Versatility of Extended Subwords and the Matrix Register File

ASADOLLAH SHAHBAHRAMI, BEN JUURLINK, and STAMATIS VASSILIADIS
Delft University of Technology

Extended subwords and the matrix register file (MRF) are two micro architectural techniques that address some of the limitations of existing SIMD architectures. Extended subwords are wider than the data stored in memory. Specifically, for every byte of data stored in memory, there are four extra bits in the media register file. This avoids the need for data-type conversion instructions. The MRF is a register file organization that provides both conventional row-wise, as well as column-wise, access to the register file. In other words, it allows to view the register file as a matrix in which corresponding subwords in different registers corresponds to a column of the matrix. It was introduced to accelerate matrix transposition which is a very common operation in multimedia applications. In this paper, we show that the MRF is very versatile, since it can also be used for other permutations than matrix transposition. Specifically, it is shown how it can be used to provide efficient access to strided data, as is needed in, e.g., color space conversion. Furthermore, it is shown that special-purpose instructions (SPIs), such as the sum-of-absolute differences (SAD) instruction, have limited usefulness when extended subwords and a few general SIMD instructions that we propose are supported, for the following reasons. First, when extended subwords are supported, the SAD instruction provides only a relatively small performance improvement. Second, the SAD instruction processes 8-bit subwords only, which is not sufficient for quarter-pixel resolution nor for cost functions used in image and video retrieval. Results obtained by extending the SimpleScalar toolset show that the proposed techniques provide a speedup of up to 3.00 over the MMX architecture. The results also show that using, at most, 13 extra media registers yields an additional performance improvement ranging from 1.38 to 1.57.

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

General Terms: Design, Performance

Additional Key Words and Phrases: SIMD architectures, SIMD programming, multimedia standards

Extension of conference papers [Shahbahrami et al. 2006b; 2006c]. This research was supported in part by the Netherlands Organization for Scientific Research (NWO).
Authors' address: Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliadis, Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, The Netherlands; email: {shahbahrami,benj}@ce.et.tudelft.nl.
Asadollah Shahbahrami is also with Department of Computer Engineering, Faculty of Engineering, The University of Guilan, Rasht, Iran.

ACM Reference Format:

Shahbahrami, A., Juurlink, B., and Vassiliadis, S. 2008. Versatility of extended subwords and the matrix register file. ACM. Trans. Architec. Code Optim. 5, 1, Article 5 (May 2008), 30 pages. DOI = 10.1145/1369396.1369401 http://doi.acm.org/10.1145/1369396.1369401

1. INTRODUCTION

The efficient processing of multimedia applications is one of the main bottlenecks in the media processing field. The main reason is that there are many multimedia standards, such as MPEG-1/2/4, JPEG, and H.263/4, to capture, manipulate, store, transmit, and compress multimedia data. These standards employ different algorithms, processes, and techniques [Lee and Smith 1996]. For example, color space conversion (CSC) algorithms such as RGB-to-YCbCr and YCbCr-to-RGB are usually used in the encoder and decoder, respectively. In addition, multimedia standards use different similarity measurements, such as the sum-of-squared differences (SSD) and the sum-of-absolute differences (SAD) for motion estimation. Each algorithm in the multimedia environment has certain characteristics. For instance, CSC and similarity measurement algorithms are computationally intensive. Some CSC algorithms process the band-separated format, which is the most convenient for SIMD processing, while others process the band-interleaved format, which is difficult for SIMD processing. Motion-estimation algorithms use variable block sizes in the H.264 standard.

Our profiling of a JPEG coder/decoder (codec) shows that RGB-to-YCbCr and YCbCr-to-RGB consume an average of 13.1 and 28.7% of the total execution time, respectively. Other researchers [Bensaali and Amira 2005; Bartkowiak 2001] have reported that CSC consumes up to 40% of the entire processing time of a highly optimized decoder. In addition, in Kuhn [1999], Rabbani and Jones [1991], and Shanableh and Ghanbari [2000] it has been indicated that motion estimation takes about 60 to 80% of the encoding time. Consequently, the performance of JPEG/MPEG codecs and H.263/4 standards can be significantly improved by accelerating the CSCs and different similarity measurements.

