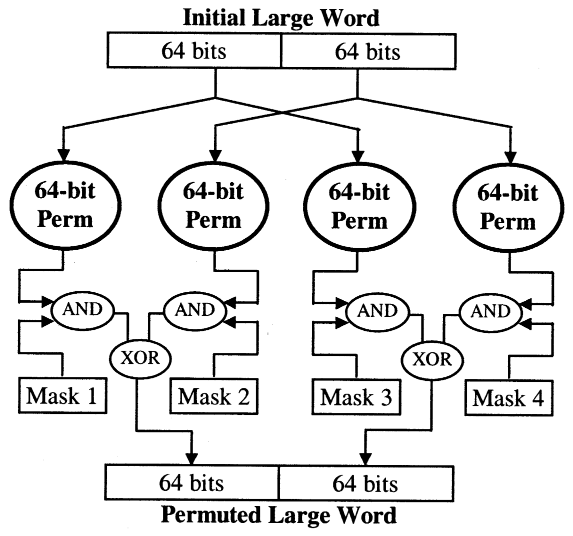

Fig. 7. Permutation with repetitions of a 128-bit word.

arbitrary permutation of a 64-bit word. Depending on the size of the subwords, this code sequence may consist of 11, 5, or 1 instruction(s), as described above. We assume that the initial large word, the masks, and the configuration information for the `sieve` and `swperm` instructions have been previously loaded into registers. Hence, if y is the number of instructions needed to perform an arbitrary permutation of a 64-bit word, the total number of instructions required to complete a permutation of a 128-bit word is $4y + 6$. In general, for a large word of size $m \geq 128$, the maximum number of instructions required to complete an arbitrary permutation is $x^2(y+1)+x(x-1) = yx^2+2x^2-x$. When using 4-bit subwords, permutations of 128-bit and 256-bit words require at most 10 instructions and 44 instructions, respectively. To permute 128 1-bit subwords stored in two 64-bit registers, we require 50 instructions.

IV. Hardware Implementation

We now describe the CMOS hardware implementation for the `swperm` and `sieve` instructions. First, we present the selection unit, which enables the execution of the `swperm` instruction. We can implement the selection unit by building a 4-bit 16-to-1 multiplexer for every 4-bit subword in `rd`. Such a design is extremely expensive in hardware, however. Using a reduced crossbar, we can greatly decrease the transistor and wire cost. The reduced crossbar only requires 1 decoder for every 16 intersections between `rs` and `rd` tracks as opposed to 1 decoder for each intersection in a full crossbar.

A high-level representation of the reduced crossbar is illustrated in Fig. 8(a). s_i is the ith 4-bit subword of `rs`, d_j is the jth 4-bit subword of `rd`, and p_j is the jth 4-bit subword of `rp`. A rectangle represents a single cell, and we present an example cell in Fig. 8(b). Each cell consists of a 4-input AND gate, 4 n-type transistors, and 0, 1, 2, 3, or 4 inverters. Recall that the `swperm` instruction directs the s_i to d_j if and only if p_j equals i. In the example cell, the leftmost and bottommost wires are the most significant bits of the subwords. From inspecting the negation bubbles on the inputs to the AND gate, we know that $i = 5$ in

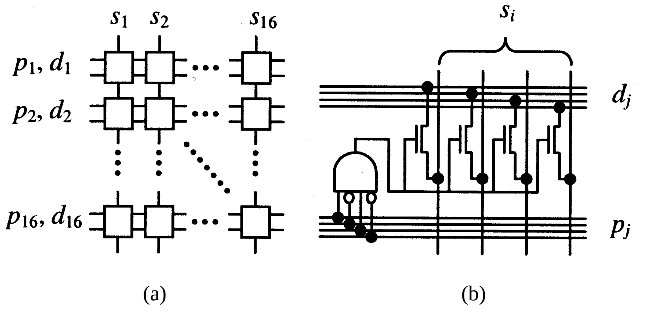

Fig. 8. Hardware implementation of the Selection Unit. (a) High-level organization. (b) Example of a selection unit cell.

