Limitations of Special-Purpose Instructions for Similarity Measurements in Media SIMD Extensions

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

Microprocessor vendors have provided special-purpose instructions such as `psadbw` and `pdist` to accelerate the sum-of-absolute differences (SAD) similarity measurement. The usefulness of these special-purpose instructions is limited except for the motion estimation kernel. This has several drawbacks. First, if the SAD becomes obsolete because a different similarity metric is going to be employed, then those special-purpose instructions are no longer useful. Second, these special instructions process 8-bit subwords only. This precision is not sufficient for some kernels such as motion estimation in the transform domain. In addition, when employing other n-way parallel SIMD instructions to implement the SAD and sum-of-squared differences (SSD), the obtained speedup is much less than n. This is because there is a mismatch between the storage and the computational format. In this paper, we design and evaluate a variety of SIMD instructions for different data types. We synthesize special-purpose instructions using a few general-purpose SIMD instructions. In addition, we employ the extended subwords technique to avoid conversion overhead and to increase parallelism. In this technique there are four extra bits for every byte of register. The results show that using different SIMD instructions and extended subwords achieve a speedup ranging from 10.39 to 14.57 over C performance for SAD, SSD with interpolation, and SSD functions in the motion estimation kernel. While, MMX achieves a speedup ranging from 4.61 to 7.42. Additionally, the proposed SIMD instructions improve the performance of similarity measurement for image histograms by a factor ranging from 8.69 (1-way) to 11.70 (4-way) over C. While for MMX speedup is between 2.90 (1-way) and 4.33 (4-way).

Categories and Subject Descriptors: C.4 [Performance of Systems]: Measurement Techniques.

General Terms: Algorithms, Performance.

Keywords: Similarity Measurements, SIMD, Sub-word Parallelism.

CASES'06, October 23–25, 2006, Seoul, Korea.

1. INTRODUCTION

Similarity measurement is an important function in many media applications such as standard video coders/decoders (codecs) and image/video retrieval. The MPEG-1/2/4 and H.263/4 standards are based on motion estimation and this important kernel uses different similarity functions. Among the different similarity measurements, the Euclidean distance or Sum-of-Squared Differences (SSD) and the Sum-of-Absolute Differences (SAD) functions have been found to be the most useful [23, 19, 29, 26]. For example, in [29] eight similarity measurements for image retrieval have been evaluated. Based on the results presented there, in terms of retrieval effectiveness and retrieval efficiency, the SSD and SAD functions are more desirable than other functions. Additionally, the performance of four motion estimation algorithms using different distortion measures has been evaluated in [19]. The best results related to the quality of motion predicted frame have been obtained using the SSD and SAD functions. Furthermore, according to [26], among all the image metrics, the Euclidean distance is the most commonly used in image recognition and computer vision.

Similarity measurement, however, is very computationally intensive. For instance, in [9, 15, 18] it has been indicated that the motion estimation step in the video codecs takes about from 60% to 80% of the encoding time. Consequently, many processor vendors have designed different Single-Instruction Multiple-Data (SIMD) instructions to implement different similarity measurements. For instance, some processor vendors have provided special-purpose instructions such as the SSE instruction `psadbw` [16] and the VIS instruction `pdist` (pixel distance) [22] to accelerate motion estimation based on the SAD function. These instructions are very special-purpose instructions, however. Since they have limited usefulness except for the motion estimation kernel. This has several drawbacks.

First, if the sum-of-absolute differences becomes obsolete because a different similarity metric is going to be employed, then those special-purpose instructions are no longer useful. For example, MIPS' MDMX [6] does not provide a SAD instruction but advocates using the SSD instead. Second, as indicated in [10], the complex CISC-like semantics of special-purpose instructions makes automatic code generation difficult. Third, these special instructions process 8-bit subwords only. This precision is not sufficient for multimedia kernels such as motion estimation in the transform domain or for cost functions used in image and video retrieval [12]. In addition, this 8-bit precision is not also suf-

ficient for using quarter-pixel resolution, which is used in some standards such as H.264 [21]. Finally, since these instructions process eight 8-bit subwords, they are most useful if the vector length is a multiple of 8. In the H.264 standard, however, variable block sizes, for instance 8×4 and 4×4 are used [21].

In addition, when employing other n-way parallel SIMD instructions to implement similarity measurements, the obtained speedup is much less than n. This is because there is a mismatch between the storage format and the computational format. Consequently, unpacking is required before operations are performed and the results also have to be packed before they can be stored back to memory. This means loss of performance due to the execution of overhead instructions and because fewer subwords can be processed in parallel.

As a result, different applications need different data types to implement different similarity measurements. For example, image/video retrieval systems need data types larger than 8- and 16-bit. So, there are no general and useful SIMD instructions to implement different similarity measurements for different data types on existing SIMD architectures.

In this paper, we design and evaluate new SIMD instructions to overcome these limitations. We use our new SIMD instructions to implement different similarity measurements focusing on the SAD and SSD for providing much higher performance compared to other media extentions such as MMX [14] and SSE [16]. This is because the increased multimedia applications, especially in the video processing domain, has given rise to its own class of processors and Instruction Set Architecture (ISA) such as embedded media processors. The reasons for using SIMD processing on embedded media processors are the following. First, ISA extensions, with the capability of SIMD processing, provide flexibility and easier upgrades from one generation to the next compared to fixed-function Application-Specific Integrated Circuits (ASICs). Second, SIMD instructions are particularly suited for embedded processors because they offer high performance at low energy consumption.

To avoid conversion overhead in the traditional SIMD processing, we employ the *extended subwords* technique, which are wider than the normal subwords that are 8-, 16-, and 32-bit. Our subwords are 12-, 24-, 48-bit. These subwords allow many operations to be performed without overflow and avoids packing/unpacking overhead instructions that are necessary when implementing similarity measurements using conventional media extensions such as MMX/SSE and VIS.

We refer to MMX equipped with general and simple SIMD instructions for different data types and the extended subwords as Modified MMX (MMMX). We have simulated SIMD instructions of the MMX/SSE and our MMMX ISA by extending the SimpleScalar toolset [2]. Our experimental results show that:

- The speedup of the MMX implementation for SAD kernel is more than MMMX implementation, because of using special-purpose `psadbw` instruction.

- MMMX achieves a speedup ranging from 10.39 to 14.57 over C performance for SAD, SSD with interpolation, and SSD functions in the motion estimation kernel. While, MMX achieves a speedup ranging from 4.61 to 7.42.

- Speedup of MMMX to implement SAD function as a similarity measurement of image histograms is between 8.69 (1-way) and 11.70 (4-way) over C. While for MMX speedup is between 2.90 (1-way) and 4.33 (4-way).

- Providing SIMD instructions for different data types is necessary to yield much more performance in new media processors compared to existing SIMD processors. This is because employing existing n-way SIMD instructions to implement multimedia kernels, the obtained speedup is much less than n, because of using overhead instructions.