Since both CSC and similarity measurement algorithms exhibit significant amounts of data-level parallelism (DLP), they could be implemented using the single-instruction multiple data (SIMD) instructions supported by most general-purpose processors (GPPs). Some processor vendors have provided special-purpose instructions (SPIs), such as the SSE instruction `psadbw` [Raman et al. 2000], the VIS instruction `pdist` (pixel distance) [Tremblay et al. 1996], and ARM instructions `usad8` and `usada8` [Goodacre and Sloss 2005] to accelerate motion estimation based on the SAD function. Figure 1 depicts the speedup of the MMX [Peleg et al. 1997] and SSE implementations of different similarity measurements and CSCs over a scalar implementation on the Pentium 4 processor. The speedup of MMX/SSE for the SAD function is higher than for other similarity measurements. The speedup of the SAD function used for histogram similarity measurement is only 1.20. As can also be seen,

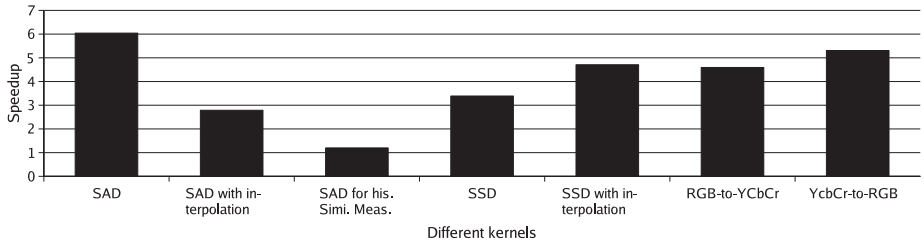

Fig. 1. Speedup of the MMX/SSE over C implementation for different kernels on the Pentium 4 processor.

the MMX implementations of RGB-to-YCbCr and YCbCr-to-RGB are 4.59 and 5.31 times faster than the corresponding C implementations, respectively.

However, there is much more DLP in these functions. In other words, CSC and similarity measurement algorithms have certain characteristics, which make them difficult to efficiently implement using existing SIMD extensions. The main performance limitations are the following.

—The image pixels are usually stored as unsigned bytes, but intermediate results to implement CSC and different similarity measurements require precision larger than 8 bits. This means that there is a mismatch between the storage and the computational format. Consequently, data-type conversion instructions, such as unpacking, are required before operations are performed and the results also have to be packed before they can be stored back to memory. As a result, performance is lost as a result of the execution of data conversion instructions and because fewer subwords can be processed in parallel.

—SIMD architectures are most efficient when the data, which is processed in parallel, is consecutively stored in memory. If not, there is a large overhead involving data-reorganization instructions. For example, in case of the CSC, the band-interleaved format is often used where the color components of each pixel are adjacent in memory. This implies that in order to efficiently employ SIMD instructions, the image pixels have to be reorganized so that the red data of different pixels are contained in one register, the green data in another, and the blue data in a third. In this case, many data reorganization instructions need to be executed.

—SPIs such as the SAD instruction have limited usefulness, except for the specific kernels they were designed to accelerate. This has several drawbacks. First, if the SAD becomes obsolete, because a different similarity metric is employed, then the SAD SPI is no longer useful. For example, MIPS' MDMX [Jennings and Conte 1998] provides no SAD SPI, but advocates using the SSD instead. Second, as indicated in Larsen and Amarasinghe [2000], the complex CISC-like semantics of SPIs makes automatic code generation difficult. Third, the SAD SPI only supports the packed-byte data type. While useful for the SAD kernel used in motion estimation, 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 [Lee et al.

2004]. In addition, this 8-bit precision is not sufficient for using quarter-pixel resolution, which is used in some standards such as H.264 [Tamhankar and Rao 2003]. 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 [Tamhankar and Rao 2003].

In order to improve the performance of the discussed multimedia functions, we accelerate them by focusing on exploiting DLP on a programmable SIMD architecture, for the following reasons. First, media applications have been changing and this proposes the use of programmable processors, instead of custom application-specific integrated circuits (ASICs) or highly specialized application-specific processors. Second, as multimedia standards become more sophisticated and larger, programmable processors need to scale their SIMD extensions in order to provide the performance required by new algorithms.

To reach this goal, we propose the use of *extended subwords* and *matrix register file* (MRF), as well as some general-purpose SIMD instructions. Extended subwords use registers that are wider than the packed format used to store the data. While conventional subwords are 8, 16, and 32 bits, extended subwords are 12, 24, and 48 bits. This avoids data-type conversion instructions. The MRF allows to consecutively load data stored in memory into a column of the register file, where a column corresponds to corresponding subwords of different registers. This technique avoids the use of data-rearrangement instructions. In addition, providing some general-purpose SIMD instructions yields much more performance than using existing SIMD and scalar instructions. Furthermore, we propose to synthesize SPIs, such as the SAD instruction, using a few general-purpose SIMD instructions and show that this can be achieved with little performance degradation.

We refer to MMX enhanced with extended subwords, the MRF, and our novel SIMD instructions as modified MMX (MMMX). We have evaluated the proposed techniques and new SIMD instructions with the MMX/SSE architectures by extending the SimpleScalar toolset [Austin et al. 2002].

