Multimedia Processor-Based Implementation of an Error-Diffusion Halftoning Algorithm Exploiting Subword Parallelism

Jae-Woo Ahn and Wonyong Sung, *Member, IEEE*

***Abstract*—Multimedia processor-based implementation of digital image processing algorithms has become important since several multimedia processors, such as the Intel Pentium MMX, are now available and can replace special-purpose hardware-based systems because of their flexibility. Multimedia processors increase throughput by processing multiple pixels simultaneously using a subword-parallel arithmetic and logic unit architecture. The error-diffusion halftoning algorithm employs feedback of quantized output signals to faithfully convert a multi-level image to a binary image or to one with fewer levels of quantization. This makes it difficult to achieve speedup by utilizing the multimedia extension. In this study, the error-diffusion halftoning algorithm is implemented for a multimedia processor using three methods: single-pixel, single-line, and multiple-line processing. The single-pixel approach is the closest to conventional implementations, but the multimedia extension is used only in the filter kernel. The single-line approach computes multiple pixels in one scan-line simultaneously, but requires a complex algorithm transformation to remove dependencies between pixels. The multiple-line method exploits parallelism by employing a skewed data structure and processing multiple pixels in different scan-lines. The Pentium MMX instruction set is used for quantitative performance evaluation including run-time overheads and misaligned memory accesses. A speedup of more than ten times is achieved compared to the software (integer C) implementation on a conventional processor for the structurally sequential error-diffusion halftoning algorithm.**

***Index Terms*—Error-diffusion halftoning algorithm, multimedia processor, Pentium MMX, subword parallelism.**

I. Introduction

RECENTLY, multimedia processors, which are implemented as embedded processors or general purpose CPUs with a multimedia extension, have been used for several multimedia applications, such as video phones or MPEG video/audio decoders [1]–[8]. The multimedia extension typically supports instructions that can process multiple subword elements or pixels. This form of parallelism is called "subword parallelism," and can be considered a kind of single-instruction-multiple-data (SIMD) computation.

The digital halftoning algorithm is required for representing a multi-level image using binary output devices, such as laser or ink-jet printers and digital copiers. The error-diffusion halftoning (ED) algorithm, which is shown in Fig. 1, is most widely used because it preserves the original image quality faithfully using an error feedback mechanism. However, this algorithm is not good for high-speed processing using parallel processor architectures including multimedia processors, because the computation of an output pixel is dependent on the filtered error of a previous pixel, which prohibits simultaneous computation of multiple output pixels.

Manuscript received December 10, 1998; revised June 12, 2000. This paper was recommended by Associate Editor N. Ranganathan.

The authors are with the School of Electrical Engineering, Seoul National University, Shinlim-dong, Kwanak-gu, Seoul 151-742 Korea (e-mail: ahnjw@dsp.snu.ac.kr; wysung @dsp.snu.ac.kr).

Publisher Item Identifier S 1051-8215(01)01243-5.

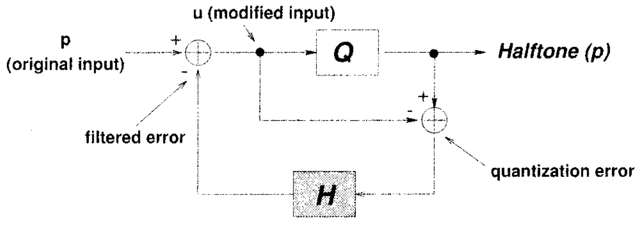


Fig. 1. Generic error-diffusion system for digital halftoning.

In this paper, we consider three approaches for efficient implementation of the ED algorithm using the multimedia extension based on single-pixel, single-line, and multiple-line processing, respectively. The single-pixel method uses multimedia extension instructions only in computing the filter kernel H; thus, it will be called the *parallel filtering* (PF) method. The single-line method computes multiple pixels in one scan-line simultaneously, but it needs an algorithm transformation to break the dependencies between neighboring pixels. The transformation increases the total number of arithmetic operations, but the total execution time can be decreased by employing the multimedia extension parallel computation method. Since this method initially quantizes the input pixel after ignoring the feedback error from the neighboring pixel, it is named the *speculative quantization* (SQ) method. The multiple-line method removes the dependencies between pixels by processing multiple pixels in different scan-lines and employing a skewed data structure. Since this *skewed scan-line* (SS) method only changes the data structure and the order of the processed pixels, it does not need any extra computation, but involves fairly complex data rearrangement operations including matrix transpositions.

For comparison purposes, these three methods are implemented using the Pentium MMX assembly language and their performances are measured using Intel's visual tuning environment—VTune [9]. In this process, run-time overheads, such as memory access penalties, are also considered. The experimental results show that our methods can lead to a speedup of more than ten times over the conventional integer C program. The

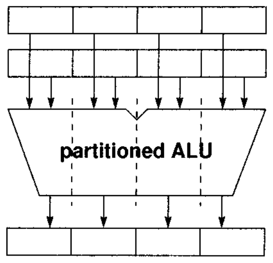


Fig. 2. A datapath for a two-source SIMD operation with a partitioned functional unit.

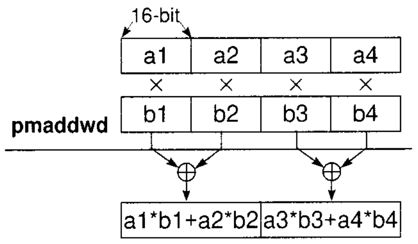


Fig. 3. Operation of the *pmaddwd* instruction in the Pentium MMX technology.

results also manifest the importance of the subword rearranging operations in developing efficient parallelization techniques using subword parallel architectures.

The rest of this paper is organized as follows. Section II introduces the characteristics of multimedia processors. Pentium MMX's specific features are also explained in this section. Section III shows the implementation details of the parallel filtering, or the single-pixel, approach. The speculative quantization and the SS methods are explained in Sections IV and V, respectively. Next, the implementation results are presented and compared in Section VI. The concluding remarks follow in Section VII.

II. Multimedia Processor Architecture

In most multimedia processors, a functional unit can be partitioned into multiple subword units, such as four 16-bit words, to process multiple data using a single instruction. A typical datapath for a two-source SIMD arithmetic or logical instruction is depicted in Fig. 2. In general, each result can be modified by saturation or modulo arithmetic.

Multiplication in multimedia processors has many variations according to the width of the source subwords and the post-processing of the products. For example, the *pmulhw* instruction in the Pentium MMX architecture performs four 16-bit signed multiplications and writes four high-order 16-bits of the 32-bit intermediate results to a destination register. On the other hand, a multiplication instruction in the MIPS-MDMX [6] performs eight 8-bit multiplications to produce eight saturated 8-bit results. In addition, the multiply-and-add feature is supported by either the classical MAC (multiply-and-accumulation) approach as in the MIPS-MDMX or the partial sum-of-products approach as in the Pentium MMX. For example, the *pmaddwd* instruction defined in the MMX technology implements the partial sum-of-products on 16-bit data producing two packed 32-bit sums of the adjacent products, as depicted in Fig. 3. Finally, all multimedia processor architectures define subword rearranging instructions having similar functionality, because it is frequently necessary to rearrange the order of subwords packed into a register.

```
1: for i ← 1 to $L$ do
2:     for j ← 1 to $W$ do
3:         $\Delta_{i,j} = \mathcal{H}\,(e_{i-1,j-1},\ e_{i-1,j},\ e_{i-1,j+1},\ e_{i,j-1})$
4:         $u_{i,j} = p_{i,j} - \Delta_{i,j}$
5:         $q_{i,j} = Q\,(u_{i,j})$
6:         $e_{i,j} = q_{i,j} - u_{i,j}$
7:     end for
8: end for
```

Fig. 4. Sequential pseudocode for the ED algorithm.

