Matrix Register File and Extended Subwords: Two Techniques for Embedded Media Processors

Asadollah Shahbahrami shahbahrami@ce.et.tudelft.nl
Ben Juurlink benj@ce.et.tudelft.nl
Stamatis Vassiliadis stamatis@ce.et.tudelft.nl

Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology, The Netherlands
Phone: +31 15 2787362, Fax: +31 15 2784898.

ABSTRACT

In this paper we employ two techniques suitable for embedded media processors. The first technique, extended subwords, uses four extra bits for every byte in a media register. This allows many SIMD operations to be performed without overflow and avoids packing/unpacking conversion overhead because of mismatch between storage and computational formats. The second technique, the Matrix Register File (MRF), allows flexible row-wise as well as column-wise access to the register file. It is useful for many block-based multimedia kernels such as (I)DCT, 2×2 Haar Transform, and pixel padding. In addition, we propose a few new media instructions. We employ Modified MMX (MMMX), MMX with extended subwords, to evaluate these techniques. Our results show that MMMX combined with an MRF reduces the dynamic number of instructions by up to 80% compared to other multimedia extensions such as MMX.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures, SIMD

General Terms: Design, Performance.

Keywords: Embedded media processors, multimedia kernels, register file, sub-word parallelism.

1. INTRODUCTION

Many microprocessor vendors have developed multimedia instructions in order to exploit the computational characteristics of multimedia applications. The unbalance between the wide data paths of contemporary General-Purpose Processors (GPPs) and the relatively narrow data types found in multimedia algorithms has led to SIMD-like instructions that operate concurrently on, e.g., eight bytes or four 16-bit values packed in a 64-bit register. Examples of such multimedia instruction set extensions are MMX [17] and SSE [18].

When employing n-way parallel SIMD instructions, the ideal speedup is n. Usually, however, the attained speedup is much smaller. This is due to several reasons. First, the way multimedia data is stored in memory (the *storage format*) is usually too small to represent intermediate results. Data, therefore, needs to be unpacked to a larger *computational format* before it can be processed and the results have to be packed again before they can be written back to memory. Obviously, this means loss of performance due to the extra cycles required for unpacking and packing. Furthermore, it also implies a loss of parallelism due to the reduction of the vector length. Table 1 shows the percentage of SIMD data rearrangement operations that are used in Berkeley Multimedia Kernel Library (BMKL) [19]. As you can see in this table, Intel MMX/SSE executes 12.6% of these instructions (overhead instructions), and Motorola AltiVec extension executes the largest percentage of these operations (17%) because, not only it's wider register width but also it uses overhead instructions to simulate unaligned access to memory. The instructions represent the overhead necessary to put data in a format suitable to SIMD operations, it is called overhead instructions, such as `packing/unpacking` and data `re-shuffling` instructions.

Second, many block-based multimedia algorithms process data along the rows as well as along the columns. This implies that in order to employ SIMD instructions, the matrix needs to be transposed frequently. Transposition takes a significant amount of time, however. For example, transposing an 8×8 matrix consisting of single-byte elements requires 56 MMX/SSE instructions to be executed. If the elements are two bytes wide, then 88 instructions are required.

Providing smaller code size is one important factor in efficient use of processors' resources like the instruction fetch and decode stage, in embedded systems. Because of this factor, embedded processors can have higher performance and consume low-power. For this, in addition to algorithmic and code optimization, data optimization and rearrangement is another important issue for providing smaller code size and efficient processing of a continuous stream in embedded media processors. The performance can be improved by changing the data organization or by changing the way data is processed. One way, for data optimization for kernels work on rows (such as DCT/IDCT, horizontal padding, subsample horizontal, vector/matrix, and matrix/matrix) is use of a matrix transposition. But, as we already indicated transposition is expensive and takes a significant amount of time, the challenge is to transpose the matrix efficiently [8].

In this paper we employ two techniques proposed to overcome these limitations. The first technique, called *extended subwords*, uses registers that are wider than the size of a

CF'05, May 4–6, 2005, Ischia, Italy.

Multimedia Extension	Data Rearrangement Operations	SIMD Integer Arithmetic Operations	SIMD Floating Point Arithmetic Operations
VIS	9.7%	13.6%	0%
AltiVec	17%	11.8%	6.9%
MMX/SSE	12.6%	18.8%	9.3%

Table 1: Percentage of overhead instructions, SIMD integer arithmetic instructions, and floating point arithmetic instructions in the BMKL kernels [19].

packed data type in memory. Specifically, for every byte of data, there are four extra bits. This allows many computations to be performed without overflow. The second approach, called the *Matrix Register File* (*MRF*), allows load and store instructions to access the register file along the rows as well as along the columns. In other words, it allows to view the register file as a matrix, where each register corresponds to the a row of the matrix and corresponding subwords in different registers correspond to a column. In addition, we have designed a few new media instructions which we have found very useful but are lacking in, for example, MMX and SSE.

We have enhanced MMX with extended subwords and the matrix register file. The resulting architecture is called Modified MMX (MMMX). Our results show that the combination of these approaches reduces the dynamic number of instructions of many multimedia kernels by up to 80%.

This paper is organized as follows. Section 2 describes the proposed architecture, i.e., extended subwords, the matrix register file, and the instruction set architecture. Section 3 evaluates the proposed architecture by comparing the dynamic number of instructions required by MMMX implementations of several kernels to the dynamic instruction count of MMX/SSE implementations. Related work is described in Section 4. Finally, we draw some conclusions in Section 5.

2. PROPOSED ARCHITECTURE

In this section we describe extended subwords, the matrix register file, and the proposed instruction set.

2.1 Extended Subwords

Image and video data is typically stored as packed 8-bit elements, but intermediate results usually require more than 8-bit precision. As a consequence, most 8-bit media instructions will be wasted on many multimedia kernels. The packed 16-bit data type, however, is often larger than necessary for many image and video applications, and reduces the amount of parallelism that can be exploited.