We make the following contributions, compared to other works.

—We propose the use of extended subwords to alleviate data-type conversion instructions. The number of subwords that can be processed simultaneously is increased using this technique.

—We propose to use the MRF to reorganize strided data. We show that the MRF can be used to transpose a matrix to avoid data-rearrangement instructions in 2-D multimedia kernels, as well as to reorganize strided data.

—We investigate new and general SIMD instructions addressing the multimedia applications domain. We did not consider an ISA that is application specific.

—Color space conversions and SPIs were usually supported by dedicated and ASICs hardware. In this paper, on the other hand, to avoid the added cost and complexity of these dedicated hardware units, we focus on maintaining programmability while increasing performance using SIMD extension

targeting media applications. In other words, SPIs are synthesized using a few general-purpose SIMD instructions.

This paper is organized as follows. Related work is discussed in Section 2. Section 3 describes the MMMX architecture. Workloads are discussed in Section 4. The SIMD implementations of some kernels are presented in Section 5 followed by performance evaluation of proposed techniques in Section 6. Finally, conclusions and future work are given in Section 7.

2. RELATED WORK

In this section we discuss related work. We remark that although we have enhanced MMX/SSE integer extension to MMX with extended subwords and the MRF, our techniques can be applied to almost any SIMD ISA extension.

We first discuss work related to extended subwords. Extended subwords are, in a way, similar to the wide accumulators used in some DSP processors such as TMS320C64x/C64x+ DSP [Texas Instruments 2007] and MIPS' MDMX [Gwennap 1996]. In the C64x and C64x+ DSP, two 32-bit registers are used to hold a value of 40 bits. This means that 24 bits of register pairs are wasted. The MIPS' MDMX extension uses a 192-bit accumulator. This 192-bit register can be partitioned into eight 24-bit values or four 48-bit values. It is mainly used for the multiply accumulate operations common in many signal-processing algorithms. Extended subwords are also useful for other algorithms that temporarily require larger precision.

Slingerland and Smith [2002] have proposed the use of extended subwords called fat subwords. However, they have not evaluated them. Furthermore, our work shows that without a method to efficiently rearrange the subwords, such as the MRF, extended subwords are not suitable for the many 2-D media algorithms that process data along the rows as well as along the columns.

Some SIMD architectures, for example AltiVec [Motorola Inc. 1998; Diefendorff et al. 2000] and the ISA of the Cell synergistic processing element (SPE) [Flachs et al. 2006; Gschwind et al. 2006; IBM 2007], have 128-bit registers. This allows the use of computational format of, e.g., 16 bits when the storage format is 8 bits. In fact, the Cell SPE does not provide arithmetic instructions for the packed-byte data type. Our work shows, however, that 12 bits are sufficient for many media kernels and, therefore, the additional 4 bits are not needed. Furthermore, the Cell SPE requires explicit pack and unpack instructions.

We now discuss related data-reorganization methods. Slingerland and Smith [2002] proposed that SIMD architectures implement strided loads and stores to gather nonadjacent data elements, as would be useful in CSC. Strided-memory accesses would eliminate the overhead instructions, but such memory accesses are naturally slower than conventional memory accesses. In Chatterji et al. [2003], it has been indicated that one reason for poor VIRAM [Kozyrakis et al. 2000] memory performance for CSC is the strided-memory accesses.

The designers of the SIMD architectures have considered different approaches for data-rearrangement. Some SIMD architectures, such as MIPS' MDMX, MMX, and SSE have a set of permutation instructions with limited

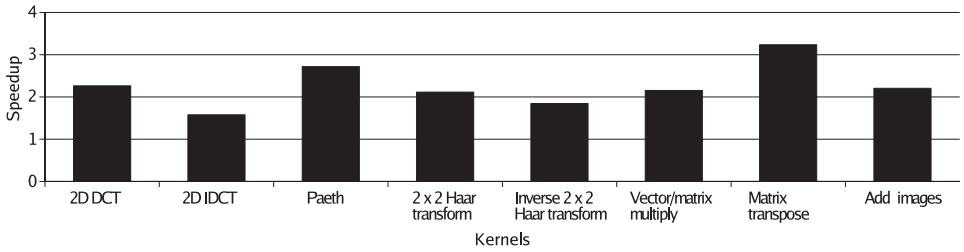

Fig. 2. Speedup of MMMX over MMX for different kernels on the single-issue processor.

capabilities. For example, SSE `pshufw` (packed shuffle word) instruction uses an immediate operand to select which of the four words in the source operand will be placed in each of the words in the destination operand. On the other hand, the AltiVec extension, the Cell SPE, and Texas Instruments C64x VLIW DSP [Seshan 1998] provide a separate permutation unit to allow an arbitrary permutation of any subword in one instruction. Although this solution provides more flexibility than MMX/SSE, many instructions are still required to transpose a matrix as is needed in many 2-D media algorithms. This is because the permutation instruction can only read two registers and write one register.