Fig. 8(b). Hence, if $p_j = 5$, only the fifth 4-bit subword, s_5, is enabled onto d_j. The other 15 4-bit subwords from `rs` similarly connected to d_j are not enabled onto d_j when p_j equals 5.

We now discuss the hardware cost of this implementation in terms of transistor and track counts. Since we need 16 cells for each of the 16 4-bit subwords of `rd`, the total number of cells in the reduced crossbar is $16 \cdot 16 = 256$. On average, there are two negation bubbles on the inputs to the AND gate per cell, so the average number of transistors per cell is 16. These 16 transistors include eight transistors to implement a 4-input AND gate, four transistors to implement two inverters, and four n-type transistors controlled by the output of the AND gate. The reduced crossbar consists of 256 cells, so the total transistor count is 4096. Note that this count does not include any buffers that we may potentially need to drive the long wires.

We define a track to be a wire routing lane that is reserved for connections between different cells. The number of vertical tracks is roughly the number of bits in `rs`, 64, and the number of horizontal tracks is the number of bits in `rd` plus the number of bits in `rp`, 128. The critical path latency of this circuit is the time needed for a signal to traverse two long wires (that each span the width of 16 selection unit cells) plus the logic delay through a single selection unit cell. This is at most the sum of the propagation delays of two long wires, a 4-input AND gate, an inverter, and an n-type transistor. Assuming the delays through the wires are not extremely high, the selection unit can complete an `swperm` instruction in a single cycle. In a deeply pipelined processor, however, the propagation delays through wires could force multiple-cycle execution of `swperm` instructions.

We present a block diagram of the filter unit, which supports the `sieve` instruction, in Fig. 9(a). Each rectangle represents a single 4-bit slice, and we can implement a 4-bit slice with four 1-bit 5-to-1 multiplexers. Each of these multiplexers simply select the bit value “0” or 1 of 4 input bits from a 4-bit subword of `rs`; the multiplexer output is directed to a single bit in the corresponding 4-bit subword of `rd`. Using the 4-bit slice structure illustrated in Fig. 9(b), however, we can reduce the transistor count without increasing the critical path latency by eliminating redundant logic operations. We replicate the slice shown in Fig. 9(b) 16 times, once for each 4-bit subword in `rd`. The variable $s_{i,j}$ represents the jth bit of the ith subword of `rs`; the variable $d_{i,j}$ represents the jth bit of the ith subword of `rd`. Each 4-bit slice requires two 1-bit 2-to-1 multiplexers and four 1-bit 4-to-1 multiplexers. In addition, the ith subword slice includes a set of signals to control these multiplexers: A_i, B_i, C_i,

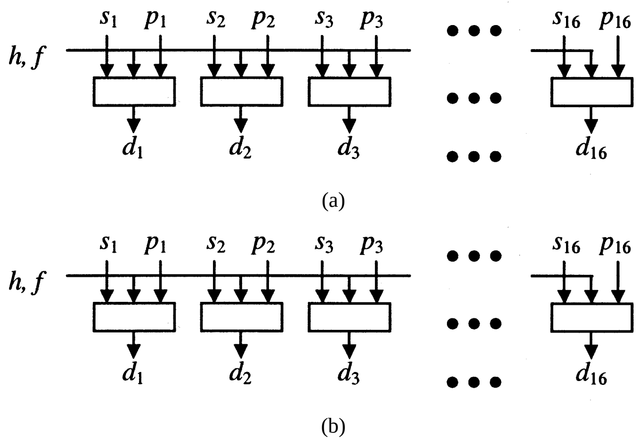
s_1 p_1 s_2 p_2 s_3 p_3 s_{16} p_{16}
h, f
d_1 d_2 d_3 d_{16}
(a)
(b)