In previous work [12], we have proposed Modified MMX (MMMX). MMMX features registers that are wider than the size of a packed data type in memory. Specifically, for every byte of data there are four bits of extra precision. Load instructions automatically unpack data and store instructions implicitly pack data. We have shown that four extra bit for every byte in a register is sufficient for many multimedia kernels. This is also supported in [6], where it was shown that a 12-bit data format is sufficient for 85.7% of the processing in MPEG-4 encoding.

In MMMX, there are eight multimedia registers as in MMX. However, these registers are 96 bits wide instead of 64 bits. Figure 1 illustrated that a 96-bit ALU can be partitioned into eight 12-bit ALUs. Such a partitioned ALU

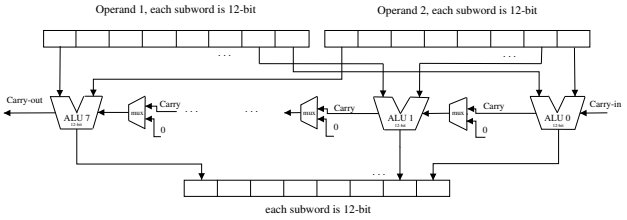

Figure 1: A 96-bit partitioned ALU.

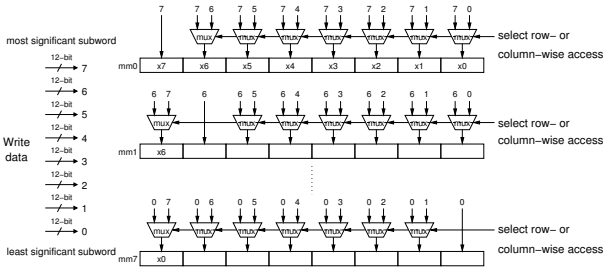

Figure 2: A matrix register file with 12-bit subwords. For simplicity, write and clock signals have been omitted.

can perform eight 12-bit, four 24-bit, two 48-bit, or a single 96-bit operation. The cost of implementing a partitioned ALU is very small.

2.2 Matrix Register File

The ability to efficiently rearrange subwords within and between registers is crucial for the performance of many media kernels. Matrix transposition, in particular, which is needed in many block-based algorithms, is a very expensive operation. To implement this operation in MMX/SSE requires many overhead instructions. To overcome this problem we propose to employ a Matrix Register File (MRF) which allows data loaded from memory to be written to a column of the register file as well as to a register (which corresponds to row-wise access).

Figure 2 illustrates an MRF with 12-bit subwords. For simplicity, write and clock signals have been omitted. Data loaded from memory can be written to a row (corresponding to a conventional media register) as well as to a column (corresponding subwords in different registers). Seven 2:1 12-bit multiplexers are needed per register/row to select between row-wise and column-wise access. For example, for register `mm0` we need to be able to select between the most significant subword of the data for column-wise access and another subword in case of row-wise access. Multiplexers are not needed for the subwords on the main diagonal.

Only load instructions can write to a column of the MRF. Thus a transposition of a matrix stored in the register file

$$
\begin{aligned}
s_{10} &= x_0 + x_7 \\
s_{11} &= x_1 + x_6 \\
s_{12} &= x_2 + x_5 \\
s_{13} &= x_3 + x_4 \\
s_{14} &= x_3 - x_4 \\
s_{15} &= x_2 - x_5 \\
s_{16} &= x_1 - x_6 \\
s_{17} &= x_0 - x_7
\end{aligned}
$$

Figure 3: First stage of the LLM algorithm for computing an 8-point DCT.

can be accomplished using a normal store followed by a column-wise load. Alternatively, we can use a column-wise store followed by a normal load. However, the first method requires fewer instructions if the matrix to be transposed is stored in memory in row-major order and needs to be processed column-wise.

As an example of where column-wise access to the register file is very useful, Figure 3 depicts the first stage of the LLM algorithm [15] for computing an 8-point DCT. An 8×8 2D DCT can be accomplished by performing a 1D DCT on each row followed by a 1D DCT on each column. Elements of each loop iteration are adjacent in memory (in a row-major storage format), the efficiency of using SIMD approach for the 1D column DCT, is more better than 1D row DCT. Because in 1D column DCT, each byte $(x_i, i = 0, 1, ...7)$ is stored in one subword of one register. This has been depicted in first subword of registers mm_i $(0 \leq i \leq 7)$ in Figure 2. While, in 1D row DCT, all of bytes of one row x_i $(0 \leq i \leq 7)$ are stored in one register. This has been depicted in $mm0$ register in Figure 2.

In order to exploit this parallelism using SIMD instructions, the elements x_0, x_7, x_1, x_6, x_2, x_5 and x_3, x_4 have to be in corresponding subwords of different registers. This implies that the high and low double words of the quadword have to be split across different registers and that the order of the subwords in one of these registers has to be reversed. An alternative way to realize a 2D DCT is by transposing the matrix so that all the x_i's of different rows are in one register. In other words, we perform several 1D DCTs in parallel rather than trying to exploit the data-level parallelism (DLP) present in a 1D DCT. If the transposition step can be implemented efficiently, this method is more efficient than the first one. Moreover, it allows to exploit 8-way SIMD parallelism if the subwords can represent the intermediate results.

We note that matrix transposition not only arises in the DCT but also in many other kernels such as the IDCT, horizontal padding, and horizontal subsampling. Furthermore, the matrix has to be transposed *twice* in order to exploit 8-way parallelism in these kernels. In addition to, using this technique, in some applications which use both rows and columns algorithms, we just use column's algorithm. For example, padding technique has four stage using matrix transpose. Matrix transposition is done twice and vertical padding algorithm is done twice as well.