ARM's Neon Technology [Baron 2005; Goodacre and Sloss 2005] is a hybrid 64/128-bit SIMD architecture where the register file can be viewed as 32 $\times$ 64-bit registers or 16 $\times$ 128-bit registers. This architecture treats memory as an array of structures (AoS). This means that a load instruction loads subwords stored consecutively in memory into different SIMD registers. For example, the `vld3.16 {D0, D1, D2}, [R0]` instruction transfers four 3 $\times$ 16-bit structures stored in memory as $x_0, y_0, z_0, x_1, \ldots, z_3$ to the registers `D0`, `D1`, and `D2` so that `D0` contains the values $x_0, \ldots, x_3$, `D1` the values $y_0, \ldots, y_3$, and `D2` the values $z_0, \ldots, z_3$. This is very useful for CSC, but cannot be used for other data-rearrangement operations.

A different approach to eliminate data-permutation instructions named single-instruction multiple disjoint data (SIMdD) has been proposed in the eLite DSP architecture [Moreno et al. 2003; Naishlos et al. 2003]. Instead of a vector register file, the eLite DSP employs a large scalar register file, the vector element file (VEF). The elements in the VEF are addressed by four indices contained in a vector pointer register. In other words, vectors are dynamically composed. While very flexible, this approach requires four read ports to the VEF and can process, at most, four values in parallel. To process more, more read ports are required. The eLite DSP also has vector accumulator registers and a vector accumulator unit.

We have evaluated our proposed techniques in a previous paper [Shahbahrami et al. 2006a] using some 2-D multimedia kernels, such as 2-D discrete cosine transform (DCT) and its inverse (IDCT), Paeth prediction, 2 $\times$ 2 Haar transform and its inverse, vector/matrix multiplication, matrix transpose, and addition of two images. Figure 2 depicts the speedup of MMMX over MMX of the multimedia kernels on the single-issue processor. MMMX improves performance by a factor of 2.26 on average over MMX. One reason why MMMX improves the performance is the MRF technique. This technique basically

```
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]);
```

Fig. 3. C code of the sum-of-squared differences.

eliminates the matrix transposition step which is required in some kernels, for instance, 2-D (I)DCT and vector/matrix multiplication. In this paper, we show that the MRF is useful also for other permutations. Furthermore, we show that when extended subwords are provided, then SPIs can be synthesized using a few general-purpose SIMD instructions.

3. MMMX ARCHITECTURE

In this section we briefly describe the MMMX architecture, which features extended subwords and the matrix register file. In addition, we discuss the new SIMD instructions and provide a preliminary evaluation of the hardware cost of the proposed techniques. More details about the MMMX architecture can be found in previous work [Shahbahrami et al. 2006a, 2006b, 2006c].

3.1 Extended Subwords and MRF

Image, video, and audio data are usually small integers such as 8 or 16 bits, while computations on these small data types often require larger data types. Consider, for example, the code that is depicted in Figure 3. This code computes the SSD between two 16×16 blocks.

The difference between `blk1[i][j]` and `blk2[i][j]` is a 9-bit value, and the result of `(blk1[i][j] - blk2[i][j]) * (blk1[i][j] - blk2[i][j])` does not fit in a subword of either 8 or 16 bits. This is because a 24-bit subword is needed for the final result, as the following Equation (1) shows:

$$\sum_{i=0}^{15}\sum_{j=0}^{15}(255)^2 < (2^8)^3 = 2^{24}. \tag{1}$$

The data, therefore, needs to be converted to a larger format and this causes data-type conversion overhead. Furthermore, the number of subwords that are processed in parallel by a single SIMD instruction is reduced. The main reason for the data-type conversion instructions is the mismatch between the storage and the computational formats. We have examined some multimedia kernels to determine their storage and computational formats. The result is depicted in Table I.