The paper is structured as follows. Related work is discussed in Section 2. Section 3 describes the MMMX architecture. The different similarity measurements and their implementations using MMX/SSE and MMMX are discussed in Section 4. The experimental results are presented in Section 5, and conclusions are drawn in Section 6.

2. RELATED WORK

Special purpose `psadbw` and `pdist` instructions have been provided in the SSE [16] and in the VIS ISA [22], respectively. These instructions compute the SAD function between the corresponding 8-bit components in a pair of 64-bit registers and accumulates the error values. Intel has provided `wsad` instruction to perform the sum-of-absolute differences on 8- or 16-bit data types in the wireless MMX technology [7, 13]. In addition, ARM also introduced two special-purpose instructions, `usad8` and `usada8` to calculate the sum-of-absolute differences between 8-bit values for 4-way parallelism [4]. As already mentioned in Section 1, these special-purpose instructions have some limitations.

Waerdt and Vassiliadis [25] have proposed new operations, which are supported by the TM3270 media processor for video processing. For example, collapsed load operations with interpolation allow for a motion estimation function that evaluates 17 macroblock candidates in the 3D recursive search. They focused on MPEG-2 standard. In addition, a data type of 8-bit has been considered for SAD function.

The reason behind the popularity of the SAD function is its relative ease of implementation even though it does not perform as good as the SSD function. For example, Vassiliadis et al. [24] have proposed a hardware unit for computing the SAD function. Two example implementations, 16×1 and 16×16 pixels have been considered. Pixels were represented in 8 bits. As another example, in [5, 28] two hardware architectures for real-time implementation of a variable block size motion estimation algorithm using SAD function for H.264 standard have been proposed. In addition, Wei and Grand [27] have proposed a hardware architecture for the SAD function. Their proposed architecture has a block of 16×1 processing elements, a 4-stage adder tree, and two flexible register arrays that supports most variable block size motion estimation. These ASIC architectures are not flexible, however.

Our work differs from others in the following manner. First, we design and evaluate many new general SIMD instructions to implement different similarity measurements. This programmability feature from our ISA gives a flexibility advantage over a dedicated hardware approach. SIMD instructions enables algorithmic changes after design, multiple applications can be mapped to the same platform, and faster time-to-market. Second, we significantly extend to use

```
unsigned char blk1[16][16], blk2[16][16];
int sad = 0; short diff;
for (i=0; i<16; i++)
  for (j=0; j<16; j++) {
    diff = blk1[i][j]  - blk2[i][j];
    if (diff<0) diff = - diff;
    sad += diff;
  }
```

Figure 1: Sum-of-absolute differences.

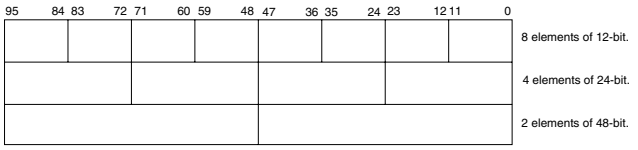

Figure 2: Different subwords in the vector register file of the MMMX architecture.

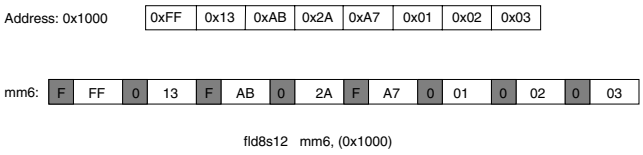

Figure 3: Illustration of the `fld8s12` instruction.

the extended subwords technique by providing experimental results obtained by a detailed, cycle-accurate simulator. Our work shows that the extended subwords can be used to avoid conversion overhead such as packing and unpacking, which are used in implementing of similarity measurements in the traditional SIMD processing. This technique significantly reduces the number of instructions that need to be fetched, decoded, and executed.

In [8] we have proposed using extended subwords to avoid data conversion overhead in many multimedia kernels. In addition, in our previous work [17], we used extended subwords and matrix register file techniques to implement many 2D media kernels such as (I)DCT, pixel padding, vector/matrix, and matrix/matrix multiply. In our previous work [8, 17] performance was evaluated by calculating the dynamic number of instructions, without using any simulators. We saw the lack of SIMD instructions for different data types. This observation motivated us for the current work.

3. MMMX ARCHITECTURE

In this section we briefly describe the MMMX architecture, which features extended subwords. In addition, we discuss the proposed instruction set.

3.1 Extended Subwords

Image and video pixels are typically stored in memory using a narrow data type, for example, 8-bit. This kind of representation is often too small for intermediate computations to occur without overflow. Consider, for example, the code that is depicted in Figure 1. This code computes the sum-of-absolute differences between two 16×16 blocks.

Since `blk1[i][j] - blk2[i][j]` is a 9-bit value, eight of these intermediate results do not fit in a single 64-bit register. The data, therefore, needs to be converted (*unpacked* or *promoted*) to the larger 16-bit format, causing conversion overhead. Furthermore, the number of subwords that are processed in parallel by a single SIMD instruction is reduced by a factor of 2. To avoid this conversion overhead and to increase parallelism, we employ the extended subwords technique. This means that the registers are wider than the data loaded into them. Specifically, for every byte of data, there are four extra bits. This implies that MMMX registers are 96 bits wide, while MMX has 64-bit registers. These registers are treated either as a vector of eight 12-bit subwords, four 24-bit subwords, or two 48-bit quantities, as is depicted in Figure 2.

3.2 Instruction Set Architecture

Multimedia applications often use different data types. It is necessary to provide SIMD instructions for different data types to yield much more performance compared to non-SIMD instructions. Since in the MMMX ISA, we try to provide a wider set of SIMD operations for different data types and also to reduce the operations complexity by providing simple and general SIMD instructions. We do not want to design special-purpose instructions with limited usefulness except for one or two kernels.

When loading data into an MMMX register, the subwords are automatically unpacked. For example, as illustrated in Figure 3, the instruction `fld8s12` loads eight signed bytes and unpacks them to signed 12-bit quantities. Vice versa, store instructions automatically saturate (clip) and pack the subwords. For example, the instruction `fst12s8u` saturates the 12-bit signed subwords to 8-bit unsigned subwords before storing them to memory.