2.3 Instruction Set Architecture

In this section we briefly describe the MMMX instruction set. Most MMMX instructions are direct counterparts of MMX and SSE instructions. For example, the MMMX instructions `fadd{12,24,48} mm,mm/mem64` correspond to the

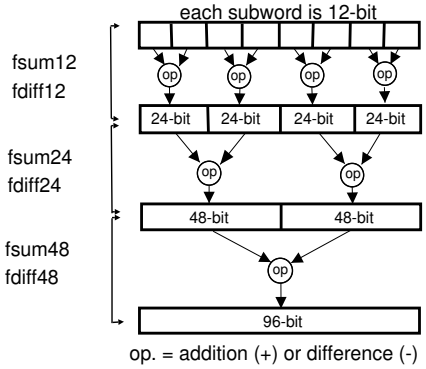

Figure 4: Reducing eight 12-bit subwords to a single 96-bit sum or 96-bit difference using the instructions `fsum{12,24,48}` and `fdiff{12,24,48}`, respectively.

MMX instructions `padd{b,w,d} mm,mm/mem64`. MMMX, however, does not support variants of these instructions that automatically saturate the results of the additions to the maximum value representable by the subword data type.

As explained before, load instructions automatically unpack the subwords and store instructions automatically pack and saturate. For example, the `fld8u12` instruction loads eight unsigned 8-bit elements from memory and zero-extends them to a 12-bit format in a 96-bit MMMX register. Vice versa, the instruction `fst12s8u` saturates the 12-bit signed subwords to 8-bit unsigned subwords before storing them to memory.

We found that in many media kernels all elements packed in one register need to be summed and in some kernels only adjacent elements need to be added or subtracted. Rather than providing different instructions for summing all elements and adding adjacent elements, we decided to support adding adjacent elements only but for every packed data type. Whereas summing all elements would probably translate to a multi-cycle operation, adding adjacent elements is a very simple operation that can most likely be implemented in a single cycle. Figure 4 illustrates how eight 12-bit subwords can be reduced to a single 96-bit sum or 96-bit difference using the instructions `fsum{12,24,48}` and `fdiff{12,24,48}`, respectively. The `fsum` instructions are used to synthesize the special-purpose SSE sum-of-absolute-differences (SAD) instruction, which is not present in MMMX because it is redundant [12].

The instructions `fmadd24l, fmadd24h` are used for multiplication and addition of two registers with some coefficients. Because using the MRF and the wider registers together in implementation of video and image kernels, we found that the multiplication and addition of two registers with third operand is very important for reducing the number of instructions. Of course, we can replace these instructions with `fmadd24` (multiplies the eight signed 12-bit of the destination operand by the eight signed 12-bit of the source operand and each two high-order 12-bit are summed and stored in the two upper 24-bit of the destination operand and the two low-order 12-bit are summed and stored in the two lower 24-bit of the destination operand), `funpckh24`, and `funpackl24` instructions that there are in MMMX ISA. Figure 5 depicts the structure of `fmadd24l, fmadd24h` instructions.

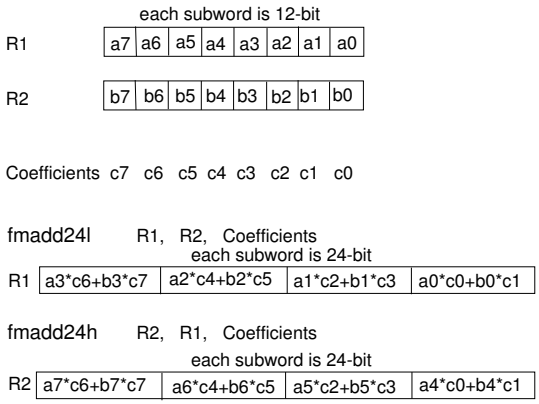

Figure 5: Structure of `fmadd24l` and `fmadd24h` instructions.

3. INITIAL EVALUATION

In this section we evaluate MMMX with and without the MRF by counting the dynamic number of instructions needed to realize several multimedia kernels using MMX as well as MMMX instructions. We need to employ this evaluation method because we do not yet have a simulator for MMMX. Nevertheless, it is a valid initial comparison because we do not replace instructions by more complex ones. In fact, MMX/SSE include multi-cycle instructions such as SAD instruction which MMMX does not support, because they can be synthesized using simpler, more general instructions. We study only kernels as opposed to entire applications. This is because they represent a major portion of many multimedia applications.

3.1 DCT

The Discrete Cosine Transform (DCT) and its inverse (IDCT) are widely used in image and video compression applications. JPEG and MPEG partition the input image into 8×8 blocks and perform the 2D DCT on each block. The input elements are either 8- or 9-bit and the output is an 8×8 block of 12-bit data between -2048 and 2047.

Our implementations are based on the LLM algorithm [15], which is performed on every row and column. The MMX/SSE implementation of the 2D DCT is due to Slingerland and Smith [19]. Because the input data is either 8- or 9-bit, they have used 16-bit functionality (4-way parallelism). For MMMX without the MRF, we have restructured the algorithm in order to be able to employ 8-way parallel SIMD instructions. Specifically, we have combined stages so that more DLP is exposed. As explained in Section 2.2, for MMMX with the MRF we perform several 1D row DCTs in parallel instead of parallelizing each 1D row DCT. This requires, however, that the matrix is transposed prior to the row and column DCTs.

Table 2 depicts the dynamic number of instructions required by the MMX implementation, the MMMX implementation without using the MRF, and the MMMX with the MRF. The number of instructions needed to realize the row DCTs and the column DCTs are presented separately. In addition, the columns labeled "Overhead instructions" present the dynamic number of data rearrangement instructions required in each implementation. This column is missing for MMMX with the MRF, because this implementation does not require any of these instructions.