In this study, the ED algorithm is implemented with the Pentium MMX architecture. The MMX technology adds 57 instructions to the existing Intel x86 architecture [10], [11]. These instructions perform parallel operations on multiple data elements packed into 64 bits: eight 8-bit, four 16-bit, or two 32-bit fixed-point data elements. Most instructions complete their execution in one cycle, but there are also exceptional multi-latency instructions. For example, the packed multiply instructions have an execution latency of three cycles, although a new multiply instruction can be started every cycle as the multiply unit is pipelined. Eight 64-bit general-purpose registers are logically defined in the MMX architecture. An entire 64-bit data value can be moved between the memory and an MMX register using a 64-bit move instruction, *movq*. A 32-bit data value can also be moved by the *movd* instruction, but in this case, only the lower 32 bits of the source or target MMX register is involved. The direct transfer of a 16-bit data value to or from an MMX register is not allowed and 16-bit move instructions incur a cycle penalty related to instruction decoding. Thus, efficient 16-bit data transfers using 32- or 64-bit move instructions are very important for a Pentium MMX program requiring many discrete memory accesses.

Note that when a 64- or 32-bit load/store accesses a memory location that is not aligned at the 8- or 4-byte boundary, a misalignment penalty of three cycles is incurred. Together with the branch-target-buffer (BTB) miss penalty of 4$\sim$5 cycles from mispredicted branches, the memory misalignment penalty can significantly degrade the run-time performance of a program. As the size of the data grows, cache miss effects should also be considered carefully. The Pentium MMX processor executes up to two integer instructions or MMX instructions in every cycle by sophisticated pairing rules for the "U" and "V" pipes. It is very important to fill the delay slots of the multi-latency instructions and to pair as many instructions as possible.

III. Parallel Filtering Method

A sequential pseudocode for the ED algorithm in Fig. 1 is shown in Fig. 4. In this code, L is the number of scan-lines in an image and W is the number of pixels in a scan-line. A filtered

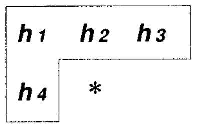


Fig. 5. Floyd–Steinberg filter kernel where "*" is the position of the current pixel.

error for the jth pixel in the ith scan-line is represented as $\Delta_{i,j}$, which is computed by the error filtering routine $\mathcal{H}(\cdot)$. By subtracting the filtered error from a pixel value $p_{i,j}$, the modified input $u_{i,j}$ is obtained, which is then quantized by $Q(\cdot)$. Finally, the quantization error $e_{i,j}$ is obtained by subtracting the modified input $u_{i,j}$ from the quantized pixel value $q_{i,j}$. The error is stored in the error buffer of the ith scan-line.

As the name implies, the parallel filtering method implements the ED algorithm in Fig. 4 using the SIMD-parallelized error filtering routine $\mathcal{H}(\cdot)$, which computes a four-tap FIR filter defined as follows:

$$\begin{aligned}\mathcal{H}(&e_{i-1,j-1},\ e_{i-1,j},\ e_{i-1,j+1},\ e_{i,j-1})\\ &= h_1 \cdot e_{i-1,j-1} + h_2 \cdot e_{i-1,j}\\ &\quad + h_3 \cdot e_{i-1,j+1} + h_4 \cdot e_{i,j-1}\end{aligned} \tag{1}$$

where the four filter coefficients (h_1, h_2, h_3, h_4) are defined by the Floyd–Steinberg filter kernel as depicted in Fig. 5.

To perform an error filtering step, four error values are loaded into an MMX register, and then two partial sums of the filtered error $(S1, S2)$ are obtained using one *pmaddwd* instruction. Note that in our implementation, all data including pixel and error values are represented as 16-bit fixed-point numbers. The final 32-bit filtered error is obtained by summing the two partial sums using a shift and a packed add instruction as depicted in Fig. 6.

In fact, the formation of the four error values in Fig. 6 is not straightforward because three of the four error values, i.e., $\{e1, e2, e3\}$ are in the error buffer of the previous scan-line, and the remaining one, $e4$, is in the error buffer of the current scan-line. We can use 16-bit loads to fetch the errors, but they are very inefficient in a Pentium MMX 32-bit program. Fig. 7 shows how to conduct the formation using three aligned 32-bit loads when the current pixel ("*") is aligned at an 8-byte boundary. Because this relative alignment condition is repeated for every 8-byte group, four pixels are processed in a group with different memory accessing routines for each pixel location. Error filtering using the PF method requires 14 cycles while the integer C program requires about 75 cycles. However, the overall speedup is limited because of the remaining sequential processing routines.

IV. SQ Method

Although concurrent computation of multiple pixels is desired for efficient parallel processing, this is very difficult for the ED algorithm because of the quantizer feedback loop in the error-diffusion process. There have been several studies on the parallel implementation of feedback-based algorithms

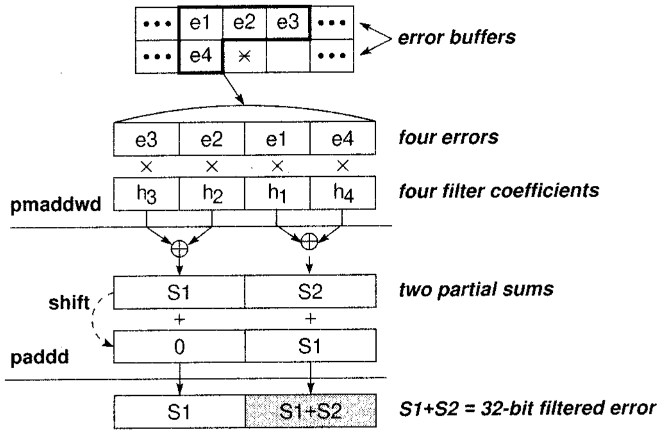


Fig. 6. SIMD-parallelized error filtering routine.

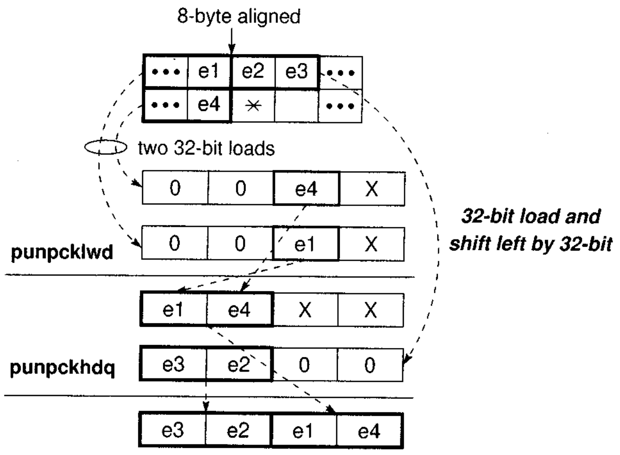


Fig. 7. Filling an MMX register with four error values.