Figure 4 illustrates the `fsum{12,24,48}` instructions, which add adjacent subwords in a media register. The instructions `fmin{12,24,48}` and `fmax{12,24,48}` return the minimum or maximum values of corresponding subwords in two registers. The instructions `fneg{12,24,48} mm,imm8` negate

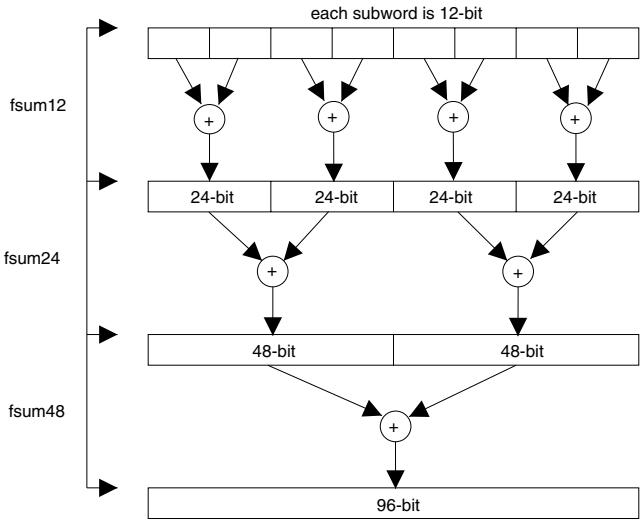

Figure 4: The structure of three fsum instructions in the MMMX architecture.

the 12-, 24-, and 48-bit subwords of the source operand if the corresponding bit in the 8-bit immediate `imm8` is set. If subwords are 24- or 48-bit, the four or six higher order bits in the 8-bit immediate are ignored. The instructions `fmadd{12,24,48}` perform the multiply-add operation on adjacent subwords. Specifically, the instruction `fmadd12` multiplies the eight signed 12-bit subwords of the first operand with the corresponding subwords of the second operand and adds adjacent 24-bit products. The instruction `fmadd24` performs the same operation but on 24-bit subwords and produces two 48-bit results. In the MMX architecture, the multiply-add operation is only supported for the packed word (4×16-bit) data type (`pmaddwd`).

The main differences between the MMX/SSE and MMMX architectures in integer part listed in Table 1. As this table depicts there are SIMD instructions for different data types in the MMMX ISA. For instance, there are full 12- and 24-bit multiply instructions in the MMMX ISA, while in the MMX/SSE ISA is not. In these instructions results are stored in both operands. Special-purpose MMX/SSE instructions such as `psadbw`, `pavg` {`b,w`} and rearrangement instructions such as `pshufw`, `packss` {`wb,dw,wb`} are not supported in the MMMX architecture [17]. In the MMMX ISA the special-purpose instructions can be synthesized using a few general-purpose instructions.

4. SIMILARITY MEASUREMENTS

In this section we discuss the two most important similarity measurements: SAD and SSD. In addition, their implementations using MMX/SSE and MMMX for motion estimation and measuring similarity between histograms for images are discussed.

4.1 SSD and SAD Functions

The cost functions of the SSD and SAD for motion estimation kernel of two $N \times N$ blocks are defined by Equation (1) and Equation (2), respectively. In these equations $x(m,n)$ represents the current block of N^2 (usually $N = 16$) pixels and $y(m+i,n+j)$ represents the corresponding block in the previous frame at new coordinates $m+i, n+j$ and w is the size of the search window.

$$SSD(i,j) = \sum_{m=1}^{N}\sum_{n=1}^{N}(x(m,n)-y(m+i,n+j))^2, -w \le i,j \le w \quad (1)$$

$$SAD(i,j) = \sum_{m=1}^{N}\sum_{n=1}^{N}|x(m,n)-y(m+i,n+j)|, -w \le i,j \le w \quad (2)$$

The SSD is more accurate and more complex than the SAD. The SAD criterion is considered a good candidate for low bit rate video applications. This is mainly due to its relatively easy hardware implementation.

These SSD and SAD functions are also used in Content-Based Image and Video Retrieval (CBIVR) systems. In CBIVR systems, images and videos are indexed into a database using a vector of features extrated from the image or video. In the retrieval stage the similarity between the features of the query image and stored feature vectors is determined. That means that computing the similarity between two images or videos can be transfered into the problem of computing the similarity between two feature vectors [11]. Hence, the large computational cost associated with CBIVR systems is related to matching algorithms for feature vectors.

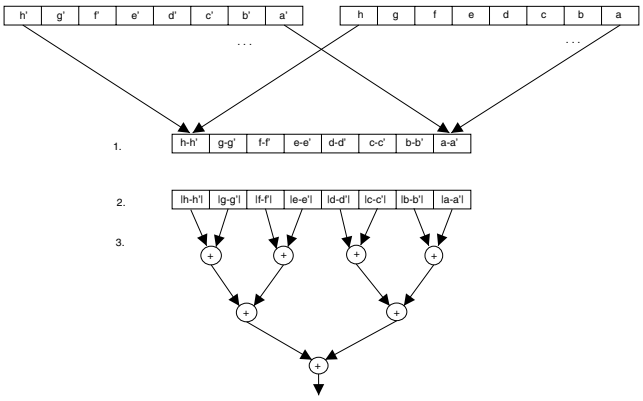

Figure 5: The structure of SAD instruction in multimedia extension.

This is because there are many feature vectors from different images and videos in the feature database.

Histogram Euclidean distance (Equation (3)) and bin-to-bin difference (b2b) (Equation (4)) are common similarity measurements, which are used in the CBIVR systems [3]. In these equations h_1 and h_2 represent two histograms, N is the number of pixels in an image, and n is the number of bits in each pixel.

$$d^2(h_1,h_2) = \sum_{i=0}^{2^n-1}(h_1[i]-h_2[i])^2 \quad (3)$$

$$fd_{b2b}(h_1,h_2) = \frac{\sum_{i=0}^{2^n-1}|(h_1[i]-h_2[i]|)}{N} \quad (4)$$

The number of elements in a histogram depends on the number of bits in each pixel in an image. For example, if we suppose a pixel depth of n bit, the pixel values will be between 0 and $2^n - 1$, and the histogram will have 2^n elements.

Components of the color histograms are usually unsigned numbers and larger than 8- and 16-bit. For instance, if we suppose a frame of size 512×512 is completely white or black, the largest element will be 2^{18}.

In the following section we discuss the MMX and MMMX implementations of these functions.

4.2 MMX/SSE and MMMX Implementations of SAD

Sum-of-absolute differences is a similarity measurement, which is usually used in the motion estimation kernel. The SAD function typically processes 16×16 blocks, as is depicted in Figure 1.

As mentioned in Section 1, this SAD function was found to be so important that many processor vendors have decided to support a special-purpose instruction for it, for example `psadbw` instruction [16]. A 64-bit `psadbw` instruction consists of 3 steps: (1) calculate eight 8-bit differences between the elements, (2) calculate the absolute value of the differences, and (3) perform three cascaded summations. These steps are illustrated in Figure 5. One reason why the `psadbw` instruction provides such a significant performance benefit is that the hardware internally keeps the carry bit.