Fig. 9. Hardware implementation of the filter unit. (a) High-level organization. (b) Structure of a 4-bit slice.

$$\boldsymbol{A_i} = \boldsymbol{B_i} = (h \cdot p_{4i+3}) + (\neg h \cdot p_{4i+1})$$
$$\boldsymbol{C_{i,1}} = f_1 \cdot (f_2 + f_0) \qquad \boldsymbol{D_{i,1}} = f_1 \cdot (f_2 + \neg f_0)$$
$$\boldsymbol{E_{i,1}} = \neg f_1 \cdot (f_2 + f_0) \qquad \boldsymbol{F_{i,1}} = \neg f_1 \cdot (f_2 + \neg f_0)$$
$$\boldsymbol{C_{i,0}} = \boldsymbol{E_{i,0}} = (\neg f_2 \cdot ((h \cdot p_{4i+2}) + (\neg h \cdot p_{4i}))) + f_2$$
$$\boldsymbol{D_{i,0}} = \boldsymbol{F_{i,0}} = (\neg f_2 \cdot ((h \cdot p_{4i+2}) + (\neg h \cdot p_{4i})))$$

Fig. 10. Control signals in the filter unit.

D_i, E_i, and F_i. We define these signals in Fig. 10, where p_k is the kth bit of `rp`, and h, f_2, f_1, and f_0 are function code bits.

We can implement a 2-to-1 multiplexer using four transistors, and we can implement a 4-to-1 multiplexer using only seven transistors each since the two lowest bit inputs are always zeroes. Using buffers to reduce the fan-out of the function code bits and logic optimization techniques to reduce the transistor count, each 4-bit subword slice requires 116 transistors. The total number of transistors required for the 16 4-bit subword slices of the filter unit is 1856. Nearly all of the data and control for each 4-bit subword slice in the filter unit is local, so we do not require many long vertical or horizontal tracks. We only need four horizontal tracks for the 4 `sieve` function code bits. The critical path latency in the filter unit is at most the sum of the propagation delays through a horizontal wire (that spans the width of 16 4-bit slices), a 2-to-1 multiplexer, a 4-to-1 multiplexer, and the logic required to compute A_i. Therefore, it is highly likely that the filter unit can complete the execution of a `sieve` instruction in a single cycle.

The total number of transistors needed to implement a permutation unit, which consists of a selection unit and a filter unit, is 5952. This transistor count is of the same order of magnitude as that required to construct a simple 64-bit CMOS ripple-carry adder [23]. We compare the hardware cost of the permutation unit to past work in Table II. Due to the imprecision of the track metric, we compare numbers of tracks using O-notation in terms of the number of bits in a register, n. When considering both transistor count and wire area, it appears that the permutation unit is nearly as efficient as a very large scale integration (VLSI) implementation of the `omflip` instruction. The permutation unit requires nearly twice as many transistors as an `omflip` implementation, but it potentially consumes only half as much wire area due to constants hidden by the O-notation. The permutation unit also requires significantly fewer transistors and tracks than a crossbar network.

TABLE II
HARDWARE COST COMPARISON

Implementation	Horizontal tracks	Vertical tracks	Transistor count
Permutation Unit (`swperm/sieve`)	$O(n)$	$O(n)$	~6000
Omega-flip network (`omflip`) [25]	$O(n)$	$O(n)$	~3100
Crossbar network [25]	$O(n)$	$O(n \log n)$	> 73,728

TABLE III
PERFORMANCE OF 64-BIT PERMUTATIONS USING `sieve` AND `swperm`

Subword size	Max. # of instructions	Minimum # of cycles			Max. # of registers
		1-issue	2-issue	4-issue	
32 bits	1	1	1	1	2
16 bits	1	1	1	1	2
8 bits	1	1	1	1	2
4 bits	1	1	1	1	2
2 bits	5	5	3	3	5
1 bit	11	11	6	4	10