[12]–[14]. The solution presented in [14] is to use the *parallel branch and delayed decision* approach. This scheme is similar to the carry-select adder, where the carry or the feedback term from a previous stage is temporarily ignored, and instead, two output values with two possible carry conditions, zero and one, are evaluated simultaneously by duplicating the hardware. Although this approach reduces the carry propagation dependency time, it requires twice as much computation at higher stages. However, the SIMD function in the MMX architecture can benefit from this parallelization because it can process four 16-bit data values at a time. Once the dependency on the previous pixel is ignored, or the feedback term is set to zero, the neighboring pixels in a scan-line can be processed in parallel. Of course, the quantized pixel values and their quantization errors computed in this way must be checked later to compensate for the effect of the feedback terms that were temporarily ignored.

Our second approach for implementing the ED algorithm is based on the above idea, and is named the SQ method because it temporarily ignores the feedback dependency in the error-diffusion process. In our Pentium MMX implementation of the SQ method, four successive pixels are processed at a time. Fig. 8 depicts the location of the four pixels $(= \{p_{n,m}, p_{n,m+1}, p_{n,m+2}, p_{n,m+3}\})$ and their seven surrounding errors for the error filtering. At every

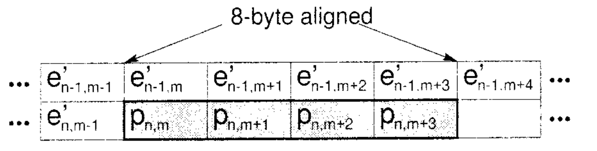


Fig. 8. Location of pixels and modified quantization errors in the SQ method.

iteration of the computation loop, four quantized pixel values $(= \{q_{n,m}, q_{n,m+1}, q_{n,m+2}, q_{n,m+3}\})$ and quantization errors $(= \{e_{n,m}, e_{n,m+1}, e_{n,m+2}, e_{n,m+3}\})$ in the shaded region are computed.

Note that $e'_{n,j}$ in Fig. 8 are modified versions of the original quantization errors that eliminate multiplication in the compensation step [15]. This will be explained later. The relation between the original and the modified errors is

$$e'_{n,j} = e_{n,j}/h_4{}^{(j-1) \bmod 4}, \quad j = 1, 2, \ldots, W. \tag{2}$$

Fig. 9 shows the signal flow graph of the SQ method. In this figure, the over-lined variables represent vectors containing four 16-bit data elements in the form of a packed 64-bit value. The thick arrows represent the flow of the 64-bit vectors. In the region surrounded by a dashed line, four pixels are processed in parallel. However, the compensation step is performed sequentially, pixel by pixel. The thin arrow on which $e'_{n,m+3}$ flows represents a feedback dependency between adjacent groups of four pixels.

The overall computation in Fig. 9 is performed in the following steps. First, four partially filtered errors $\overline{\mathrm{pe}}$ $(= \{pe_{n,m}, pe_{n,m+1}, pe_{n,m+2}, pe_{n,m+3}\})$ are computed in $\mathcal{H}'$ using four *pmaddwd*'s. Here, because the error from the previous group of four pixels, i.e., $e'_{n,m-1}$, is already available at the time of this partial error filtering, we can compute the correct filtered error $pe_{n,m}$. However, the computation of the three remaining ones, $pe_{n,m+1}$, $pe_{n,m+2}$, and $pe_{n,m+3}$, is conducted ignoring the error from the previous pixel. The actual MMX implementation of this partial error filtering is depicted in Fig. 10. In this figure, the computation required for $pe_{n,m}$ and $pe_{n,m+2}$ is shown. In the computation of $pe_{n,m+2}$, we find that one filter coefficient is set to "0" to ignore the error of the previous pixel, which is not available at that time. Note that the filter coefficients are different from the original ones since the modified errors in (2) are used. The six errors required for this error filtering, i.e., $e'_{n-1,m-1}, \ldots, e'_{n-1,m+4}$, are fetched by three 64-bit aligned loads using the relative alignment condition of each pixel location. The location of $p_{n,m}$ is assumed to be aligned at the 8-byte boundary. The one remaining error $e'_{n,m-1}$ is already available in an MMX register from the processing of the previous group of four pixels. Note that by rearranging the seven errors, the two partial sums A and B to be added are computed using two different *pmaddwd*'s. This removes the shift operation for adding two partial sums resulting from a *pmaddwd* instruction as in Fig. 6.

With one correctly filtered error and three partially filtered errors in $\overline{\mathrm{pe}}$, the four input pixels are processed using only SIMD operations. A speculative quantization mask $\overline{c}$ and two scaled pseudo-error vectors, $\overline{d}'$ and $\overline{a}'$, are then obtained as shown in

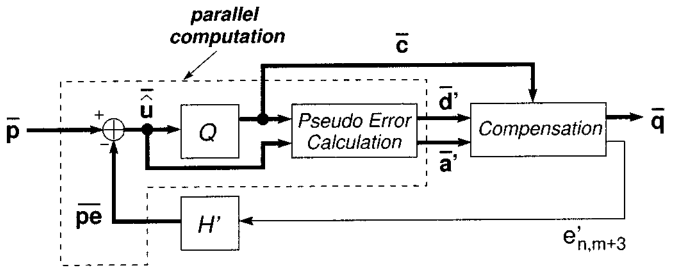


Fig. 9. Signal flow-graph of the SQ method.

Fig. 11. Note that this corresponds to the parallel computation region enclosed by the dashed line in Fig. 9.

The speculative quantization mask $\overline{c}$ contains four 16-bit all-ones or all-zeros values, according to a comparison result for the quantization. The pseudo-error vector $\overline{d}$ contains four quantization errors resulting from the speculatively quantized pixel values $\overline{\hat{q}}$ and the pseudo-modified inputs $\overline{\hat{u}}$. The pseudo-modified inputs are obtained by subtracting the partially filtered errors from the input pixel values. Since the speculation can be incorrect, we must compute another pseudo-error vector $\overline{a}$ with the inverse of the speculative quantized pixel values. The scaled pseudo-error vector $\overline{d}'$ is computed by multiplying each element of $\overline{d}$ by $r_k = (1/h_4{}^{(k-1)})$, where k is one to four, and is used in the later compensation step. The scaled pseudo-error vector for incorrect speculation, i.e., $\overline{a}'$, is obtained in a similar way using $\overline{a}$ and r_k's.