Table 2 shows that to perform the row DCTs, MMX requires 462 instructions and MMMX without MRF requires 334 instructions. This is because we restructured the algorithm to expose more SIMD parallelism, as described above. Furthermore, because MMMX supports 12-bit subwords, it reduces the number of overhead instructions from 168 to 48. After the row DCTs, the points are at most 12 bits wide. Because there is not a 4-way 16-bit multiplication operation in MMX (full, meaning that all 32 bits of the results are produced), the authors of [20, 19] have used the multiply-add instruction `pmaddwd` with two sub-words set to 0. This implies that only two points are processed in a single SIMD operation. MMMX, on the other hand, can process four points simultaneously. This explains why MMMX reduces the dynamic number of instructions for the column DCTs by approximately a factor of 2.

The MRF significantly reduces the dynamic number of instructions to perform the row DCTs from 462 to 120. Here we do not need any rearrangement instructions to bring elements which need to be added together in corresponding subwords of different registers. Moreover, we can employ 8-way parallel SIMD instructions. The dynamic number of instructions required to perform the column DCTs is unchanged w.r.t. MMMX without the MRF. In total, the dynamic number of instructions needed for performing a 2D DCT on an 8×8 block is reduced from 909 to 553 using MMMX without the MRF (corresponding to a 39% reduction) and from 909 to 339 using MMMX with the MRF (63% reduction).

3.2 IDCT

The IDCT is the inverse of the DCT and can be accomplished using the same algorithm except that the process is reversed. Table 3 shows that MMMX (without the MRF) reduces the dynamic number of instructions from 685 to 641 (by 6%) compared to MMX. The reason that this reduction is smaller than for the DCT is that the input data is 12-bit and intermediate results are larger than 12-bit. MMMX is, therefore, mostly unable to exploit the 12-bit functionality. Combining MMMX with the MRF provides a much larger performance benefit. MMMX with the MRF reduces the dynamic number of instructions by 36% compared to MMX. This reduction is mostly due to the fact that the overhead instructions required to transpose the matrix have been completely eliminated.

3.3 Repetitive Padding

One important new feature in MPEG-4 is *padding*, defined at all levels in the Core and Main Profiles of the standard. Profiling results, reported in, e.g., [2, 3, 21], indicate that padding is a computationally demanding process.

MPEG-4 defines the concept of a Video Object Plane (VOP); an arbitrarily shaped region of a frame which usually corresponds to an object in the visual scene. In MPEG-4, motion estimation is defined over VOPs instead of frames and for more accurate block matching MPEG-4 adopts the padding process. The padding process defines the color values of pixels outside the VOP. It consists of two steps. First, each horizontal line of a block is scanned. If a pixel is outside the VOP and between an end point of the line and an end point of a segment inside the VOP, then it is replaced by the value of the end pixel of the segment inside the VOP. Otherwise, if the pixel is outside the VOP and between two end points of segments inside the VOP, it is replaced by the

2D DCT	MMX		MMMX without MRF		MMMX with MRF
	# Dynamic Instructions	# Overhead Instructions	# Dynamic Instructions	# Overhead Instructions	# Dynamic Instructions
Row DCTs	462	168	334	48	120
Column DCTs	447	64	219	0	219
Total	909	232	553	48	339

Table 2: The total dynamic number of instructions and the dynamic number of overhead instructions required to compute the 2D DCT of an 8×8 block for MMX and for MMMX without and with the MRF.

2D IDCT	MMX		MMMX without MRF		MMMX with MRF
	# Dynamic Instructions	# Overhead Instructions	# Dynamic Instructions	# Overhead Instructions	# Dynamic Instructions
Row IDCTs	415	118	381	152	178
Column IDCTs	270	10	260	0	260
Total	685	128	641	152	438

Table 3: The total dynamic number of instructions and the dynamic number of pack/unpack and data rearrangement instructions required to compute the 2D IDCT of an 8×8 block for MMX and for MMMX without and with the MRF.

average of these two end points. In the second step each vertical line of the block is scanned and the same procedure as described above is applied. Figure 6 illustrates horizontal and vertical repetitive padding for an 8×8 pixel block. In this figure VOP boundary pixels are indicated by a numerical value, interior pixels are denoted by an X, and pixels outside the VOP are blank.

We have implemented the algorithm described in [1], where special instructions have been proposed for both horizontal as well as vertical repetitive padding. If column-wise access to the register file is supported, however, then both steps can be performed identically in an efficient manner, and separate instructions for horizontal and vertical repetitive padding are not needed.

Table 4 depicts the dynamic number of instructions required by MMX and by MMMX without and with the MRF to perform repetitive padding on an 8×8 block and on a 16×16 block. For an 8×8 block, the MMX program requires $2 \times (56 + 85 + 139) = 560$ instructions; 56 instructions are needed to transpose the matrix, 85 instructions are needed for the first phase of the algorithm and 139 for the second phase. It can be seen that MMMX without the MRF does not decrease the dynamic number of instructions but actually slightly increases it. This reason is that the MMX (SSE extension to MMX) code employs the special-purpose packed average `pavgb` instruction which MMMX does not support. This is because with extended subwords the `pavgb` instruction offers little extra functionality because it can be synthesized using the more general-purpose instructions `fadd12` and `fsar12` (fat shift right arithmetic). Combining MMMX with the MRF, however, reduces the dynamic number of instructions to $2 \times (86 + 140) = 460$ (86 for the first phase and 140 for the second phase. This corresponds to a reduction of 18% and is completely due to the fact that we do not need to transpose the matrix using data overhead instructions.