The code in Figure 6, that has been written by MMX/SSE instructions shows how the kernel listed in Figure 1 can be

	MMX/SSE (integer part)	**MMMX**
Datapath	64-bit	96-bit
Size of register file	8 x 64-bit	8 x 96-bit
Access to register file	row-wise	row-wise + column-wise
# of partitioned ALU	8	8
Size of the integer subwords	8-, 16-, and 32-bit	12-, 24-, and 48-bit
Full multiply instruction	No	12-, 24-bit
High and low multiply inst.	16-bit	12-, 24-, and 48-bit
The size of MAC operation	16-bit	12-, 24-, and 48-bit
MAC instruction	pmaddwd	fmadd12, fmadd24, fmadd48
Special purpose instruction	No/pavgb, pavgw, psadbw	No
Saturate Add/Sub.	Yes	No
Overhead instructions	packsswb, packssdw packuswb, punpckhbw punpckhwd, punpckhdq punpcklbw, punpcklwd punpckldq/pshufw	funpckl12, funpckl24 funpckh12, funpckh24
Maximum/Minimum inst.	No/ pmaxub, pmaxsw pminub, pminsw	fmax12, fmax24, fmax48 fmin12, fmin24, fmin48
Add/Sub of adjacent subwords	No	fsum12, fsum24, fsum48 fdiff12, fdiff24, fdiff48

Table 1: The main differences between MMX/SSE and MMMX architectures.

```
(1)   mov      eax , 16
(2)   pxor     mm5 , mm5
(3) loop:
(4)   movq     mm1 , [blk1]
(5)   movq     mm2 , [blk2]
(6)   movq     mm3 , [blk1+8]
(7)   movq     mm4 , [blk2+8]
(8)
(9)   psadbw   mm1 , mm2
(10)  psadbw   mm3 , mm4
(11)
(12)  paddd    mm1 , mm3
(13)  paddd    mm5 , mm1
(14)  add      blk1, 16
(15)  add      blk2, 16
(16)  dec      eax
(17)  jnz      .loop
```

Figure 6: MMX/SSE program of SAD listed in Figure 1.

implemented using the `psadbw` instruction.

As indicated, the difference between corresponding pixels is a 9-bit value. In the MMMX architecture, we have implemented SIMD instructions to replace the `psadbw` instruction, which are more general purpose instructions and can be used in many multimedia kernels and also in other similarity measurements. This means that the SAD function can be synthesized using a small number of general-purpose SIMD instructions with only a small performance degradation, so the `psadbw` instruction essentially becomes obsolete.

Figure 7 shows how the SAD function can be implemented using MMMX instructions. Instead of the two `psadbw` instructions in the MMX/SSE program, it can be synthesized using the SIMD instructions `fsub12`, `fneg12`, `fmax12`, `fadd12`, and `fsum{12,24,48}` of the MMMX architecture, which are more general-purpose than the SAD. To provide 8-way parallelism using the extended subwords, we divided a 16×16 block into two 8×16 blocks. In the first iteration of the inner loop, the SAD function of the first 8×16 block is calculated and in the next iteration the SAD function of the second 8×16 block is performed. Finally, the re-

```
(1)   mov        ecx , 2
(2) loop2:
(3)   fxor       mm5 , mm5
(4)   mov        eax , 8
(5) loop1:
(6)   fld8u12s   mm1 , [blk1]
(7)   fld8u12s   mm2 , [blk2]
(8)   fld8u12s   mm3 , [blk1+8]
(9)   fld8u12s   mm4 , [blk2+8]
(10)
(11)  fsub12     mm1 , mm2
(12)  fneg12     mm7 , mm1
(13)  fmax12     mm1 , mm7
(14)
(15)  fsub12     mm3 , mm4
(16)  fneg12     mm7 , mm3
(17)  fmax12     mm3 , mm7
(18)
(19)  fadd12     mm1 , mm3
(20)  fadd12     mm5 , mm1
(21)  add        blk1, 16
(22)  add        blk2, 16
(23)  dec        eax
(24)  jnz        .loop1
(25)  fsum12     mm5
(26)  fsum24     mm5
(27)  fsum48     mm5
(28)  fadd96     mm6 , mm5
(29)  dec        ecx
(30)  jnz        .loop2
```

Figure 7: MMMX implementation of the SAD function listed in Figure 1.

```
(1) loop_row:
(2)   movq      mm1, [His_Current]
(3)   movq      mm2, [His_Reference]
(4)   movq      mm3, mm1
(5)   psubd     mm1, mm2
(6)   psubd     mm2, mm3
(7)   movq      mm3, mm1
(8)   movq      mm4, mm2
(9)   pcmpgtd   mm1, mm2
(10)  pcmpgtd   mm2, mm3
(11)  pand      mm1, mm3
(12)  pand      mm2, mm4
(13)  paddd     mm1, mm2
(14)  paddd     mm5, mm1
 .
 .
```

Figure 8: Part of the MMX implementation of the sum-of-absolute differences for similarity measurement of histograms.

```
(1) loop_row:
(2)   fld32s24u   mm1, [His_Current]
(3)   fld32s24u   mm2, [His_Reference]
(4)   fsub24      mm1, mm2
(5)   fneg24      mm7, mm1
(6)   fmax24      mm1, mm7
 .
 .
```

Figure 9: Part of the MMMX implementation of the sum-of-absolute differences for similarity measurement of histograms.

sults accumulate into one register using `fsum{12, 24, 48}` instructions.

Figure 8 and Figure 9 depict part of the MMX and MMMX implementations of the SAD function for similarity measurement of two histograms, respectively. In the MMX implementation 2-way parallelism is used. This is because the elements of the color histograms for different image and video frame sizes that are used in existing standards are usually unsigned numbers and larger than 8- and 16-bit and smaller than 24-bit. Elements of the histograms are stored in memory as 32-bit data type. Additionally, there is no special-purpose instruction for SAD function of 32-bit data type in the MMX ISA. This means that we have to use about 10 other instructions to implement it. In the MMMX implementation, on the other hand, 4-way parallelism is used as shown in Figure 9. This is because 24-bit subwords are sufficient for computational results of the histograms.

4.3 MMX/SSE and MMMX Implementations of SSD

The SSD function for processing a block of size 16×16 is listed in Figure 10. It can be shown analytically based on Equation (5) that 24 bits of precision is sufficient for accumulation range of the SSD implementation.

$$\sum_{i=0}^{15}\sum_{j=0}^{15}(255)^2 < (2^8)^3 = 2^{24} \tag{5}$$

This means that an unsigned 24-bit data type is sufficient, which does not map orderly to a general purpose data type.

Figure 11 and Figure 12 show how the SSD function depicted in Figure 10 can be implemented using MMX/SSE and MMMX instructions, respectively. In the MMX/SSE code 16-bit subwords are used for computation and final results are stored in 32-bit subwords. Because of this, we have to use many times punpck instructions for promotion of 8-bit to 16-bit data type. This is the reason why the static number of instructions in each iteration of the MMX/SSE implementation is 36 compared to 15 in the MMMX code. In the MMMX implementation 12-bit subwords are used for processing and final computational results are stored in 24-bit subwords.