V. PERFORMANCE

A. 64-Bit Permutation Performance

Table III summarizes the number of instructions, cycles, and registers required to complete arbitrary permutations of different-sized subwords packed into a 64-bit register. For subword sizes of 4 bits or larger, we only need one `swperm` instruction and two registers to complete an arbitrary 64-bit permutation with repetitions. Using both `sieve` and `swperm`, arbitrary 64-bit permutations with repetitions of 2-bit and 1-bit subwords require 5 and 11 instructions, respectively. In past work, Yang and Lee demonstrated that the `omflip` instruction could be used to complete 64-bit permutations without repetitions using five and six instructions, respectively [25]. These `omflip` instruction sequences must be executed serially, however. Therefore, even on an ultra-wide superscalar processor, a 64-bit permutation of 1-bit subwords without repetitions requires six cycles using `omflip` instructions.

Superscalar execution can accelerate permutations that employ `sieve` and `swperm`, however. True data dependencies do not exist between any of the `swperm` instructions or between any of the `sieve` instructions listed in Fig. 5. Hence, a multiple-issue processor can improve the performance of an arbitrary 64-bit permutation that employs the proposed instructions by executing certain instructions in parallel. On a four-way superscalar processor, we can complete permutations of 1-bit and 2-bit subwords in as few as four and three cycles, respectively. For 1-bit subwords, the 4 `swperm` instructions can be executed in parallel in a single cycle, and the 4 `sieve` instructions can be executed in parallel in the following cycle. The 3 `xor` instructions must be executed in two cycles following the completion of the `sieve` instructions due to data dependencies.

We compare the performance of `sieve` and `swperm` to past work in Table IV. The table lists the number of instructions and

TABLE IV
PERMUTATION PERFORMANCE COMPARISON

Instruction(s) used to perform a 64-bit permutation	Maximum # of instructions per subword size						Number of cycles for 4-way superscalar					
Subword Size	32 bits	16 bits	8 bits	4 bits	2 bits	1 bit	32 bits	16 bits	8 bits	4 bits	2 bits	1 bit
`sieve/swperm`	1	1	1	1	5	11	1	1	1	1	3	4
`pperm` [12] and `xbox` [4]	15	15	15	15	15	15	5	5	5	5	5	5
`omflip` [25], `cross` [26], and `grp` [20]	1	2	3	4	5	6	1	2	3	4	5	6
Existing ISAs	1	1	≥ 1	23	23	23	1	1	≥ 1	10	10	10

cycles required by the different methods to complete a 64-bit permutation and to write the result to a single 64-bit register. The bit values in the heading of the table indicate the size of the subwords to be permuted within a 64-bit word. We determine the cycle counts using a simulation of a 4-way superscalar processor with four integer execution units and a single load/store unit.

The *Existing ISAs* row indicates the minimum number of instructions in conventional ISAs required to perform a 64-bit permutation using eight lookup tables or existing permutation instructions. Note that instructions in existing ISAs that permute 8-bit subwords are generally limited to performing a small set of predefined permutations. Also, the instruction counts listed for permutations of 4-bit and smaller subwords using existing ISAs are only achievable if the permutation is statically encoded in lookup tables. The cycle counts listed in Table IV were obtained using a perfect data cache model with a single-cycle access latency. If the data cache were small or initially cold, however, table lookup operations could require many additional cycles to complete due to cache misses. Hence, the cycle counts in the *Existing ISAs* row could be much larger in certain scenarios.

Other than `sieve/swperm`, only the `pperm` instruction can efficiently complete bit-level permutations with repetitions. The `omflip`, `cross` and `grp` instructions only perform permutations without repetitions. However, `cross`, `omflip`, and `grp` can be applied to any register size n that is a power of 2, whereas `swperm` and `sieve` are only defined for $n = 64$.