To avoid the data-type 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

Table I. Storage and Computational Formats of Some Multimedia Kernels

Multimedia Kernels	Storage Format	Computational Format
RGB-to-YCbCr	Unsigned byte	12-bit
YCbCr-to-RGB	Unsigned byte	12-bit
SAD function	Unsigned byte	9-bit
SAD function with interpolation	Unsigned byte	10-bit
SSD function	Unsigned byte	16-bit
SSD function with interpolation	Unsigned byte	16-bit
Add Block	Unsigned byte	9-bit
2D DCT	(un)signed byte	12-bit
2×2 Haar transform	Unsigned byte	10-bit
Paeth prediction	Unsigned byte	10-bit

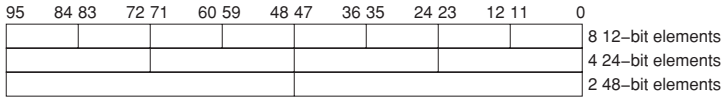

Fig. 4. Different subwords in the media register file of the MMMX architecture.

depicted in Figure 4. The extended-subwords technique increases the number of subwords that can be packed into a media register. This feature allows to perform more operations in parallel by packing more data elements into a single media register.

Conventional SIMD load and store instructions access adjacent elements in memory. Because of this, many rearrangement instructions are needed to bring strided data in a form amenable to SIMD processing. To avoid these rearrangement instructions and to reduce the code size, we use the MRF technique to reorganize strided data. The MRF allows to view the register file as a matrix. Each register corresponds to a row of the matrix and corresponding subwords in different registers correspond to a column. In other words, it provides row-wise, as well as column-wise, access to the media register file. "Load-column" instructions load data elements stored consecutively in memory into a column of the MRF. Only load-column instructions access the media register file column-wise.

Figure 5 illustrates how the MRF can be used to reorganize the band-interleaved RGB data to band separated. With eight load-column instructions (`fldc8u12`) eight red, eight green, and eight blue values are loaded into each register. Each load-column instruction loads 8 bytes (three red, three green, and two blue) values, as is shown in Figure 5. To provide correct arrangement of RGB values, an offset, which is a multiple of 6 bytes, is used for each `fldc8u12` instruction. We remark that this also works for other strides. For example, where the stride is 4, an offset, which is a multiple of 8, can be used.

3.2 MMMX Instructions

Most MMMX instructions are direct counterparts of MMX/SSE instructions, such as addition and subtraction. MMMX, however, does not support saturation arithmetic. It is not needed because load instructions automatically unpack and store instructions automatically pack and saturate, as illustrated

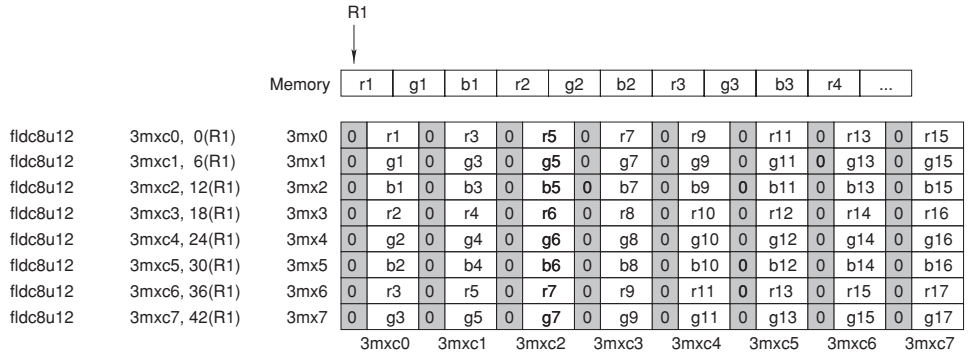

Fig. 5. Loading eight red, eight green, and eight blue values into the matrix register file using the `fldc8u12` instruction.

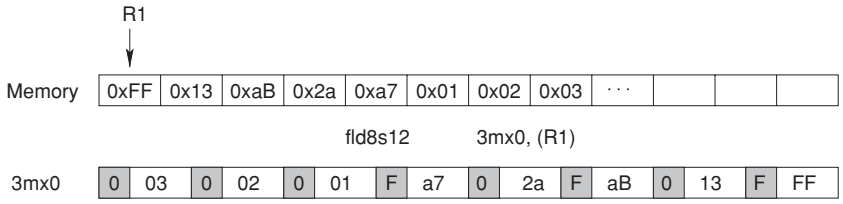

Fig. 6. The `fld8s12` instruction loads 8 signed bytes and unpacks them to signed 12-bit values.

in Figure 6 for little endian. As this figure shows, the instruction `fld8s12` loads 8 signed bytes and unpacks them to signed 12-bit quantities. On the contrary, 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.

The main differences between the MMX/SSE and MMMX ISAs are depicted in Table II. There are some general-purpose SIMD instructions in the MMMX ISA. For example, MMMX provides the `fsum{12,24,48}` and `fdiff{12,24,48}` instructions, which add and subtract adjacent elements packed in a media register, respectively. Special-purpose MMX/SSE instructions, such as `psadbw` and `pavg{b,w}`, are not supported in the MMMX architecture. In MMMX, these SPIs can be synthesized using a few general-purpose SIMD instructions. For instance, the `fsum` instructions are used to synthesize the special-purpose SAD instruction.