Similarity measure for two histograms using SSD kernel with MMX/SSE ISA is also not easy. In MMX/SSE there is neither full 16-bit multiply instruction nor 32-bit multiply instruction. It instead offers two 16-bit multiply operations, `pmulhw` (packed multiply high) and `pmullw` (packed multiply low) instructions. Dividing an operation into several instructions increases register pressure and creates additional data dependencies. Additionally, most MMX/SSE SIMD instructions process integers of only 8 or 16 bits. As a result, to provide variety of the SIMD instructions is necessary

```
unsigned char blk1[16][16], blk2[16][16];
int ssd = 0;
for (i=0; i<16; i++)
  for (j=0; j<16; j++)
    ssd += (blk1[i][j] - blk2[i][j])
        *  (blk1[i][j] - blk2[i][j]);
```

Figure 10: Sum-of-squared differences.

```
(1)   mov         eax , 16
(2)   pxor        mm0 , mm0
(3)   pxor        mm7 , mm7
(4) loop:
(5)   movq        mm1 , [blk1]
(6)   movq        mm2 , [blk2]
(7)   movq        mm3 , mm1
(8)   movq        mm4 , mm2
(9)   punpcklbw   mm1 , mm0
(10)  punpckhbw   mm3 , mm0
(11)  punpcklbw   mm2 , mm0
(12)  punpckhbw   mm4 , mm0
(13)  psubw       mm1 , mm2
(14)  psubw       mm3 , mm4
(15)  movq        mm2 , mm1
(16)  movq        mm4 , mm3
(17)  pmaddwd     mm1 , mm2
(18)  pmaddwd     mm3 , mm4
(19)  paddd       mm1 , mm3
(20)  paddd       mm7 , mm1
(21)   ; for other 8 pixels
(22)  movq        mm1 , [blk1+8]
(23)  movq        mm2 , [blk2+8]
(24)   ; 13 instructions like above
(37)  paddd       mm7 , mm1
(38)  add         blk1, 16
(39)  add         blk2, 16
(40)  dec         eax
(41)  jnz         .loop
(42)  movq        mm6 , mm7
(43)  psrlq       mm7 , 32
(44)  paddd       mm7 , mm6
```

Figure 11: MMX/SSE program of the sum-of-squared differences function.

```
(1)   mov       eax , 16
(2)   fxor      mm7 , mm7
(3) loop:
(4)   fld8u12   mm1 , [blk1]
(5)   fld8u12   mm2 , [blk2]
(6)   fld8u12   mm3 , [blk1+8]
(7)   fld8u12   mm4 , [blk2+8]
(8)   fsub12    mm1 , mm2
(9)   fsub12    mm3 , mm4
(10)  fmov      mm2 , mm1
(11)  fmov      mm4 , mm3
(12)  fmadd24   mm1 , mm2
(13)  fmadd24   mm3 , mm4
(14)  fadd24    mm1 , mm3
(15)  fadd24    mm7 , mm1
(16)  add       blk1, 16
(17)  add       blk2, 16
(18)  dec       eax
(19)  jnz       .loop
(20)  fsum24    mm7
(21)  fsum48    mm7
```

Figure 12: MMMX implementation of the SSD function.

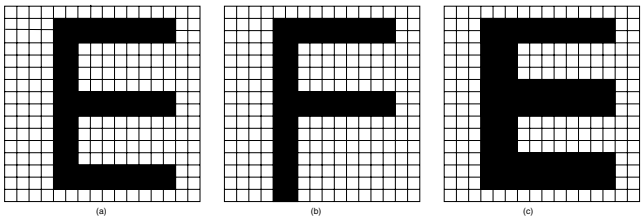

Figure 13: Similar and dissimilar blocks.

for different data types, to yield much more performance in implementation of the multimedia applications, as we have provided in the MMMX ISA.

4.4 Interpolation

The SAD and SSD similarity measurements are only a summation of the pixel-wise intensity differences and, consequently, small changes may result in a large similarity distance. For example, the Euclidean distance of Figure 13(a) and (b) is less than the Euclidean distance of (a) and (c), even though Figure 13(a) in more similar to Figure 13(c) than to (b).

For images, there are spatial relationships between pixels. There are many ways to consider the relationships between pixels, for example, averaging. Averaging neighboring pixels can be done either on two adjacent pixels horizontally, two adjacent pixels vertically, or four adjacent pixels in both horizontal and vertical dimensions.

The SSE ISA provides a special averaging instruction `pavgb` for 8-bit subwords. In addition, `wavg2` instruction has also been provided in the wireless MMX technology to perform a 2-pixel average on unsigned vectors of 8- or 16-bit data [7, 13]. To consider relationships between pixels in this paper, we implement averaging four neighboring pixels of the reference block.

The `pavgb` instruction averages two pixels, unsigned values are rounded up to the nearest integer. Averaging four pixels may produce an error of 1 when performing 3 average operations, $pavgb(x, y, z, t) = pavgb[pavgb(x, y), pavgb(z, t)]$.

```
(1) loop:
(2)   ;  Pixels 0..7
(3)   movq        mm1, [blk1]
(4)   movq        mm3, [blk1+16]
(5)   movq        mm2, mm1
(6)   movq        mm4, mm3
(7)   punpcklbw   mm1, mm0
(8)   punpcklbw   mm3, mm0
(9)   movd        mm5, [blk1+1]
(10)  movd        mm6, [blk1+17]
(11)  punpcklbw   mm5, mm0
(12)  punpcklbw   mm6, mm0
 .
 .
(30)  packuswb    mm1, mm2
(31)  psadbw      mm1, [blk2]
 .
 .
(34)  ;  Pixels 8..F
(35)  movq        mm1, [blk1+8]
(36)  movq        mm3, [blk1+24]
(37)  movq        mm2, mm1
(38)  movq        mm4, mm3
(39)  punpcklbw   mm1, mm0
(40)  punpcklbw   mm3, mm0
(41)  movd        mm5, [blk1+9]
(42)  movd        mm6, [blk1+25]
(43)  punpcklbw   mm5, mm0
(44)  punpcklbw   mm6, mm0
 .
 .
```

Figure 14: MMX/SSE program of the sum-of-absolute difference function using horizontal and vertical interpolation.

To avoid this error in the MMX/SSE implementation we use 16-bit operations using pack/unpack instructions. Figure 14 and Figure 15 show the MMX/SSE and MMMX implementations for SAD function using horizontal and vertical interpolation, respectively.

The intermediate sum of four neighboring pixels is larger than 8-bit. Hence, in the MMX/SSE implementation we should unpack data type 8-bit to 16-bit. This means that 4-

```
(1) loop:
(2)   fld8u12   mm1, [blk1]
(3)   fld8u12   mm2, [blk1+8]
(4)   fld8u12   mm3, [blk1+16]
(5)   fld8u12   mm4, [blk1+24]
(6)   fadd12    mm1, mm3
(7)   fadd12    mm2, mm4
(8)   fld8u12   mm3, [blk1+1]
(9)   fld8u12   mm4, [blk1+9]
(10)  fld8u12   mm5, [blk1+17]
(11)  fld8u12   mm6, [blk1+25]
(12)  fadd12    mm3, mm5
(13)  fadd12    mm4, mm6
(14)  fadd12    mm1, mm3
(15)  fadd12    mm2, mm4
(16)  fsra12    mm1, 2
(17)  fsra12    mm2, 2
 .
 .
```

Figure 15: MMMX implementation of the sum-of-absolute difference function using horizontal and vertical interpolation.

```
(1)   fxor        mm7, mm7
(2) loop_row:
(3)   fld32s24u   mm1, [His_Current]
(4)   fld32s24u   mm2, [His_Reference]
(5)   fmin24      mm1, mm2
(6)   fsum24      mm1
(7)   fadd48      mm7, mm1
 .
 .
```

Figure 16: Part of the MMMX code for implementation of the histogram intersection.

way parallelism is used in this code, as depicted in Figure 14. In the MMMX implementation, on the other hand, employs 8-way parallelism because 12-bit is sufficient for addition of four pixels.