For 64-bit permutations, we observe that `sieve` and `swperm` perform as well as or better than all previously proposed permutation instructions and existing ISAs with the exception of the number of instructions required to complete a 64-bit permutation using 1-bit subwords. Note that the performance improvement provided by `sieve` and `swperm` over existing methods on 2-way and 4-way superscalar processors requires 2 or 4 permutation units, respectively. Methods that employ `cross`, `grp`, and `omflip` only require 1 unit to achieve the cycle counts listed in Table IV.

B. Performance Improvement for DES

We now demonstrate the degree to which our permutation instructions can improve the performance of a highly popular symmetric-key block cipher, the Data Encryption Standard (DES) [14]. A large number of secure communications, banking, and storage protocols employ DES (and its more secure variant, 3DES) to provide services such as data confidentiality and data integrity. We begin with an optimized C implementation of the DES algorithm that is based upon Eric Young's libdes [27]. We compile the implementation for the 64-bit Alpha ISA (augmented with the proposed permutation instructions) using gcc with the -O2 optimization flag. To improve the performance of the block cipher, we apply our permutation instructions to four permutation operations within DES: the initial permutation (IP), the final permutation (FP), the P-box permutation (PP), and the compression permutation (CP). Most software implementations of DES complete these permutations using a series of table lookup operations. We seek to increase performance by replacing these table lookup operations with our permutation instructions.

For processors with small and simple caches, we can achieve a significant speedup for the P-box permutation. In our baseline software implementation, the P-box permutation is built into the lookup tables used to complete operations known as S-box substitutions. Performing the P-box permutation using our proposed instructions allows us to decrease the size of the S-box lookup tables and consequently reduce cache misses. Also, the compression permutation in the round key computation function can consume a large percentage of the total clock cycles involved in a DES operation. By accelerating the compression permutation using the new permutation instructions, we can greatly improve performance in some scenarios, which we describe below. We can also accelerate the performance of the IP and the FP, although these permutations only account for a small percentage of the computation required per DES operation.

We use the SimpleScalar superscalar processor simulator [3] to obtain cycle-accurate performance statistics concerning the execution of DES. We perform simulations for four different processor configurations, which range from a typical embedded processor found in low-power wireless information appliances to a wide superscalar processor used in high-end servers. The four microarchitectural configurations consist of a single-issue processor core with small cache, a 2-issue superscalar processor, a 4-issue superscalar processor, and an 8-issue superscalar processor. For each model, the fetch, decode, and commit widths equal the issue width. Also, the number of ALUs equals the issue width, and we assume that each ALU contains a permutation unit. In Table V, we summarize the memory system parameters used in the SimpleScalar simulations. The L2 latency for the 2-way processor is larger than those of the 4-way and 8-way processors because we model the 2-way superscalar's Level 2 cache (L2) as being off-die Rather than modify the C compiler to identify and utilize our permutation instructions, we strategically insert standard RISC integer ALU instructions that represent our permutation instructions into the DES source code. The DES implementation that uses these special integer ALU

TABLE V
SIMULATION MEMORY PARAMETERS

Parameter	Processor model			
	1-issue	2-issue	4-issue	8-issue
Memory ports	1	1	2	2
L/S queue size	4	16	32	64
L1 data cache	1-way 2 KB	1-way 8 KB	2-way 16 KB	2-way 32 KB
L1 inst. cache	1-way 2 KB	1-way 8 KB	2-way 16 KB	2-way 32 KB
L2 cache	none	128 KB 2-way	256 KB 4-way	256 KB 4-way
L1/L2 latency	1/-	1/20	1/5	1/5
Memory latency	50	50	100	100

instructions maintains the same instruction-level control dependence and data dependence structure as that of a DES implementation that employs the proposed permutation instructions. We carefully choose the special ALU instructions such that the compiler does not eliminate or combine any of those instructions during code optimization. In addition, we modify the SimpleScalar simulator to recognize the special integer ALU instructions and treat them as permutation instructions.