We have also included some partitioned multiplications such as `fmadd{12,24}` and `fmul12{l,h}` instructions. Partitioned multiplication involves the multiplication of corresponding subwords. The result of a partitioned multiplication is larger than either subword. In the MMMX architecture, we have provided two kinds of multiplications. First, multiply-accumulate (MAC) and truncation operations. The MAC operation is an important operation in digital signal processing. The MMX instruction `pmaddwd` performs the MAC operation on 16-bit subwords. In the MMMX architecture, the `fmadd{12,24}` instructions perform the operation on 12- and 24-bit subwords, respectively. Figure 7 illustrates the operation of `fmadd12 3mx0, 3mx1` instruction. This instruction multiplies each

Table II. Main Differences between MMX/SSE and MMMX ISAs

Differences	MMX/SSE (integer part)	MMMX
Datapath	64-bit	96-bit
Size of register file	8 x 64-bit	8 x 96-bit
Shared with	Floating point registers	Dedicated
Access to register file	row-wise	row-wise + column-wise
Size of the partitioned ALU	64-bit	96-bit
Size of the integer subwords	8-, 16-, and 32-bit	12-, 24-, and 48-bit
Addition instructions	padd{b, w, d}	fadd{12, 24, 48}
Subtract instructions	psub{b, w, d}	fsub{12, 24, 48}
Saturate add instructions	padds{b, w}, paddus{b, w}	No
Saturate subtract instructions	psubs{b, w}, psubus{b, w}	No
Full multiply instruction	No	fmulf{12, 24}
High and low multiply inst.	pmul{hw, lw, huw}	fmul{12l, 12h, 24l, 24h}
The size of MAC operation	16-bit	12- and 24-bit
MAC instructions	pmaddwd	fmadd{12, 24}
Increment instruction	No	finc{12, 24, 48}
Decrement instruction	No	fdec{12, 24, 48}
Negate instruction	No	fneg{12, 24, 48}
Minimum selection instructions	pmin{ub, sw}	fmin{12, 24, 48}
Maximum selection instructions	pmax{ub, sw}	fmax{12, 24, 48}
Adjacent subwords addition	No	fsum{12, 24, 48}
Adjacent subwords subtraction	No	fdiff{12, 24, 48}
Special-purpose instructions	No/pavg{b, w}, psadbw	No
Overhead instructions	packss{wb, dw} packuswb, punpckh{bw, wd, dq} punpckl{bw, wd, dq}, pshufw	funpckl{12, 24} funpckh{12, 24}

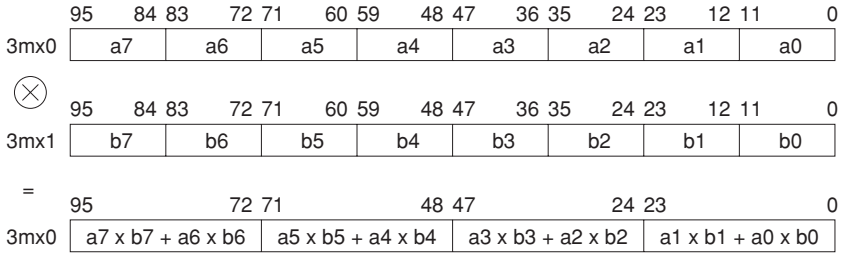

Fig. 7. Partitioned multiplication using the `fmadd12 3mx0, 3mx1` instruction.

12-bit subword of the destination operand by the corresponding 12-bit subword of the source operand. Thereafter, adjacent products are added and stored in the 24-bit subwords of the destination operand.

The second type of multiplication is truncation. Truncation means that the high or low result bits are discarded. When n-bit fixed-point values are multiplied with fractional components, the result should be n-bit of precision. Specifically, the instructions `fmul12{l,h}` multiply the eight corresponding subwords of the source and destination operands and write the low- (`fmul12l`) or high-order (`fmul12h`) 12 bits of the 24-bit product to the destination operand. This type of partitioned multiplication can be used in some applications. For example, we have used the `fmul12h` instruction in the fixed-point MMMX