Finally, the speculative quantizations for three pixels need compensation because of the ignored error value of the previous pixel. Note that the first pixel $p_{n,m}$ needs no compensation. The correction is performed sequentially in the order $p_{n,m+1}$, $p_{n,m+2}$, and then $p_{n,m+3}$. The correctness of the speculative quantization for $p_{n,j}$ is determined by testing whether or not the following inequality holds:

$$\begin{aligned} -127.5 < (d_{n,j} + h_4 \cdot e_{n,j-1}) < 127.5 \\ (m+1 \le j \le m+3). \end{aligned} \tag{3}$$

If this inequality is true, the speculation is determined to be correct and the true quantization error $e_{n,j}$ is obtained as follows:

$$e_{n,j} = d_{n,j} + h_4 \cdot e_{n,j-1}, \quad (m+1 \le j \le m+3). \tag{4}$$

According to (3) and (4), we must calculate $h_4 \cdot e_{n,j-1}$, which requires one 16-bit multiplication. However, this multiplication is not required because of the premultiplication of $r_k = (1/h_4{}^{(k-1)})$ to obtain $\overline{d}'$ and $\overline{a}'$ using MMX instructions. If we define $q_{n,j}$ as a true quantized pixel value and use the modified error defined in (2), the true modified error calculation is performed as follows:

$$\begin{aligned} e'_{n,j} &= e_{n,j}/h_4{}^{(j-1) \bmod 4} & (5)\\ &= (q_{n,j} - \hat{u}_{n,j})/h_4{}^{(j-1) \bmod 4} \\ &\quad + e_{n,j-1}/h_4{}^{(j-2) \bmod 4} & (6)\\ &= (q_{n,j} - \hat{u}_{n,j})/h_4{}^{(j-1) \bmod 4} \\ &\quad + e'_{n,j-1}, \quad (m+1 \le j \le m+3). & (7) \end{aligned}$$

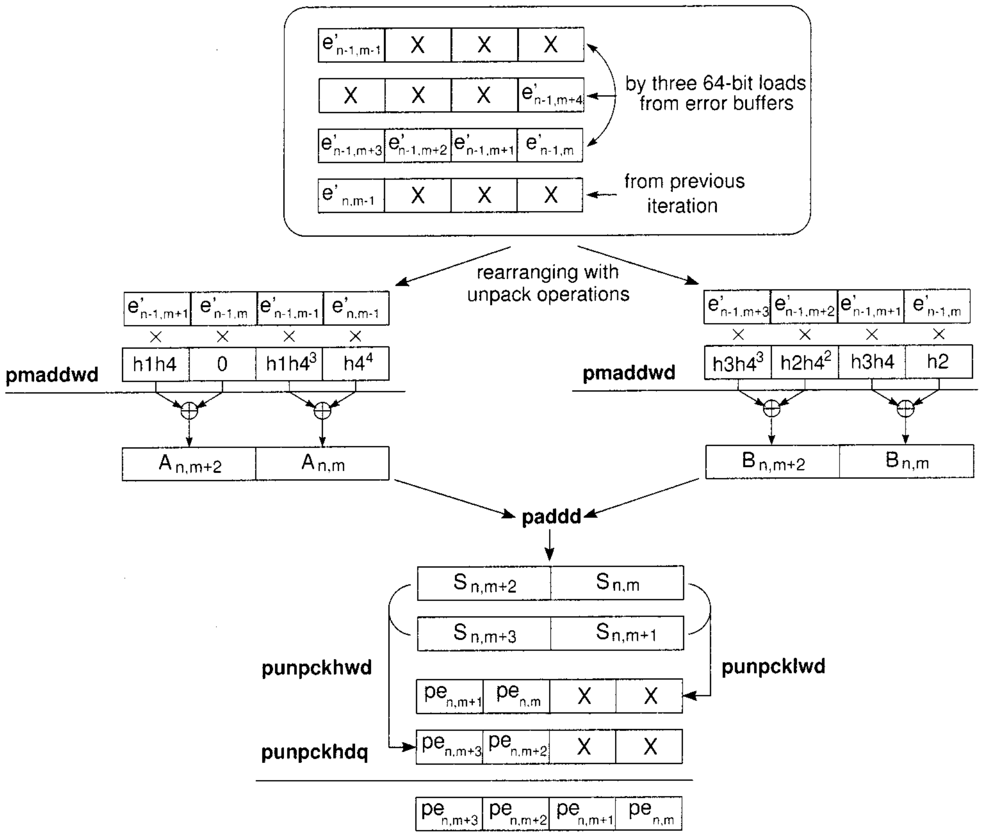

by three 64-bit loads from error buffers
from previous iteration
rearranging with unpack operations
pmaddwd
paddd
punpckhwd
punpcklwd
punpckhdq

Fig. 10. MMX implementation of partial error filtering. $S_{n,j}$ equals $A_{n,j} + B_{n,j}$ and $pe_{n,j}$ is the higher 16 bits of $S_{n,j}(m \le j \le m+3)$. Only the computation flow for $\{S_{n,m+2}, S_{n,m}\}$ is illustrated. $\{S_{n,m+3}, S_{n,m+1}\}$ are computed in a similar way.

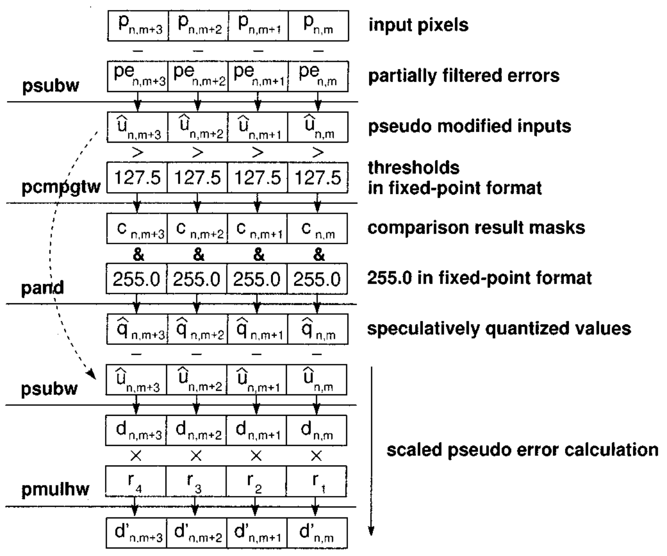

input pixels
partially filtered errors
pseudo modified inputs
thresholds in fixed-point format
comparison result masks
255.0 in fixed-point format
speculatively quantized values
scaled pseudo error calculation
psubw
pcmpgtw
pand
psubw
pmulhw

Fig. 11. MMX implementation of the signal flow-graph enclosed by a dashed line in Fig. 9. Here, $r_k = 1/h_4{}^{(k-1)}$, $(k = 1, 2, 3, 4)$.

In other words, the modified error for $p_{n,j}$, i.e., $e'_{n,j}$, is computed by simply adding the modified error of a previous pixel, i.e., $e'_{n,j-1}$ to $d'_{n,j}$ or $a'_{n,j}$ according to the correctness of the speculation. This simplification explains the purpose

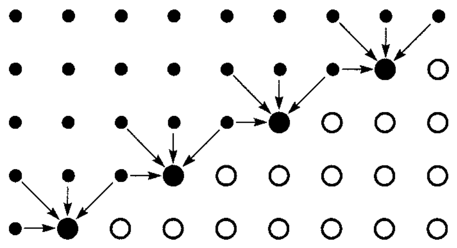


Fig. 12. Dependencies and available parallelism between pixels.

of the modification in (2). It also justifies the MMX-based premultiplication of $d_{n,j}$ and $a_{n,j}$ by r_k since it eliminates integer multiplications in the compensation step.

Note that although this compensation step does not use any pixel-level parallelism, it is actually implemented using only MMX instructions to avoid BTB misses because of conditional branches for correctness checking or prefix penalties from 16-bit instructions for the modified error calculation. Since most of the computation, except for the compensation step, is conducted in parallel, the SQ method requires 26.5% fewer cycles than the PF method.