4.5 Histogram Intersection Distance

In this section we want to show generality of our SIMD instructions compared to the MMX/SSE ISA. For this, we implement another distance measurement, histogram intersection.

The histogram intersection distance between the two histograms h_1 and h_2, $fd_{int}(h_1, h_2)$ was proposed by Swain and Ballard [20] and is used in image and video retrieval [29, 3]. It is defined as:

$$intersection(h_1, h_2) = \frac{\sum_{i=0}^{2^n-1} min(h_1[i], h_2[i])}{N} \quad (6)$$

$$fd_{int}(h_1, h_2) = 1 - intersection(h_1, h_2)$$

The elements of the histograms are larger than 16-bit. There are no suitable SIMD instructions for data types larger than the *short* data type in the MMX/SSE ISA. As a result, the implementation of this cost function using MMX/SSE is difficult. The above equation shows that we need SIMD instructions for finding the minimum values and addition of adjacent elements. Such SIMD instructions are available in the MMMX ISA. Figure 16 depicts part of the MMMX implementation of the histogram intersection for distance measurement of the two histograms.

5. EXPERIMENTAL SETUP

5.1 Simulation Environment

In order to evaluate MMMX, we have used the `sim-outorder` simulator of the SimpleScalar toolset [1]. `sim-outorder` is a detailed, execution-driven simulator that supports out-of-order issue and execution.

We remark that we have not simulated MMX and MMMX but rather RISC-like approximations. For example, one operand of many MMX and MMMX instructions can be a memory location, but we have simulated load/store architectures. This was done because the SimpleScalar architecture is RISC. This does not affect the validity of our simulations because our main objective is to compare the performance of an SIMD architecture without extended subwords to the same architecture with this feature. Furthermore, in the Pentium 4 MMX instructions involving memory operands are translated to RISC-like micro-operations (μOPs). We also remark that the correctness of the MMX and MMMX codes has been validated by comparing their output to the output of C programs.

The main parameters of the modeled processors are depicted in Table 2. We modeled processors by varying the issue width from 1 to 4 instructions per cycle. So, when four SIMD instructions are issued simultaneously, up to 32 data operations are executed in parallel. When the issue width is doubled, the number of functional units is scaled accordingly. For most parameters we used the default values, except for the size of the register update unit (RUU), which is 16 by default. This, however, is insufficient to find many independent instructions. We, therefore, used an RUU size of 64 instead. The latency and throughput of SIMD instructions is set equal to the latency and throughput of the corresponding scalar instructions. This is a very reasonable assumption given that the SIMD instructions perform the same operation but on narrower data types. More precisely, the latency of the integer and SIMD ALUs is set to one cycle and the latency of the integer and SIMD multiplications set to three cycles. In addition, we set the latency of the special-purpose `psadbw` instruction to five cycles, the same as in the Pentium 3.

In the experiments, three programs have been implemented and simulated using the Simplescalar simulator for each kernel. These programs employ the same algorithm and data types. Each program consists of three parts. One part is for reading the image, the second part is for similarity measurement, and the last part is for storing the results. One program is completely written in C. It was compiled using Simplescalar compiler with optimization level *-O2*. The reading and storing parts of the other two programs were also written in C, but the similarity measurement part was implemented using MMX and MMMX. These programs are referred to as C, MMX, and MMMX for each kernel. To obtain execution cycle count and ratio of dynamic number of instruction count for similarity measurement part of each case form the basis of the comparative study. Our mean from ratio of dynamic number of instruction is the ratio of the number of committed instructions for the C implementation of algorithm to the number of committed instructions for either MMX or MMMX implementation.

To evaluate the effect of the similarity measurements on the whole image, we used them for implementation of full search algorithm, on an image size of Quarter Common Intermediate Format (QCIF) size 144×176. The QCIF is a standard video format, which is used in videoconferencing and some video coding, for example, MPEG and H.26X. To determine the motion vectors for the reference blocks in the current frame, we used a macroblock of 8×8 pixel region as the basic block and $+/-16$ for the search range in the process of motion estimation.

5.2 Experimental Results

Figure 17 depicts the speedup of MMX and MMMX over the C implementation for *one execution of each kernel* with block size 16×16 on the 1-way issue out-of-order processor. It can be seen that the speedup of the MMX implementation (23.44) for the SAD kernel is higher than the speedup obtained by MMMX (17.27). This is because of the special-purpose `psadbw` instruction, which is used in the MMX code. We replaced this instruction by other simple and more general SIMD instructions. The speedup of MMMX for the other kernels is much higher than the speedup of MMX. MMMX achieves a speedup ranging from 10.39 to 14.57 over C performance, while MMX achieves a speedup ranging from 4.61 to 7.42 over C. The reasons for this are following. First,

Parameter			
Issue width	1	2	4
Integer ALU, SIMD ALU	1	2	4
Integer MULT, SIMD MULT	1	2	2
L1 Instruction cache	512-set, direct-mapped 64-byte line LRU, 1-cycle hit, total of 32 KB		
L1 Data cache	128-set, 4-way, 64-byte line, LRU, 1-cycle hit time, total of 32 KB		
L2 Unified cache	1024-set, 4-way, 64-byte line, LRU, 6-cycle hit, total of 256 KB		
Main memory latency	18 cycles for the first chunk, 2 thereafter		
Memory bus width	16 bytes		
RUU (register update unit) entries	64		
Load-store queue size	8		
Execution	out-of-order		

Table 2: Processor configuration.

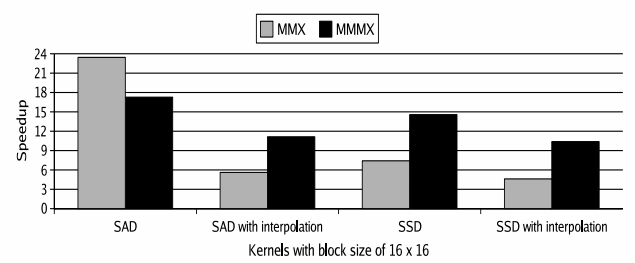

Figure 17: Speedup of MMX and MMMX over the C implementation for different kernels with block size 16×16 on 1-way issue out-of-order processor.