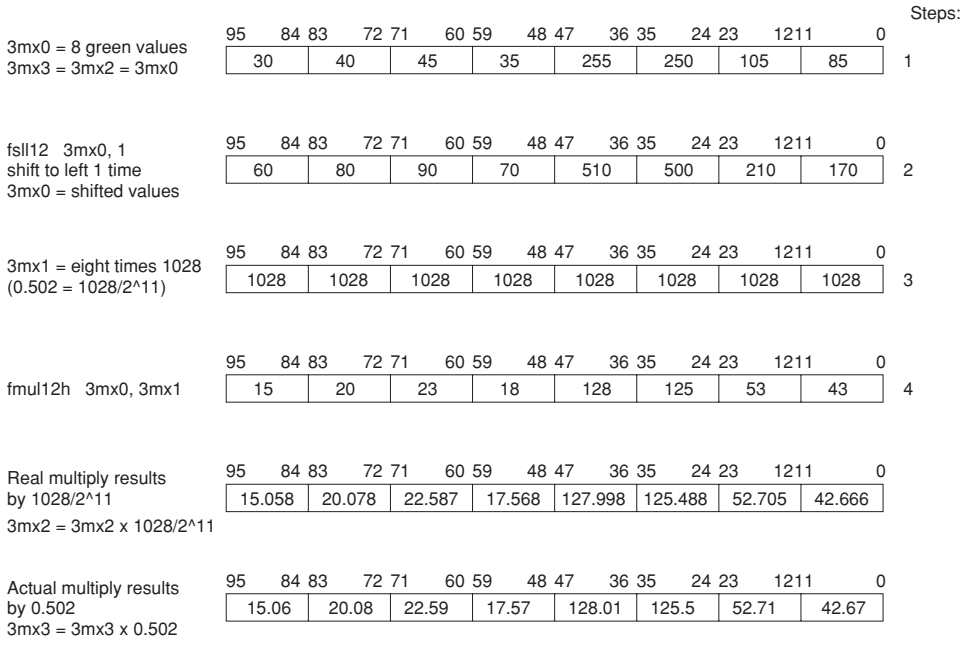

Fig. 8. Partitioned multiplication using the `fmul12h 3mx0, 3mx1` instruction.

implementation of CSC. We explain it in detail. Every green value should be multiplied by the constant coefficient 0.502. We approximate 0.502 by $1028/2^{11}$. Figure 8 depicts an example that illustrates how we use the `fmul12h` instruction to provide eight-way parallelism.

This figure shows four steps. In the first step, we load eight green pixel values into a media register (3mx0). In the second step, the subwords are shifted left by one bit. This is accomplished through the MMMX's `fsll12` instruction. This is because the `fmul12h` instruction truncates the result between the 11th and 12th bit position of the internal 24-bit result. The lower 12 bits will be discarded. For this, we need to shift the subwords 1 bit to the left. The fixed-point coefficient 1028 would exceed the 12-bit signed range if it was shifted left by 1 bit. Based on that, we shift the first operand. In the third step, the value 1028 is stored in another media register (3mx1) eight times. Finally, the shifted values are multiplied by the value 1028 using the `fmul12h 3mx0, 3mx1` instruction.

There can be some loss of precision because of this type of instruction. The first error is a result of quantizing. The coefficient is 0.502, while $1028/2^{11} = 0.501953125$. The second reason for loss of precision is because of the nature of truncation. In order to reduce the effect of this error, we first, internally round the intermediate 24-bit result; after that, we truncate the 12-bit result. On the MMX architecture, on the other hand, the `pmulhw` instruction truncates the lower 16 bits rather than rounding it. As a result, if we compare the fixed-point with the floating-point results shown in the last row of Figure 8, we can see there is a small error.

We summarize the characteristics of the MMMX architecture as follows. First, the media register file is wider than the data to be loaded into it. Second, the MMMX ISA can implicitly unpack data with load instructions. Third, store instructions implicitly pack and saturate data. Fourth, media registers can be accessed row-wise as well as column-wise. In addition, there are some general SIMD instructions for different operations to process multimedia applications, as illustrated in Table II. Furthermore, the MMMX architecture provides more subword parallelism than MMX that is shown for some multimedia kernels in the Sections 5 and 6.

3.3 Hardware Cost of the Proposed Techniques

In this section, the overhead hardware cost of the MMMX architecture over the MMX architecture in terms of area and critical-path delay is discussed and preliminary VHDL synthesis results are provided. Evaluation of the power consumption is future work.

The following are differences between the MMX and MMMX architectures from the hardware point of view. First, each MMMX register is 32 bits wider than each MMX register. Second, the MMMX register file is accessible in both directions, while in the MMX architecture it is not. This means that for column-wise access to the MMMX register file, multiplexers, and an additional decoder as well as wiring are required. Third, the MMMX ISA needs to be able to address the column registers. Finally, in the MMMX architecture, a 96-bit partitioned ALU is required to provide eight 12-bit, four 24-bit, and two 48-bit subword parallel processing. In the MMX architecture, on the other hand, a 64-bit partitioned ALU is sufficient. Furthermore, in the MMMX architecture, there are some other SIMD instructions compared to the MMX architecture.