V. SS Method

The SS method proposed in this section processes independent pixels in different scan-lines simultaneously. When

```
1:  for $i \leftarrow 1$ to $L$ by four do
2:      Prologue();
3:      for $j \leftarrow 7$ to $W$ do
4:          $\overline{\Delta}_{i,j}$ = $\overline{\mathcal{H}}$ ( $\overline{e}_{i-1,j-1}$, $\overline{e}_{i-1,j}$, $\overline{e}_{i-1,j+1}$, $e_{i+3,j-7}$ )
5:          $\overline{u}_{i,j}$ = $\overline{p}_{i,j}$ - $\overline{\Delta}_{i,j}$
6:          $\overline{q}_{i,j}$ = $\overline{Q}$ ( $\overline{u}_{i,j}$ )
7:          $\overline{e}_{i,j}$ = $\overline{q}_{i,j}$ - $\overline{u}_{i,j}$
8:      end for
9:      Epilogue();
10: end for
```

Fig. 13. Pseudocode for the error-diffusion halftoning algorithm with the SS method.

the Floyd–Steinberg kernel is used, the dependency relation between pixels can be depicted by arrows as shown in Fig. 12. In this figure, a small black dot represents a quantized, binary pixel with a corresponding quantization error signal. The large white dots are the unquantized pixels, and the large black dots are the pixels currently being quantized. It is apparent that the four large black dots, which will be called a *skewed group*, can be quantized simultaneously without any algorithm transformation as in the SQ method since the computation is only dependent on previously quantized pixels. Note that this idea can be easily mapped to a systolic-array implementation [16].

A pseudocode for quantizing the four pixels simultaneously is shown in Fig. 13 using vector notation. In this pseudocode, an over-lined variable is equivalent to four pixels in the same skewed group. For example, $\overline{e}_{i,j}$ in line seven is equal to a quadruple of $\{e_{i,j}, e_{i+1,j-2}, e_{i+2,j-4}, e_{i+3,j-6}\}$. In addition, two over-lined functions $\overline{\mathcal{H}}(\cdot)$ and $\overline{Q}(\cdot)$ represent the parallelized version of $\mathcal{H}(\cdot)$ and $Q(\cdot)$ in Fig. 4. When comparing the pseudocode in Fig. 13 with that in Fig. 4, we find that the SS method is actually a vector version of the sequential code except for some data rearranging operations. The SS method can be much faster than the PF or SQ method since it contains no sequential parts and requires no extra computation for parallel transformation. However, it requires the minimization of subword rearranging overhead and memory misalignment penalties.

As illustrated in Fig. 14, our MMX implementation of the SS method processes four skewed groups, corresponding to a total of 16 pixels, in a single iteration to avoid inefficient 16-bit data loads. Note that 16-bit loads are inevitable if only one skewed group is processed separately. After loading four skewed groups using four 32-bit and two 64-bit loads, a 4-by-4 matrix transpose is performed to move four pixels in one skewed group to one MMX register.

One skewed group requires 13 error values that are scattered through five error buffers. Because there can be four different memory alignment conditions according to the starting-point of the current skewed group, four different memory access routines are used to load the 13 errors for each skewed group. Fig. 15 shows the error locations for the first and second skewed groups. The errors in the shaded region are accessed by using nine aligned 32-bit loads.

As in the SQ method, two 32-bit filtered errors can be computed efficiently by using one packed-add operation after performing two *pmaddwd*'s with interleaved errors. In Fig. 16, two 32-bit filtered errors for $p_{1,1}$ and $p_{1,3}$ are computed by two *pmaddwd*'s and one *paddd*. Eight errors required for the two filtering operations, i.e., $\{e1, e2, e3, e4\}$ and $\{e7, e8, e9, e10\}$, are fetched by six 32-bit aligned loads from the locations depicted in Fig. 15(a). Note that one source operand of each *pmaddwd* operation has two 16-bit errors for the computation of $p_{1,1}$ and two for $p_{1,3}$. Another two 32-bit filtered errors for $p_{1,2}$ and $p_{1,4}$ are computed in a similar manner. Four 32-bit filtered errors are converted to four 16-bit values, $\{fe_{1,1}, fe_{1,2}, fe_{1,3}, fe_{1,4}\}$, and combined into one MMX register by using three unpack operations.

Now, we can quantize four pixels in a skewed group and compute the resultant error by utilizing four filtered errors in an MMX register. The four quantization errors for a skewed group must be rescattered to the appropriate locations of the four error buffers before processing the next skewed group, because the next group requires them to perform the error filtering. Since straightforward 32-bit stores for these errors incur one misalignment penalty for every two skewed groups, we use a scheme that stores a 16-bit error using a shift and a 32-bit memory access, as shown in Fig. 17. Here, only the least significant 16 bits of a source MMX register for a store operation correspond to a valid error. In the 32-bit store operations, the other 16-bit value is invalid, but these incorrect values will not be used in the processing of the second skewed group. Since the store operation for the second skewed group is performed at the same aligned location as the first group, the least significant 16-bit value should be equal to the previously stored valid error value. This is achieved by interleaving the errors from the first group and those from the second group to fill one MMX register for the store operation. The resultant two MMX registers and their shifted versions are shown in Fig. 17(b). The errors of the third and fourth skewed groups are stored in a similar way. Note that this error-storing step requires an asymptotically constant overhead per pixel and could be a bottleneck for higher performance with more powerful SIMD architectures.

Finally, after quantizing the four successive skewed groups, 16 quantized pixels are available in four MMX registers in the form of a 4-by-4 matrix. Because each register contains four quantized pixels of each skewed group, the 4-by-4 matrix should be transposed to store four quantized pixels for each scan-line in a group. Actually, the overall processing in this step is the reverse of that depicted in Fig. 14. There are 24 pixels in the prologue and epilogue regions in Fig. 14, which require sequential processing. However, this overhead is relatively small because there are usually more than 1000 pixels in a scan-line.