We obtain performance data by simulating the execution of DES for 8 kB input data blocks after allowing the caches sufficient time to warm up. The input size is not a critical simulation parameter, however; we find that once the caches are warm, the performance speedup results are independent of the input data size. The speedups effected by the proposed permutation instructions are presented in Fig. 11. The graph illustrates the speedups associated with each processor configuration for different numbers of input bytes per round key computation. For example, data points associated with the number 32 on the horizontal axis of the figure corresponds to 8 kB inputs in which the round key computation is performed one time for each 32 B block of the input. We use the variable Z to represent the number of input bytes per round key computation.

Although, the simulation results are independent of the total input size, the results are heavily dependent on the value of Z. The round key computation must be performed at least once for each unique key used to complete DES operations. When DES is used for encryption, a single key is often employed to encrypt all input data. As a result, we only need to perform the round key computation once during the encryption of an entire input block. Z is therefore equal to the total input size in this case. However, when DES (or any other block cipher) is used to implement a cryptographic hash function for digital signature and data integrity operations, a different key is often employed for each 8-byte input block, for the key is a function of the 8-byte input block [17], [24]. Hence, we must perform the round key computation once for every 8 B of input, so Z equals 8.

Fig. 11 displays speedup results when we complete the IP, FP, and CP (but not the PP) using the new permutation instructions. We attain speedups of 2.37 and 1.71 when Z equals 8 on a single-issue processor and a four-way superscalar processor, respectively. As the number of input bytes per round key computation increases, the speedups decrease to 1.11 or less, however. This occurs because the relative computational cost of the CP

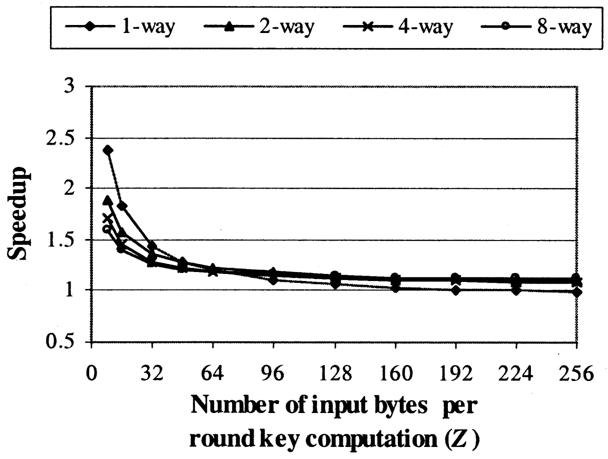

Fig. 11. DES speedups for IP/FP/CP optimization.

decreases as the number of input bytes per round key computation increases. Consequently, the CP performance acceleration caused by the permutation instructions becomes less significant.

We obtain different results when we complete all DES permutations of interest, i.e., the IP, FP, CP, and PP, using `sieve` and `swperm`. For a single-issue processor with a small cache, we achieve a speedup of 3.71 when Z is 8. As the number of input bytes per round key computation increases, the speedup falls to 1.55. The single-issue processor experiences a much larger performance improvement for all values of Z in the IP/FP/CP/PP optimization case than in the IP/FP/CP case due to memory system behavior. When the PP is built into the S-box lookup tables, the amount of memory required to store the tables and intermediate values exceeds the size of the single-issue processor's data cache. As a result, performance suffers due to frequent cache misses. By implementing the PP using the proposed permutation instructions, the number of cache misses experienced by the single-issue processor is greatly reduced, and therefore performance is significantly enhanced.

The wider processors do not suffer many cache misses because their caches easily accommodate the S-box lookup tables. Consequently, reducing the size of the lookup tables by implementing the PP with permutation instructions does not improve performance for any value of Z. We achieve the highest performance for 2-way and wider superscalar processors when we only use the permutation instructions to implement the IP, FP, and CP.