When the block size is 16×16, the results have to be multiplied approximately by a factor of 4, because we have to perform repetitive padding on four blocks of size 8×8 and four times we have to transpose a matrix of size 8×8. The following equation shows that a matrix A of size $N \times N$, where $N = 2^n$, can be transposed by splitting the matrix into four sub-matrices a_{ij}, $i, j = 1, 2$, of size $N/2 \times N/2$ and by transposing the sub-matrices recursively.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad A^T = \begin{bmatrix} a_{11}^T & a_{21}^T \\ a_{12}^T & a_{22}^T \end{bmatrix}.$$

3.4 2×2 Haar Transform

The 2×2 Haar transform is needed to decompose an image into four different bands. A 2D Haar transform can be performed by first performing a 1D Haar transform on each row followed by a 1D Haar transform on each column. The 2×2 Haar transform is given by the following equation.

$$\underset{\text{transform of}}{\overset{2\times 2 \text{ Haar}}{\longrightarrow}} \left(\begin{bmatrix} x0 & x1 \\ x2 & x3 \end{bmatrix} \right) \underset{\text{transform}}{\overset{1D-row \text{ Haar}}{\longrightarrow}} \begin{bmatrix} x0+x1 & x0-x1 \\ x2+x3 & x2-x3 \end{bmatrix},$$

$$\underset{\text{transform}}{\overset{1D-column \text{ Haar}}{\longrightarrow}} \begin{bmatrix} x0+x1+(x2+x3) & x0-x1+(x2-x3) \\ x0+x1-(x2+x3) & x0-x1-(x2-x3) \end{bmatrix}.$$

The MMX code for the inner loop of the 2×2 Haar transform is depicted in Figure 7. The `punpcklbw` and `punpckhbw` instructions (instructions 9, 10, 12, and 13) expand the data to two bytes. Because both operands are in the same register and because MMX does not have an instruction that adds or subtracts adjacent elements, the instruction `pmaddwd` with some multiplicands set to 1 and others to -1 is used for the final addition or subtraction. The MMMX code for the 2×2 Haar transform is depicted in Figure 8.

As Table 5 depicts for an image of size $N \times M$, the dynamic number of instructions required by MMX is $(7 + 43 \times M/8) \times N/2$. For MMMX without the MRF it is $(7 + 23 \times M/8) \times N/2$, and for MMMX with the MRF it is $(7 + 60 \times M/8) \times N/8$. For large images, MMMX without the MRF decreases the dynamic number of instructions by 47% and combining MMMX with the MRF reduces it by 65%. This reduction is due to two reasons. First, 8 pixels can be processed in parallel because 12 bits is sufficient for adding or subtracting 4 pixels. Second, as described in Section 2.3, MMMX includes instructions for adding and subtracting adjacent elements in a register.

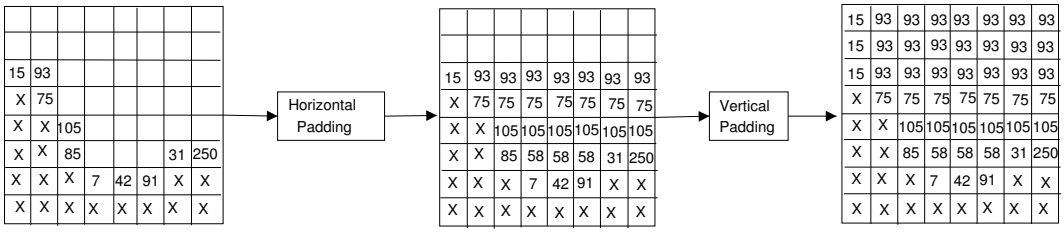

Figure 6: Repetitive padding for VOP boundary blocks.

Padding	8×8 blocks	16×16 blocks
MMX	560	2200
MMMX without MRF	572	2228
MMMX with MRF	460	1780

Table 4: Dynamic number of instructions required for performing repetitive padding on 8×8 and 16×16 blocks using MMX and MMMX with and without the MRF.

```
1   LHB01   dw  1,  1, 1,  1
2   LHB23   dw  1, -1, 1, -1
3   .row_loop:
4   movq        mm0 , [esi]
5   movq        mm1 , [esi+ ebx]
6   pxor        mm7 , mm7
7   movq        mm4 , mm0
8   movq        mm5 , mm1
9   punpcklbw   mm0 , mm7
10  punpckhbw   mm4 , mm7
11  movq        mm2 , mm0
12  punpcklbw   mm1 , mm7
13  punpckhbw   mm5 , mm7
14  paddw       mm0 , mm1
15  psubw       mm2 , mm1
16  movq        mm1 , mm0
17  pmaddwd     mm0 , LHB01
18  movq        mm3 , mm2
19  pmaddwd     mm2 , LHB01
20  movq        mm6 , mm4
21  pmaddwd     mm1 , LHB23
22  paddw       mm4 , mm5
23  pmaddwd     mm3 , LHB23
```

Figure 7: 2×2 Haar transform on an image of size $N \times M$ using MMX code.

```
1   row_loop:
2   fld8u12   mm0 , [esi]
3   fld8u12   mm1 , [esi+ ebx]
4   fld12     mm2 , mm0
5   fadd12    mm0 , mm1
6   fsub12    mm2 , mm1
7   fld12     mm1 , mm0
8   fsum12    mm0
9   fdiff12   mm1
10  fld12     mm3 , mm2
11  fsum12    mm2
12  fdiff12   mm3
```

Figure 8: 2×2 Haar Transform on an image of size $N \times M$ using MMMX code.

```
1   int16  Vec[N];
2   int16  Mat[N][M];
3   int16  Res[M];
4   int32  Acc[4];
5   for (i=0 i<M; i+=4) {
6      Acc[0..3]=0;
7      for (j=0; j<N; j+=2)
8         Acc[0..3] += Mult4x2(&Vec[j], &Mat[j][i]);
9      Res[i..i+3] = Acc[0..3];
10  }
```

Figure 9: Pseudo C code for vector matrix multiply.

Additionally, the MRF reduces the dynamic number of instructions more because it eliminates the need to explicitly transpose the matrix using rearrangement instructions. An additional advantage of the implementation that employs the MRF is that the four bands are stored in consecutive memory locations, in contrast to the other two implementations in which the four bands are stored in separate buffers.