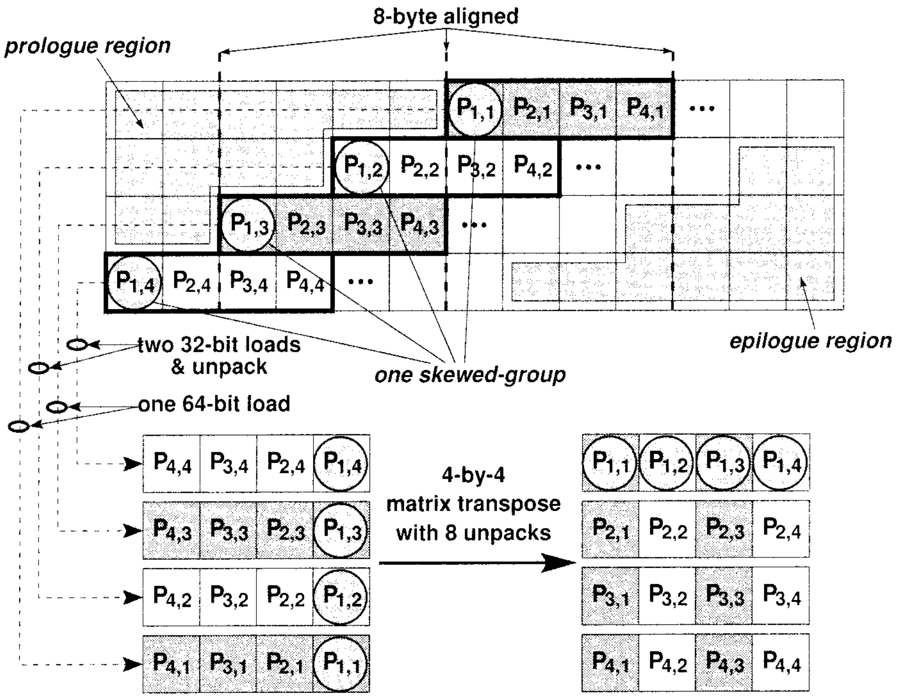
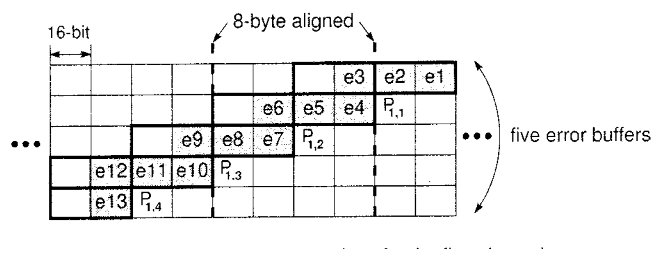
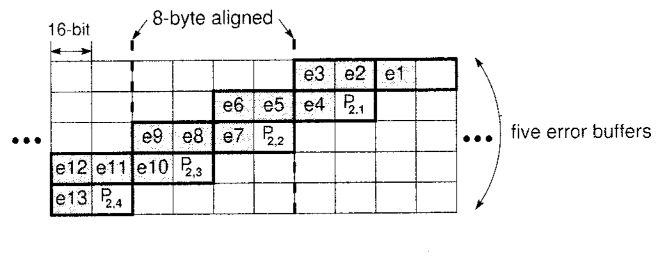


Fig. 14. Sixteen pixels in the four successive skewed groups are loaded into four MMX registers by four 32-bit loads, two 64-bit loads, and one 4-by-4 matrix transposition.



(a)



(b)

Fig. 15. Error locations for the first and second skewed group. (a) Nine 32–bit error-pair locations for the first skewed group. (b) Nine 32–bit error-pair locations for the second skewed group.

VI. Experimental Results and Discussions

A. Performance Evaluation Results

A compiler optimized[1] integer version of a C program and assembly programs based on the PF, SQ, and SS methods were profiled using *VTune*, Intel's visual tuning environment [9]. The intrinsic support of Intel's C-compiler was used only for the functional verification of the developed algorithms. Final optimization was performed by hand-coded assembly to achieve maximum performance. The number of machine cycles required for processing one pixel is used as a primary performance measure in all of the implementations. To include the effects of the run-time penalties because of memory misalignments, cache misses and the BTB misses from mispredicted branches, dynamic as well as static instruction profiling was performed for each implementation. In Table I, the number of cycles per pixel, average cycles per instruction (CPI), pairing ratio, and processing latency are presented to compare the tradeoffs of the four implementations. For performance comparison, the speedup of each Pentium MMX-based implementation over the integer C program is also presented. Note that the results in this table were obtained using 512-by-512-sized images and no cache miss penalties are incurred. The cache effects with larger images will be considered shortly.

By investigating Table I, we can summarize the characteristics of the three SIMD implementations as follows.

1) All of the proposed SIMD implementations operate on 16-bit data elements packed four to a register, yet the speedup is greater than 4.0. This superlinear speedup is mainly because of the Pentium MMX's *pmaddwd* instruction, which performs four multiplications and two additions in three cycles [17]. Note that the integer multiplication instruction used in the C program requires ten cycles.
2) The PF method suffers from run-time cycle penalties because of conditional branches for quantization and inevitable misaligned memory accesses. In contrast, there are neither conditional branches nor misaligned memory accesses in the loop kernels of the SQ and SS methods, resulting in zero run-time cycle penalty.

[1]Visual C++ 5.0 with the highest speed-optimization level was used for compilation.

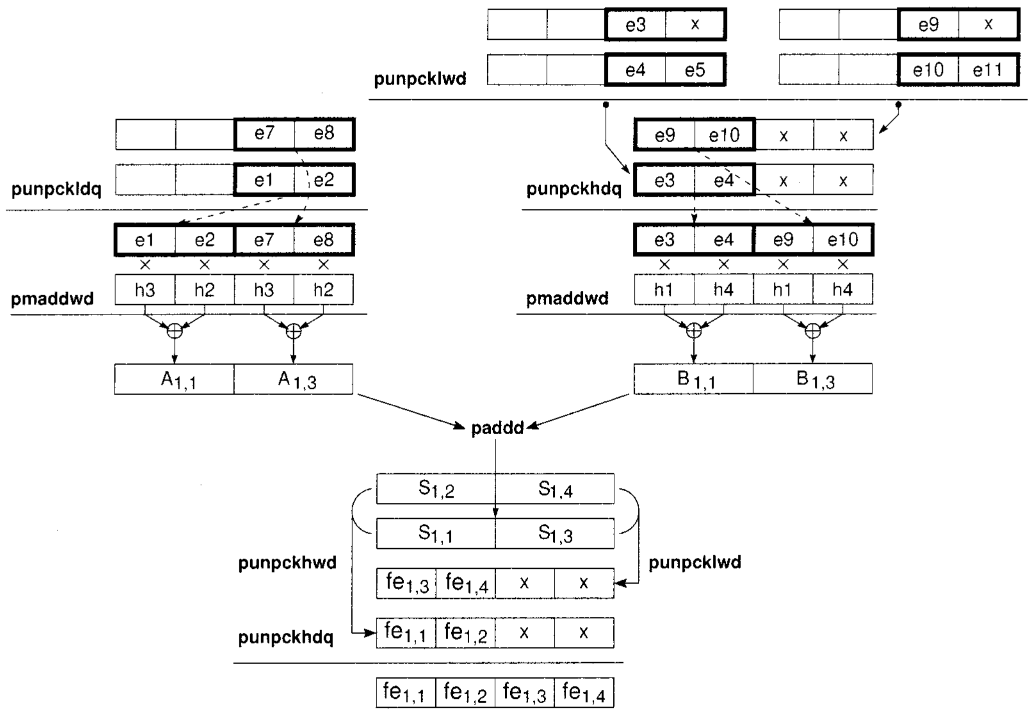


Fig. 16. Computation of four filtered errors. The filtered error for $p_{n,m}$, i.e., $fe_{n,m}$, is the higher 16 bits of $S_{n,m}$. Only the computation flow for $\{S_{1,1}, S_{1,3}\}$ is illustrated. $\{S_{1,2}, S_{1,4}\}$ are computed in a similar way.

3) For the SQ and SS methods, almost all delays of the multi-latency operations, such as *pmaddwd*, are hidden by independent operations and over 80% of the instructions are paired to be executed in the same cycle. This is mainly because of the parallel processing of the independent pixels, which has an effect equivalent to unrolling the independent loop iterations, thereby enlarging the loop kernel. This results in CPI of 0.63 and 0.62, respectively, which are near to 0.5, the minimum CPI possible with the Pentium MMX processor. The PF method generates unfilled pipeline delays and shows a low pairing ratio because of the short loop kernel. It is not useful to unroll the loop kernel of the PF method because of the tight dependency between the unrolled loop iterations.
4) Many halftoning applications employ a multi-bit quantizer, instead of 1-bit, to improve the halftoned image quality. In this case, the PF and the SS methods can be easily modified without greatly affecting the performance. However, the SQ method becomes impractical because the amount of extra computation for the transformation to break the dependency is proportional to the number of quantizer levels.