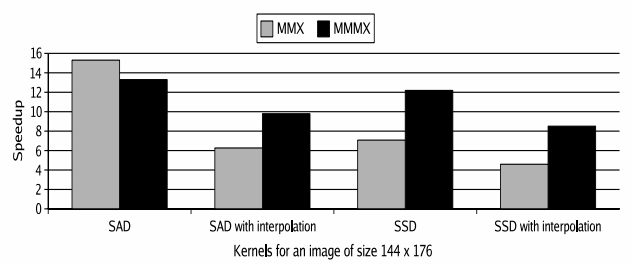

Figure 18: Speedup of MMX and MMMX over the C implementation for full search algorithm for frames size of 144×176 on 1-way issue out-of-order processor.

8-way parallelism is exploited in the MMMX code because of using the extended subwords. In the MMX code, 4-way parallelism is implemented because intermediate results are larger than 8-bit data type. This means that SIMD instructions of the MMMX architecture can pack more scalar memory and arithmetic instructions into a single SIMD instruction compared to the MMX ISA. Second, there are SIMD instrutions for different data types in the MMMX ISA.

As explained before, the results presented in Figure 17 are for one execution on a single block. The kernels are executed on all blocks of an image or frame. To investigate if this changes the results fundamentally, Figure 18 depicts the frame-level speedups (i.e., the speedups obtained when the kernels are executed on all blocks). The behavior of the results is almost similar to Figure 17. It can be seen that the frame-level speedups are smaller than the block-level speedups for both MMX and MMMX. For example, the block-level speedup using MMMX is between 10.39 and 17.27 while the frame-level speedup is between 8.51 and 13.30. The block-level speedup using MMX is between 4.61 and 23.44 while the frame-level speedup is between 4.59 and 15.30. The most important reason for this is that there are some parts of the full search algorithm that cannot be vectorized. Boundary checking and conditional operations are examples.

MMMX performs better than MMX for all kernels except for the SAD function. The main reason why MMMX improves performance compared to MMX is that it needs to execute fewer instrutions than MMX. While in the SAD kernel MMMX needs to execute more instructions than MMX.

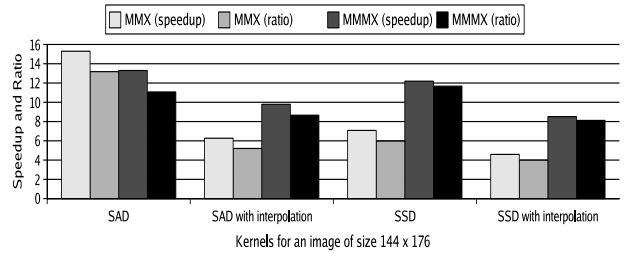

Figure 19: Ratio of committed instructions (C implementation to MMX and MMMX) versus speedup.

For explanation, Figure 19 depicts the ratio of committed instructions versus speedup. As this figure shows the ratio of committed instructions for full search algorithm that uses SAD kernel is 13.18 and 11.07 using MMX and MMMX codes, respectively. It is remarkable that the speedup is larger than the ratio of committed instructions, especially for the SAD function. This is due to the following reasons. First, in all kernels, 8-way parallelism is used in the MMMX code using extended subwords. In the MMX code special-purpose `psadbw` instruction is employed for SAD function and 4-way parallelism is employed in others kernels. This means that both codes perform more operations in a single SIMD instruction. Second, reduction of loop overhead

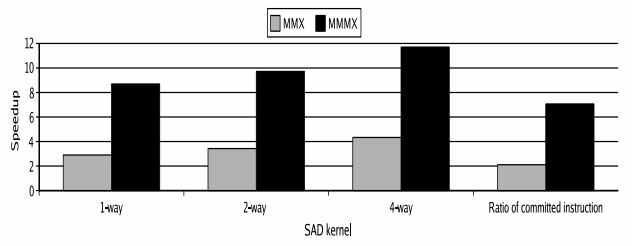

Figure 21: Speedup of MMX and MMMX over the C implementation for similarity measurement of histograms of images of size 1024×1024 using SAD kernel for different issue widths using out-of-order execution.

instructions. Both MMX and MMMX reduce a significant number of loop overhead instructions, which increments or decrements index and address values. Third, both MMX and MMMX code use short vector load and store instructions (8 bytes) compared to the C implementation that load one unsigned char in each load instruction.

Figure 20 depicts the effect of increasing the issue width. As this figure illustrates the speedup on 4-way issue width is higher than 1- and 2-way for both architectures. Additionally, more speedup is achieved on 4-way issue width for SAD kernel compared to SSD kernel.

We use much more load instructions for pixel averaging in the SAD and SSD functions with interpolation in both MMX and MMMX implementations. In addition, in the MMX implementation for all functions except SAD function we have to use promotion instructions. Based on these reasons there are much more data dependency in these codes than SAD function. Consequently, with increasing issue width less performance is yielded than SAD kernel. With increasing issue width from 2 to 4 for SSD function with interpolation in the MMMX implementation, performance is decreased. This is because of data dependency that there is between instructions. The MMMX arithmetic and logical instructions allow multiple arithmetic and logical instructions as well as multiple iterations with one MMMX instruction.

Figure 21 shows the obtained speedup and ratio of committed instructions of MMX and MMMX over the C implementation for similarity measurements of image histograms. Speedup of the MMMX is between 8.69 (1-way) and 11.70 (4-way) while for MMX speedup is between 2.90 (1-way) and 4.33 (4-way). In addition, the ratio of committed instructions is 2.11 and 7.07 using MMX and MMMX, respectively. The reasons why much more performance is yielded by MMMX especially for 4-way issue than to MMX are the following. First, 4-way parallelism is used in the MMMX implementation by using the extended subwords compared to 2-way parallelism in MMX. Second, in the MMMX code SAD function is synthesized using general and simple SIMD instructions. While in the MMX we have to used many other SIMD instructions to synthesize it that use same registers. For example, the static number of instructions in the loop body is 10 and 17 for the MMMX and MMX, respectively. This means that there is much more data dependency in the MMX code than MMMX. Third, MMMX implementation reduces loop overhead instructions two times more than MMX by 4-way parallelism.

6. CONCLUSIONS

In this paper we have designed and evaluated different, general, and simple SIMD instructions to implement some of the similarity measurements using the extended subwords technique. We have also synthesized special-purpose instructions, which are in the existing SIMD processors using a few general-purpose SIMD instructions. Our SIMD instructions can be used to implement a variety of similarity measurements. To avoid conversion overhead in the existing SIMD processing, we employed the extended subwords technique. Our subwords are wider than conventional subwords. For every byte of a media register there are four extra bits. These extra bits provide much more room for many operations to be performed without overflow and avoid packing/unpacking overhead instructions. In addition, this technique allows to perform more operations in parallel in the MMMX ISA by packing more data elements into a single media register compared to the MMX ISA.