3.5 Vector/Matrix and Matrix/Matrix Multiply

The native vector/matrix and matrix/matrix multiply algorithms traverse the matrices in columns. Matrices are typically stored in row order leaving the column elements scattered in memory. Therefore, the straightforward approach applying SIMD techniques to the inner loop is not feasible. In [10] two methods has been explained for implementation of vector/matrix multiply using MMX technology. The first method is, the matrix is split into elements of 4×2, and the input vector is also split into elements of two. The C algorithm of this method is depicted in Figure 9.

This algorithm works on four columns of the matrix in parallel (Mult4x2 function) and accumulates results in a set of four accumulators. This algorithm has many drawback [10], for an example, the code shows poor cache utilization, that has been optimized with second algorithm. In second method, the outer loop of algorithm (Figure 9) is unrolled four times. This algorithm works on 16 columns of the matrix in each iteration. Each iteration of the outer loop calculates 16 results. Some part of this algorithm using MMX code is depicted in Figure 10. We compare our

Kernel	MMX	MMMX without MRF	MMMX with MRF
	# Dynamic Instructions	# Dynamic Instructions	# Dynamic Instructions
2×2 Haar Transform	$(7 + 43 \cdot M/8) \cdot N/2$	$(7 + 23 \cdot M/8) \cdot N/2$	$(7 + 60 \cdot M/8) \cdot N/8$

Table 5: Dynamic number of instructions required for performing 2×2 Haar transform using MMX, and MMMX with and without the MRF for an image of size $N \times M$.

```
1   row_loop:
2   mov         ebx , N
3   mov         esi , Vec
4   pxor        mm2 , mm2
5   pxor        mm3 , mm3
6   col_loop:
7   movd        mm7 , [esi]
8   punpckldq   mm7 , mm7
9   movq        mm0 , [edx+0]
10  movq        mm6 , [edx+2*ecx]
11  movq        mm1 , mm0
12  punpcklwd   mm0 , mm6
13  punpckhwd   mm1 , mm6
14  pmaddwd     mm0 , mm7
15  pmaddwd     mm1 , mm7
16  paddd       mm2 , mm0
17  paddd       mm3 , mm1
18  lea         edx , [edx+4*ecx]
19  add         esi , 4
20  sub         ebx , 2
21  jnz         col_loop
```

Figure 10: MMX code for *Mult4x2* function.

MMMX code with this MMX code, because this second algorithm rather than first algorithm based on speed and the number of instructions is better.

As Table 6 depicts the dynamic number of instructions using MMX code for a vector of size $1 \times N$, and a matrix of size $N \times M$ is $16 + (39 + 48 \times N/2) \times M/16$. For 16-bit input data, the MMMX code with MRF, can reduce the dynamic number of instructions 25% than the MMX code. For 12-bit input data, using 12-bit functionality in the MMMX code is possible, so in this case, 59% of the number of dynamic instructions is decreased using MMMX with MRF than MMX code. One reason for this reduction is that the MMX codes use 21 times `punpck/pack` instructions in each loop iteration.

The number of dynamic instructions for implementation of the matrix/matrix multiply kernel for two matrices of size $P \times N$ and $N \times M$ using MMX and MMMX with MRF code for 16-bit data is $17 + ((39 + 48 \cdot N/2) \cdot M/16 + 4) \cdot P$, $6 + ((20 + 18 \cdot N/4) \cdot M/4 + 4) \cdot P$, respectively. Table 6 depicts the number of dynamic instructions for different codes and different input data. Reduction of 24%, and 58% in the dynamic number of instructions for 16- and 12-bit input data is obtained using the MMMX with MRF code than MMX code, respectively. One reason for this reduction is, there is no difference between MMX code for 12- and 16-bit input data.

3.6 Subsample Horizontal/Vertical

There are usually two subsample kernels, horizontal and vertical. In the vertical sub-sampling kernel, twelve pixels are weighted with coefficients and their products are summed together to create a composite pixel. Processing more than two pixels using the MMX architecture for this kernel is difficult. Because adding of two pixels is 9-bit and, we have to use 16-bit representation for it. Table 7 depicts the dynamic number of instructions for vertical filter kernel. As this table illustrates the dynamic number of instructions using MMX and MMMX without and with MRF code for an image of size $N \times M$ is $20 + (N/2 \cdot 143 + 5) \cdot M/4$, $20 + (N/2 \cdot 126 + 5) \cdot M/16$, and $20 + (N/2 \cdot 120 + 5) \cdot M/16$, respectively. In the horizontal sub-sampling kernel, seven pixels are weighted with coefficients and their products are summed together to create a composite pixel. The number of instructions has been shown in Table 7. If we transpose the matrix of image for horizontal filtering and use 12-bit architecture, the number of instructions will be reduced.

Reduction of 80% and 78% in the number of dynamic instructions can be achieved using MMMX with and without MRF than MMX code in implementation of subsample vertical kernel, respectively. Also reduction of 80% and 45% in the number of dynamic instructions can be achieved using MMMX with and without MRF than MMX code in implementation of subsample horizontal kernel, respectively.

4. RELATED WORK

Extended subwords have been previously proposed in [20], where they are called fat subwords. A register file architecture that provides both row- and column-wise accesses has been proposed in [11]. We build on these previous works but significantly extend on them. Specifically, our main contributions are:

- In [20] extended subwords have been proposed but not evaluated. Our work shows that extended subwords can be employed for many important multimedia kernels and that this technique significantly reduces the number of instructions that need to be fetched, decoded, and executed.
- We combine extended subwords with the MRF. Our results show that using either of these techniques is insufficient to eliminate all overhead instructions (see Figure 11).
- We have designed a few new instructions (see Section 2.3) which have been found very useful for several kernels and allow to eliminate all overhead instructions for the considered kernels.