B. Cache Effect Consideration

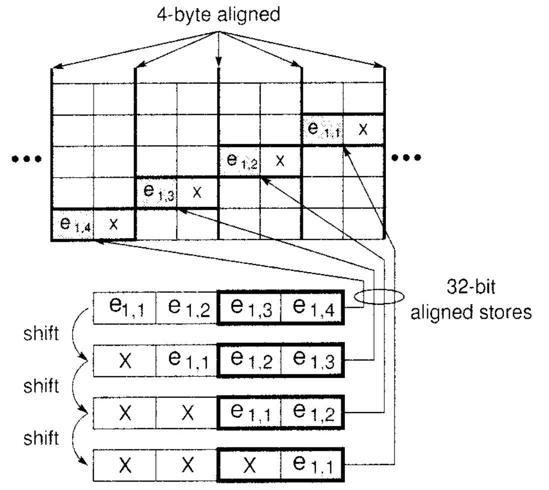
Both the PF and SQ methods require one input, one output and two error buffers for processing one scan-line. When the number of pixels in a scan-line is W and all elements are 16 bits, this corresponds to $8W$ bytes. The required number of buffers for the SS method is much larger: four inputs, four outputs and five error buffers, which amounts to $26W$ bytes. Although a store operation does not allocate a cache line and uses one of the four 64-bit write buffers in a Pentium MMX processor, output buffers should be considered for cache effects because the write buffers are also used in write-back operations when read misses occur. Note that a stall incurred in contention for the write buffers causes an extra penalty of 15 cycles.

For an A4-sized image at 400-dpi resolution, one scan-line amounts to about 9 KB, so the buffers required in the three methods cannot simultaneously reside in a 16-KB L1 data cache of a Pentium MMX processor. However, because of the high locality in the three methods, the overall overhead of cache miss penalties is not serious. Successive memory accesses from left to right along a scan-line give an effect equivalent to cache preloading [11].

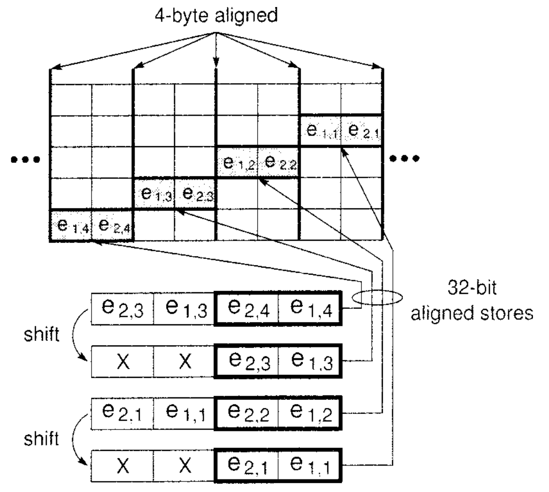
Table II shows the performance and cache miss overhead of the three methods for W =512, 1024, 2048, 4096, and 4680, respectively. For the PF, SQ, and SS methods with a 400-dpi A4-sized image, which has 4680 pixels in a scan-line, cache miss penalties per pixel are only 2.66, 2.05, and 1.82 cycles, respectively. Note that most cache misses are caused by capacity misses in the L1 data cache. This is unavoidable with the line-by-line processing schemes used in most scanning or printing devices.

C. Scalability and Applicability of the Proposed Methods

The performance of an SIMD algorithm is apparently dependent on the subword parallel processing capability of a given architecture. By estimating the performance of an SIMD-parallelized algorithm as the number of subwords processed in parallel ($\equiv\mathcal{N}$) is varied, we can assess the scalability and determine the performance bottleneck of the algorithm. Fig. 18 shows the estimated performance of the three proposed methods as $\mathcal{N}$ =2, 4, 8, and 16 using the Pentium MMX as a base architecture. For example, if $\mathcal{N} = 8$, it is assumed that an MMX



(a)



(b)

Fig. 17. Storing quantization errors for the first and second skewed group. Errors for the third and fourth groups are stored in a similar way.

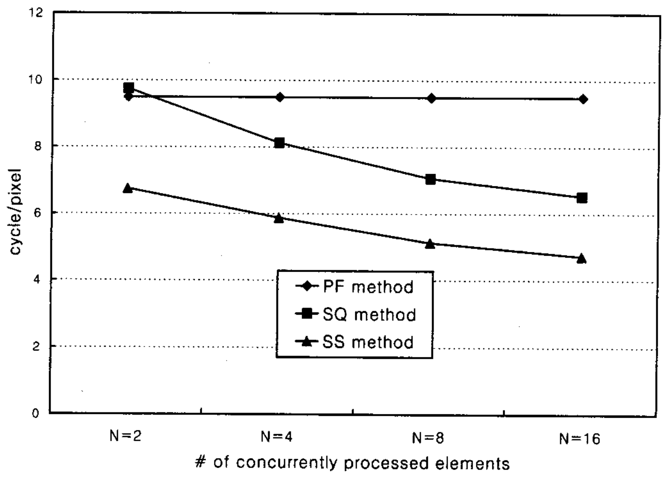


Fig. 18. Scalability of the three SIMD parallelizing methods for the error-diffusion halftoning algorithm.

register is 128-bit and 8 multiplications are performed by one packed-integer multiply instruction. Performance is estimated

TABLE I
PERFORMANCE OF THE FOUR IMPLEMENTATIONS

Method	Integer C	PF	SQ	SS
Performance (cycle/pixel)	92.7	21.1	15.5	8.9
Run-time Penalty (cycle/pixel)	3.7	3.1	0.0	0.0
Speedup over Integer C	–	4.28	5.83	10.19
CPI	2.27	0.96	0.63	0.62
Pairing (%)	40.0%	59.0%	82.8%	80.0%
Processing latency	1 line	1 line	1 line	4 lines

by counting the minimum number of machine cycles for processing one pixel where all instructions are assumed to be perfectly paired. In addition, since the amount of computation required is of concern, it is assumed that there is a sufficient number of registers and a general RISC-style two-source and one-destination format is used so that temporary copy instructions are not required.

As shown in Fig. 18, although the SQ and SS methods exhibit good scalability as $\mathcal{N}$, linear speedup is not achieved. For the SQ method, the sublinear speedup is due to the overhead of the compensation step, which is proportional to $\mathcal{N}/\mathcal{N}-1$, since only the first pixel out of $\mathcal{N}$ successive pixels need not be checked. As $\mathcal{N}$ increases, the overhead in this step converges to a constant, which limits the maximum performance. The bottleneck of the SS method is the quantization error storing routine, which also shows an asymptotically constant overhead per pixel. However, the relative portion of this routine is smaller than that of the compensation routine in the SQ method. On the other hand, the PF method benefits from increased $\mathcal{N}$ only when the halftoning algorithm employs a higher order error filter, e.g. Jarvis, Judice and Ninke's algorithm or Stucki's algorithm with a 12-tap FIR filter [18], [19]. This can be considered a severe drawback of the PF method as a parallel algorithm.