We conclude that we should always employ the proposed permutation instructions to perform the IP, FP, and CP in software implementations of DES. When using DES as a cryptographic hash function, the performance impact of the proposed permutation instructions is substantial: we obtain speedups ranging from 1.59–2.37. Software implementations of DES should only use 1-bit permutation instructions to perform the PP if the target processor contains an extremely small or nonexistent cache, however. This is often true for processors found in smart cards and wireless information appliances. Such processors containing our permutation unit could achieve large speedups for DES encryption without incurring the cost and power consumption associated with the extra memory required by table lookup schemes.

VI. Conclusion

In this paper, we proposed two 64-bit instructions for accelerating the performance of subword permutations with repetitions: `swperm` and `sieve`. Using these two instructions, we can complete 64-bit permutations with repetitions of 4-bit or larger subwords in one instruction. In addition, we can achieve permutations with repetitions of 1-bit and 2-bit subwords using 11 instructions and five instructions, respectively. These instructions are highly parallelizable, and a 4-way superscalar processor can execute these two instruction sequences in four cycles and three cycles, respectively. Furthermore, we can employ the proposed instructions to improve the performance of the popular DES block cipher, especially in constrained environments with small cache memories. We also described efficient hardware that enables the single-cycle execution of `swperm` and `sieve`.

Using these instructions, cryptographers can design ciphers and hash algorithms that obtain a desirable level of diffusion more rapidly. As a result, less encryption rounds may be required to achieve adequate security, and the throughput of encryption algorithms could be significantly improved. Future work includes investigating the degree to which permutations with and without repetitions contribute to the security of a cipher.

References

[1] M. Artin, *Algebra*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[2] E. Biham, R. Anderson, and L. Knudsen, "Serpent: a new block cipher proposal," in *Proc. 5th Int. Workshop Fast Software Encryption*. New York: Springer-Verlag, 1998, pp. 260–271.

[3] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. Wisconsin-Madison Comput. Sci. Dept. , Tech. Rep. 1342, June 1997.

[4] J. Burke, J. McDonald, and T. Austin, "Architectural support for fast symmetric-key cryptography," in *Proc. ASPLOS-IX*, Nov. 2000, pp. 178–189.

[5] K. Diefendorff *et al.*, "AltiVec extension to PowerPC accelerates media processing," *IEEE Micro*, vol. 20, no. 2, pp. 85–95, Mar./Apr. 2000.

[6] *IA-64 Application Developer's Architecture Guide*, Intel Corp., 1999.

[7] R. B. Lee, "Accelerating multimedia with enhanced microprocessors," *IEEE Micro*, vol. 15, Apr. 22–32, 1995.

[8] ——, "Multimedia extensions for general-purpose processors," in *Proc. IEEE Workshop on Signal Processing Syst: Design and Implementation*, Nov. 1997, pp. 9–23.

[9] ——, "Precision architecture," *IEEE Comput.*, vol. 22, pp. 78–91, Jan. 1989.

[10] ——, "Subword parallelism with MAX-2," *IEEE Micro*, vol. 16, pp. 51–59, Aug. 1996.

[11] R. B. Lee, A. M. Fiskiran, and A. Bubshait, "Multimedia instructions in IA-64," in *Proc. Int. Conf. Multimedia Expo*, Aug. 2001.

[12] R. B. Lee, Z. Shi, and X. Yang, "Efficient permutation instructions for fast software cryptography," *IEEE Micro*, vol. 21, pp. 56–69, Dec. 2001.

[13] J. P. McGregor and R. B. Lee, "Architectural enhancements for fast subword permutations with repetitions in cryptographic applications," in *Proc. 2001 Int. Conf. Comput. Design*, Sept. 2001, pp. 453–461.

[14] National Bureau of Standards, Data Encryption Standard, Jan. 1977.

[15] S. Oberman, G. Favor, and F. Weber, "AMD 3 DNow! technology: architecture and implementations," *IEEE Micro*, vol. 19, pp. 37–48, Apr. 1999.