We have implemented sum-of-absolute differences and sum-of-squared differences for motion estimation using the full search algorithm. We have also implemented the SAD function for similarity measurement of the image histograms. Our experimental results obtained by extending the SimpleScalar toolset show that providing SIMD instructions for different data types is necessary to yield much more performance in the new processors compared to existing SIMD instructions. For example, the results show that MMMX achieves a speedup ranging from 10.39 to 14.57 over C performance for SAD and SSD with interpolation and SSD functions in the motion estimation kernel. While, MMX achieves a speedup ranging from 4.61 to 7.42 over C. In addition, the speedup of MMMX to implement SAD function as a similarity measurement of image histograms is between 8.69 (1-way) and 11.70 (4-way) over C. While for MMX, the speedup is between 2.90 (1-way) and 4.33 (4-way).

7. REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[2] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. www.simplescalar.com.

[3] S. Deb. *"Video Data Management and Information Retrieval"*. IRM Press, 2005.

[4] J. Goodacre and A. N. Sloss. Parallelism and the ARM Instruction Set Architecture. *IEEE Computer*, 38(7):42–50, 2005.

[5] Y. W. Huang, T. C. Wang, B. Y. Hsieh, and L. G. Chen. Hardware Achitecture Design for Variable Block Size Motion Estimation in MPEG-4 AVC/JVT/ITU-T H.264. In *Proc. IEEE Int. Symp. on Circuits and Systems*, pages 796–799, May 2003.

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

[7] Intel Corporation. *Intel Wireless MMX Technology*, 2002. Order Number: 251793-001.

[8] B. Juurlink, A. Shahbahrami, and A. Vassiliadis. Avoiding Data Conversions in Embedded Media

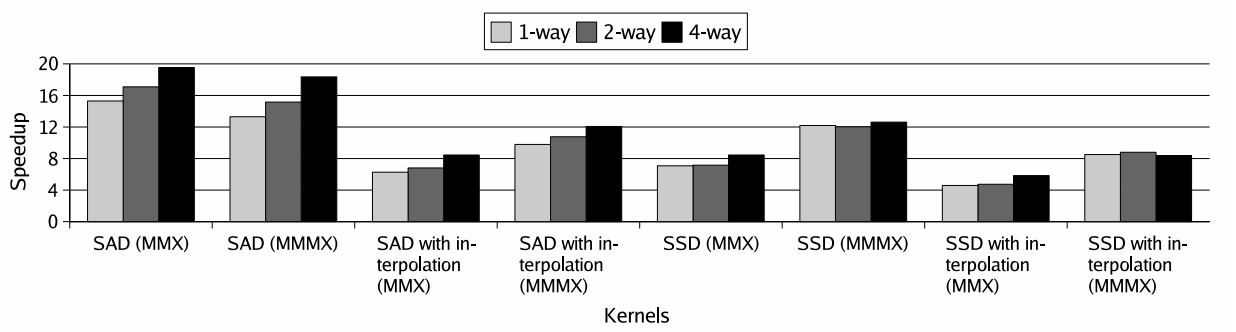

Figure 20: Speedup of MMX and MMMX over the C implementation for an image size of 144×176 for different issue widths out-of-order processors.

Processors. In *Proc. 20th Annual ACM Symp. on Applied Computing*, pages 901–902, March 2005.

[9] P. Kuhn. *"Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation"*. Kluwer Academic Publishers, 1999.

[10] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism With Multimedia Instruction Sets. In *Proc. ACM SIGPLAN 2000 Conf. on Programming language design and implementation*, pages 145–156, 2000.

[11] A. J. T. Lee, R. W. Hong, and M. F. Chang. An Approach to Content-based Video Retrieval. In *Proc. IEEE Int. Conf. on Multimedia and Expo*, volume 1, pages 273–276, June 2004.

[12] J. Lee, N. Vijaykrishnan, M. J. Irwin, and W. Wolf. An Architecture for Motion EStimation in the Transform Domain. In *Proc. 17th IEEE Int. Conf. on VLSI Design*, 2004.

[13] N. C. Paver, M. H. Khan, and B. C. Aldrich. Accelerating Mobile Multimedia Using Intel Wireless MMX Technology. In *Proc. 6th IEEE Int. Symp. on Multimedia Software Engineering*, pages 491–498, December 2004.

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

[15] M. Rabbani and P. W. Jones. *"Digital Image Compression Techniques"*. Bellinghan, 1991.

[16] 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.

[17] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. Matrix Register File and Extended Subwords: Two Techniques for Embedded Media Processors. In *Proc. 2nd ACM Int. Conf. on Computing Frontiers*, pages 171–180, May 2005.

[18] T. Shanableh and M. Ghanbari. Heterogeneous Video Transcoding to Lower Spatio-Temporal Resolutions and Different Encoding Formats. *IEEE Trans. on Multimedia*, 2(2):101–110, June 2000.

[19] S. R. Subramanya, H. Patel, and I. Ersoy. Performance Evaluation of Block-Based Motion Estimation Algorithms and Distortion Measures. In *Proc. IEEE Int. Conf. on Information Technology: Coding and Computing*, pages 2–7, 2004.

[20] M. Swain and D. Ballard. Color Indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[21] A. Tamhankar and K. R. Rao. An Overview of H.264/MPEG-4 Part 10. In *Proc. 4th Int. Conf. on Video and Image Processing and Multimedia Communications*, pages 1–51, July 2003.

[22] M. Tremblay, J. Michael 0'Connor, V. Narayanan, and L. He. VIS Speeds New Media Processing. *IEEE Micro*, pages 10–20, August 1996.

[23] S. M. Vajdic and A. R. Downing. Similarity Measures for Image Matching Architectures a Review with Classification. In *Proc. IEEE Symp. on Data Fusion*, pages 165–170, November 1996.

[24] S. Vassiliadis, E. A. Hakkennes, S. Wong, and G. G. Pechanek. The Sum-Absolute-Difference Motion Estimation Accelerator. In *Proc. 24th IEEE Euromicro Conf.*, pages 559–566, August 1998.

[25] J. W. Waerdt and S. Vassiliadis. Instruction Set Architecture Enhancements for Video Processing. In *Proc. 16th IEEE Int. Conf. on Application Specific Systems Architectures and Processors (ASAP)*, July 2005.

[26] L. Wang, Y. Zhang, and J. Feng. On the Euclidean Distance of Images. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 27(8):1334–1339, August 2005.

[27] C. Wei and M. Z. Gang. A Novel SAD Computing Hardware Architecture for Variable-Size Block Motion Estimation and Its Implementation with FPGA. In *Proc. 5th IEEE Int. Conf. on ASIC*, pages 950–953, October 2003.

[28] S. Yalcin, H. F. Ates, and I. Hamzaoglu. A High Performance Hardware Architecture for an SAD Reuse Based Hierarchical Motion Estimation Algorithm for H.264 Video Coding. In *Proc. IEEE Int. Conf. on Field Programmable Logic and Applications*, pages 509–514, August 2005.

[29] D. Zhang and G. Lu. Evaluation of Similarity Measurement for Image Retrieval. In *Proc. IEEE Int. Conf. on Neural Networks and Signal Processing*, pages 928–931, December 2003.