Note that although the proposed methods were implemented and analyzed with Pentium MMX instructions, they are still applicable to other subword parallel architectures. Using other architectures with more registers than Pentium MMX, slightly better performance can be expected.

VII. Concluding Remarks

The error-diffusion halftoning algorithm, which is difficult to parallelize because of nonlinear feedback dependencies, is implemented using a multimedia processor with a subword-parallel ALU architecture. Three implementation approaches, i.e., the PF, the SQ, and the SS methods are proposed, and their performances in the Intel Pentium MMX are quantitatively analyzed. The PF method exploits parallelism only in the filter kernel, thus performance improvement is very limited and the algorithm cannot effectively utilize subword parallelism greater than the filter order. The SQ method employs a complex algorithm transformation that resembles the carry-select adder, and can achieve fairly good performance by utilizing the multimedia extension instructions. However, it is only suitable for binary-level quantization because the amount of extra computation required for the transformation is proportional to the number of

TABLE II
CACHE EFFECTS OF THE THREE MMX IMPLEMENTATIONS

		$W = 512$	$W = 1024$	$W = 2048$	$W = 4096$	$W = 4680$
PF	Performance	21.10	21.77	21.76	23.76	23.76
	Cache Overhead	0	0.67	0.66	2.66	2.66
SQ	Performance	15.50	15.50	15.50	17.55	17.55
	Cache Overhead	0	0	0	2.05	2.05
SS	Performance	8.9	9.73	10.70	10.71	10.72
	Cache Overhead	0	0.83	1.80	1.81	1.82

output levels. The SS method processes multiple lines of data at a time to remove the dependency between pixels without any transformation. It can achieve the highest speedup since there is no extra computation and all the routines can utilize the multimedia extension instructions. This method, however, requires very complex data manipulations, such as matrix transposes, to eliminate inefficient memory accesses. In the Pentium MMX-based implementation, the extensive use of multiple pixel load and store operations are as important as the parallel computation itself because a single-pixel, 16-bit memory operation requires several cycles. In addition, the actual program should be implemented by considering the current pixel location and the 64-bit word boundary to eliminate memory misalignment penalties. The results show that the development of efficient parallelization techniques in both computation and memory accesses is very important for multimedia processor-based implementations, and can lead to a speedup of over ten times compared with the conventional integer C program for the structurally Psequential ED algorithm.

REFERENCES

[1] K. Suzuki, T. Arai, K. Nadehara, and I. Kuroda, “V830R/AV: An embedded multimedia superscalar RISC processor,” *IEEE Micro*, pp. 36–47, Mar./Apr. 1998.

[2] S. Rathnam and G. Slavenburg, “Processing the new world of interactive media,” *IEEE Signal Processing Mag.*, pp. 108–117, Mar. 1998.

[3] E. Holmann, T. Yoshida, A. Yamada, and A. Mohri, “A media processor for multimedia signal processing applications,” in *Proc. 1997 IEEE Workshop on Signal Processing Systems (SiPS97)*, Nov. 1997, pp. 86–96.

[4] S. Purcell, “The impact of Mpact2,” *IEEE Signal Processing Mag.*, pp. 102–107, Mar. 1998.

[5] A. Peleg and U. Weiser, “MMX technology extension to the Intel architecture,” *IEEE Micro*, vol. 16, no. 4, pp. 42–50, Aug. 1996.

[6] MIPS Technologies. (1997) MIPS extension for digital media with 3D. [Online]. Available: http://www.mips.com

[7] M. Tremblay, J. M. O’Connor, V. Narayanan, and L. He, “VIS speeds new media processing,” *IEEE Micro*, vol. 16, pp. 10–20, Aug. 1996.

[8] R. Lee, “Subword parallelism with MAX2,” *IEEE Micro*, vol. 16, pp. 51–59, Aug. 1996.

[9] Intel Corporation. (1998) VTune CD. [Online]. Available: http://developer.intel.com/design/perftool/vtune

[10] *Intel MMX Technology Overview*, Santa Clara, CA, 1996.

[11] ——, *The Complete Guide to MMX Technology*. New York: McGraw-Hill, 1997.

[12] P. M. Kogge and H. S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations,” *IEEE Trans. Comput.*, vol. C-22, pp. 786–793, Aug. 1973.

[13] W. Sung and S. K. Mitra, “Implementation of digital filtering algorithms using pipelined vector processors,” *Proc. IEEE*, pp. 1293–1303, Sept. 1987.

[14] K. K. Parhi, “Pipelining in algorithms with quantizer loops,” *IEEE Trans. Circuits Syst.*, pp. 745–754, July 1991.

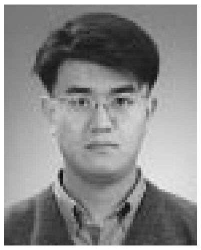
[15] J.-W. Ahn and W. Sung, “Pentium-MMX based implementation of a digital copier,” in *Proc. 1998 IEEE Workshop on Signal Processing Systems (SiPS98)*, Oct. 1998, pp. 142–151.

[16] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[17] R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan, “Evaluating MMX technology using DSP and multimedia applications,” in *Proc. 31st Annu. ACM/IEEE Int. Symp. on Microarchitecture*, Nov. 1998, pp. 37–46.

[18] J. F. Jarvis, C. N. Judice, and W. H. Ninke, “A survey of techniques for the display of continuous tone pictures on bilevel displays,” *Comput. Graphics and Image Processing*, pp. 13–40, 1976.

[19] P. Stucki, “MECCA—A multiple-error correcting computation algorithm for bilevel image hardcopy reproduction,” IBM Research Lab., Zurich, Switzerland, Tech. Rep. RZ1060, 1981.



Jae-Woo Ahn received the B.S. and M.S. degrees in electronic engineering from Seoul National University, Seoul, Korea, in 1995 and 1997, respectively. He is currently working toward the Ph.D. degree at the School of Electrical Engineering, Seoul National University.

His research interests include software design and implementation of parallel DSP algorithms, optimizing compilers, computer networks, and multimedia processor architectures.

Wonyong Sung (S’84–M’87) received the B.S. degree in electronics engineering from the Seoul National University, Seoul, Korea, in 1978, the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Seoul, Korea, in 1980, and the Ph.D. degree in electrical and computer engineering from the University of California at Santa Barbara in 1987.

From 1980 to 1983, he was with the Central Research Laboratory, Gold Star (currently LG Electronics), Seoul, Korea. He has been a Member of the Faculty of Seoul National University since 1989. During 1993–1994, he consulted with the Alta Group, Foster City, CA, on the development of the fixed-point optimizer, automatic wordlength determination, and scaling software. From January 1998 to December 1999, he was a Chief of the System Engineering and Design Center, Seoul National University. During his Ph.D. study, he developed parallel processing algorithms, vector and multiprocessor implementation, and low-complexity FIR filter design. His major research interests are the development of fixed-point optimization tools, implementation of VLSI for digital signal processing, and multiprocessor-based implementations..

Dr. Sung is a Member of the IEEE Signal Processing Society Design and Implementation technical committee.