We briefly summarize other related approaches. In [14], new subword permutation instructions across multiple registers have been presented, which can perform all permutations of a 2×2 matrix. MIPS' MDMX [9] uses a predefined set of eight 8-bit and eight 16-bit wide shuffles to implement partial shuffle operations. The `Mix` instruction in HP's MAX [13] can perform any permutation of the four 16-bit elements within a 64-bit register. Motorola's AltiVec [7]

Kernels	MMX	MMMX without MRF	MMMX with MRF
	# Dynamic Instructions	# Dynamic Instructions	# Dynamic Instructions
Vector/Matrix Multiply Input data less than or equal 16-bit	$16 + (39 + 48 \cdot N/2) \cdot M/16$	$16 + (39 + 48 \cdot N/2) \cdot M/16$	$5 + (20 + 18 \cdot N/4) \cdot M/4$
Vector/Matrix Multiply input data is less than or equal 12-bit	$16 + (39 + 48 \cdot N/2) \cdot M/16$	$5 + (10 + 15 \cdot N/2) \cdot M/8$	$5 + (47 + 39 \cdot N/8) \cdot M/8$
Matrix/Matrix Multiply Input data is less than or equal 16-bit	$17 + ((39 + 48 \cdot N/2) \cdot M/16 + 4) \cdot P$	$17 + ((39 + 48 \cdot N/2) \cdot M/16 + 4) \cdot P$	$6 + ((20 + 18 \cdot N/4) \cdot M/4 + 4) \cdot P$
Matrix/Matrix Multiply Input data is less than or equal 12-bit	$17 + ((39 + 48 \cdot N/2) \cdot M/16 + 4) \cdot P$	$6 + ((10 + 15 \cdot N/2) \cdot M/8 + 4) \cdot P$	$6 + ((47 + 39 \cdot N/8) \cdot M/8 + 4) \cdot P$

Table 6: Dynamic number of instructions required for performing vector/matrix and matrix/matrix multiply using MMX and MMMX with and without the MRF for a vector of size $1 \times N$, and a matrix of size $N \times M$.

Kernels	MMX	MMMX without MRF	MMMX with MRF
	# Dynamic Instructions.	# Dynamic Instructions	# Dynamic Instructions
Subsample Vertical	$20 + (N/2 \cdot 143 + 5) \cdot M/4$	$20 + (N/2 \cdot 126 + 5) \cdot M/16$	$20 + (N/2 \cdot 120 + 5) \cdot M/16$
Subsample Horizontal	$17 + (301 + 96 \cdot M/4) \cdot N$	$17 + (465 + 96 \cdot M/8) \cdot N$	$23 + N + 40 \cdot M \cdot N/64 + (72 \cdot N/16 + 7) \cdot M$

Table 7: Dynamic number of instructions required for performing subsample horizontal/vertical using MMX and MMMX with and without the MRF for an image of size $N \times M$.

includes a three operand instruction (`vperm`) for data rearrangement. Oliver et al. [16] propose to include a subword permutation unit (SPU) in the execution pipeline. This SPU allows data permutation operations to be performed before other operations by removing permutation instructions from the instruction stream and instead having the SPU controller schedule the rearrangement instructions.

In [4] a memory-to-memory architecture for two-dimensional vectors is proposed. Control registers are used to specify the size of data in memory and the size of data during computation. If the computational format is larger than the storage format, the input data is automatically unpacked before being processed. A related proposal is the Matrix Oriented Multimedia (MOM) extension [5]. MOM contains instructions that can be viewed as vector versions of SIMD instruction, i.e., they operate on matrices and each matrix row corresponds to a packed data type. MOM supports a matrix transpose instruction that transposes an 8×8 matrix with a latency of $8 + C$ cycles but this operation cannot be pipelined.

5. CONCLUSIONS

In this paper we have evaluated two techniques proposed to reduce data reorganization overhead. The first technique, extended subwords, uses four extra bits for every byte in a media register. This allows many SIMD operations to be performed without overflow and avoids packing/unpacking conversion overhead because of mismatch between the storage and computational formats. The second technique is the Matrix Register File (MRF), which allows flexible row-wise as well as column-wise access to the register file. This eliminates the expensive transposition steps which are required for many kernels that process two-dimensional images.

MMX has been enhanced with extended subwords and the resulting architecture is called Modified MMX (MMMX). We have implemented many important multimedia kernels using MMX, MMMX without the MRF, and MMMX with the MRF. Our results, which are summarized in Figure 11 and Figure 12, show that many kernels can be implemented efficiently using MMMX enhanced with the MRF. Combining MMMX with the MRF reduces the dynamic number of instructions by up to 80% compared to MMX. Although MMMX without the MRF also reduces the dynamic number of instructions in most cases, it still incurs much data conversion and reorganization overhead. As demonstrated in Figure 11, the coalescence of both techniques completely eliminates the conversion and reorganization overhead. Hence, using this modified architecture (MMMX with MRF), efficient use of SIMD architectures, increasing parallelism, and reducing the dynamic number of instructions which are most important factors in design of embedded multimedia processors are achieved.

Future work includes developing a detailed simulator of MMMX or another multimedia instruction set extension enhanced with extended subwords in order to evaluate the proposed techniques and other techniques more elaborately.

6. REFERENCES

[1] M. Berekovic, H. J. Stolberg, M. B. Kulaczewski, and P. Pirsch. Instruction Set Extensions for MPEG-4 Video. *Journal of VLSI Signal Processing*, 23:27–49, 1999.

[2] H. C. Chang, L. G. Chen, M. Y. Hsu, and Y. C. Chang. Performance Analysis and Architecture Evaluation of MPEG-4 Video Codec System. In *IEEE Int. Symp. on Circuits and Systems*, volume 2, pages 449–452, May 2000.