[16] A. Peleg and U. Weiser, "MMX technology extension to the intel architecture," *IEEE Micro*, vol. 16, pp. 42–50, Aug. 1996.

[17] B. Preneel, *Analysis and Design of Cryptographic Hash Functions*. Leuven, Belgium: Katholieke Universiteit, Jan. 1993.

[18] B. Schneier, *Applied Cryptography*, 2nd ed. New York: Wiley, 1996.

[19] B. Schneier *et al.*, *The Twofish Encryption Algorithm: A 128-bit Block Cipher*: Wiley, 1999.

[20] Z. Shi and R. B. Lee, "Bit permutation instructions for accelerating software cryptography," in *Proc. IEEE Int. Conf. Application-Specific Syst., Architectures Processors*, July 2000, pp. 80–86.

[21] D. Stinson, *Cryptography: Theory and Practice*. Boca Raton, FL: CRC, 1995.

[22] M. Tremblay *et al.*, "VIS speeds new media processing," *IEEE Micro*, vol. 16, pp. 10–20, Aug. 1996.

[23] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, 2nd ed. Reading, MA: Addison-Wesley, 1993.

[24] R. S. Winternitz, "Producing one-way hash functions from DES," in *Proc. CRYPTO '83 Advances Cryptology*, 1984, pp. 203–207.

[25] X. Yang and R. B. Lee, "Fast subword permutation instructions using omega and flip network stages," in *Proc. Int. Conf. Comput. Design*, Sept. 2000, pp. 15–22.

[26] X. Yang, M. Vachharajani, and R. Lee, "Fast subword permutation instructions based on butterfly networks," in *Proc. SPIE: Media Processors 2000*, 3970, Jan. 2000, pp. 80–86.

[27] E. Young. (1997, Jan.) libdes. [Online]http://www.shmoo.com/crypto/

John P. McGregor received the B.S. and M.S. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1998, and the M.A. degree in electrical engineering from Princeton University, NJ, in 2000. He is currently working toward the Ph.D. degree at Princeton University.

His research interests include processor architecture, cryptography, and computer security.

Ruby B. Lee (S'75–M'79–SM'01–F'02) received the M.S. degree in computer science and the Ph.D. degree in electrical engineering from Stanford University, Palo Alto, CA, and the A.B. degree from Cornell University, Ithaca, NY.

In 1998, she joined the faculty at Princeton University where she is currently the Forrest G. Hamrick Professor of Engineering and Professor of Electrical Engineering. She also has an appointment in the Department of Computer Science, and is the Director of the Princeton Architecture Laboratory for Multimedia and Security (PALMS). Prior to joining Princeton University, she was a Chief Architect at Hewlett-Packard (HP), responsible for processor architecture, multimedia architecture, and security architecture for e-commerce and extended enterprises. She was a key architect in the initial definition and the evolution of the PA-RISC processor architecture used in HP servers and workstations. As Chief Architect for HP's multimedia architecture team, she led an interdisciplinary team focused on architecture to facilitate pervasive multimedia information processing using general-purpose computers. She introduced innovative multimedia instruction-set architecture (MAX and MAX-2) in microprocessors, resulting in the industry's first real-time, high fidelity MPEG video and audio player, implemented in software on low-end desktop computers. She also co-led an Intel-HP multimedia architectural team for IA-64, recently released in Intel's Itanium microprocessors. While at HP, she also served as a Consulting Professor in the Department of Electrical Engineering at Stanford University. She has been granted over 90 U.S. and international patents, with several patent applications pending. Her current research interests include designing security and new media support into core architecture.

Dr. Lee is a Fellow of ACM, a Member of IS&T, Phi Beta Kappa, and Alpha Lambda Delta. She has served on the editorial boards of IEEE Micro and Spectrum magazines and as Program Chair or Committee Member for several IEEE conferences.