[3] H. C. Chang, Y. C. Wang, M. Y. Hsu, and L. G. Chen. Efficient Algorithms and Architectures for MPEG-4 Object-Based Video Coding. In *Proc. IEEE Workshop on Signal Processing Systems*, 2000.

[4] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. A. G.

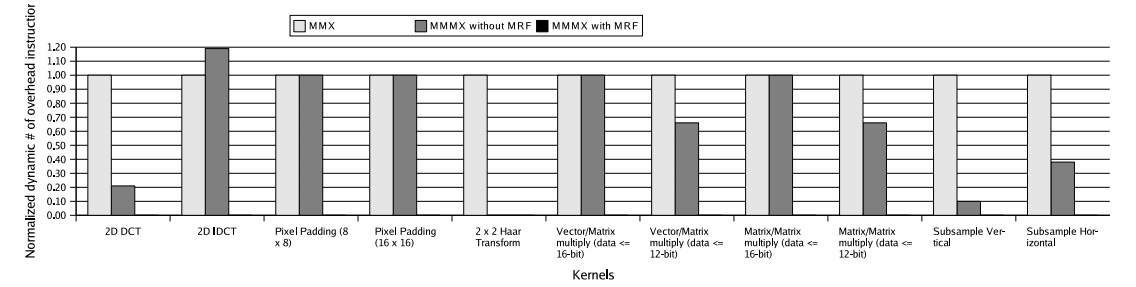

Figure 11: Comparison between the dynamic number of overhead instructions required to perform several kernels using MMX and using MMMX with and without the MRF. The results are normalized w.r.t. the number of such overhead instructions required using MMX.

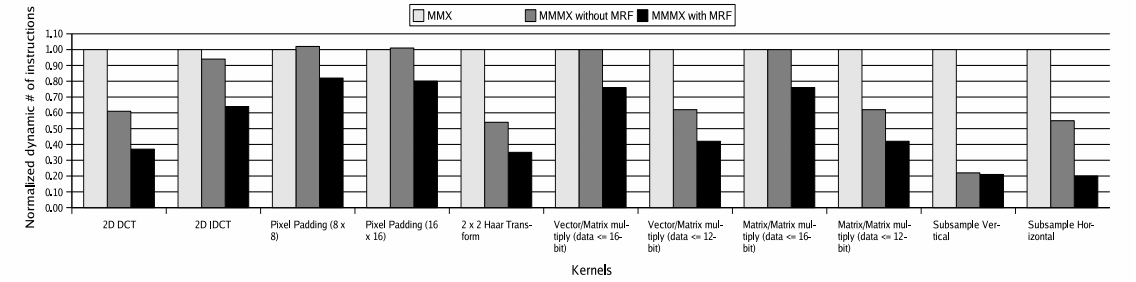

Figure 12: Comparison between the dynamic number of instructions required to perform several kernels using MMX and using MMMX with and without the MRF. The results are normalized w.r.t. the number of instructions required using MMX.

Wijshoff. The CSI Multimedia Architecture. *IEEE Trans. on VLSI Systems*, 13(1):1–13, January 2005.

[5] J. Corbal, M. Valero, and R. Espasa. Exploiting a New Level of DLP in Multimedia Applications. In *Proc. Int. Symp. on Microarchitecture*, 1999.

[6] A. Dasu and S. Panchanathan. Reconfigurable Media Processing. *Parallel computing*, 28(7), 2002.

[7] K. Diefendorff, P. K. Dubey, R. H., and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, pages 85–95, March-April 2000.

[8] B. Hanounik and X. Hu. Linear-Time Matrix Transpose Algorithms Using Vector Register File with Diagonal Registers. In *Proc. 15th Int. on Parallel and Distributed Processing*, April 2001.

[9] MIPS Technologies Inc. MIPS Extension for Digital Media with 3D. www.mips.com.

[10] Intel. An Efficient Vector/Matrix Multiply Routine using MMX Technology. Technical report, Intel Developer Services, 2004.

[11] Y. Jung, S. G. Berg, D. Kim, and Y. Kim. A Register File with Transposed Access Mode. In *Proc. Int. Conf. on Computer Design*, pages 559–560, September 2000.

[12] B. Juurlink, A. Shahbahrami, and S. Vassiliadis. Avoiding Data Conversions in Embedded Media Processors. In *Proc. 20th Annual ACM Symp. on Applied Computing*, 2005. To appear.

[13] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, pages 51–59, August 1996.

[14] R. B. Lee. Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures. In *Proc. of IEEE Int. Conf. on Application-Specific Systems Architectures and Processors*, pages 9–23, July 2000.

[15] C. Loeffler, A. Ligtenberg, and G. S. Moschytz. Practical Fast 1-D DCT Algorithms With 11 Multiplications. In *Proc. Int. Conf. on Acoustical and Speech*, volume 2, pages 988–991, 1989.

[16] J. Oliver, V. Akella, and F. Chong. Efficient Orchestration of Sub-Word Parallelism in Media Processors. In *Proc. Symp. on Parallel Algorithms and Architecture*, 2004.

[17] A. Peleg, S. Wiljie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, pages 25–38, January 1997.

[18] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing Streaming SIMD Extensions on the Pentium 3 Processor. *IEEE Micro*, pages 47–57, July-August 2000.

[19] N. Slingerland and A. J. Smith. Design and Characterization of the Berkeley Multimedia Workload. *Multimedia Systems*, 8:315–327, 2002.

[20] N. Slingerland and A. J. Smith. Measuring the Performance of Multimedia Instruction Sets. *IEEE Trans. on Computers*, 51(11):1317–1332, Nov. 2002.

[21] S. Vassiliadis, G. Kuzmanov, and S. Wong. MPEG-4 and the New Multimedia Architectural Challenges. In *Proc. 15th Int. Conf. on Systems for Automation of Engineering and Research*, pages 24–32, Sep. 2001.