In order to reduce the hardware cost of the MMMX architecture, column-wise access on the write port of the register file has been provided. This is because the number of write ports is usually less than the number of read ports. Only load-column instructions can access the column registers, while the other instructions cannot. The number of load-column instructions in the MMMX ISA is two. This is because 8- and 16-bit image data are loaded into column registers. These instructions are used for those kernels that use the MRF technique, such as the RGB-to-YCbCr kernel. A single bit to the instruction format of load instructions is used in order to distinguish between normal load and load-column instructions.

The register file, a 64-bit partitioned ALU, and a multiplication unit of the MMX architecture and the MRF, extended subwords, a 96-bit partitioned ALU, and a multiplication unit of the MMMX architecture have been implemented in VHDL. In addition, all SIMD arithmetic, logical, and shift instructions of both architectures have also been implemented in VHDL. In the VHDL implementation of both architectures, the same techniques and methods have been used. We target the FPGA Xilinx Virtex-II Pro xc2vp30 device. The hardware implementations have been synthesized, placed, and routed using the Xilinx ISE tool. The ratio of the MMMX area in terms of utilized

LUTs and critical-path delay over the MMX area and critical-path delay are presented.

The results show that the area utilization of the register file of the MMMX architecture is 2.89 times larger than the register file of the MMX architecture. This is because in the former eight 96-bit registers, 672 2:1 multiplexers, and two 3:8 decoders are used, while in the latter, eight 64-bit registers and one 3:8 decoder are sufficient. In addition, our timing result shows that the critical-path delay of the MRF is 5% larger than the critical-path delay of the MMX register file.

The partitioned ALU of the MMMX architecture is 1.41 times larger than the partitioned ALU of the MMX architecture. The former ALU is 30% slower than the latter ALU. The critical-path delay is because of the subword adder. We have used multiplexers in the subword boundaries to propagate or prevent the subword carries in the carry chain [Huang et al. 2007]. The partitioned ALU and the multiplication unit of the MMMX architecture are 2.27 times larger than the partitioned ALU and multiplication unit of the MMX architecture. This is because of the following reasons. First, the partitioned ALU of the MMMX architecture is wider than the partitioned ALU of the MMX architecture. Second, there are more general SIMD instructions in the MMMX ISA, such as full 12- and 24-bit multiplications. Finally, we did not consider the overhead instructions of the MMX architecture depicted in the last row of Table II. The critical-path delay of MMMX, which is related to 24-bit multiplication, is 40% longer than that of MMX. It needs to be mentioned that we have not considered pipelining the multiplication operation. In addition, in this paper, we have not used the 24-bit multiplication instruction. It has been provided for future use.

4. BENCHMARKS

To show that the MRF can be used to reorganize strided data and that SPIs provide limited benefit when extended subwords and a few general-purpose SIMD instructions are supported, we use the kernels summarized in Table III. These kernels form significant components of many media applications such as content-based image and video retrieval (CBIVR) systems and multimedia standards. In the following sections, these functions are briefly described.

4.1 Color Space Conversion

Conversion between the YCbCr and RGB formats and vice versa can be represented with the following equations [Poynton 1996].

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.256 & 0.502 & 0.098 \\ -0.148 & -0.290 & 0.438 \\ 0.438 & -0.366 & -0.071 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 16.5 \\ 128.5 \\ 128.5 \end{pmatrix} \tag{2}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.164 & 0.000 & 1.596 \\ 1.164 & -0.392 & -0.813 \\ 1.164 & 2.017 & 0.000 \end{pmatrix} \begin{pmatrix} Y - 16.5 \\ Cb - 128.5 \\ Cr - 128.5 \end{pmatrix} \tag{3}$$

Table III. Summary of the Kernels

Kernels	Description
RGB-to-YCbCr	Color space conversion, which is usually used in the encoder stage.
YCbCr-to-RGB	Color space conversion, which is usually used in the decoder stage.
SAD function	The SAD function, which is used in motion estimation kernel to remove temporal redundancies between video frames.
SAD function with interpolation	Using the SAD function with horizontal and vertical interpolation for the motion-estimation kernel.
SSD function SSD function	The SSD function, which is used in motion-estimation kernel to remove temporal redundancies between video frames.
SSD function with interpolation	Using the SSD function with horizontal and vertical interpolation for the motion-estimation kernel.
SAD function for image histograms	Using the SAD function for similarity measurements of image histograms.

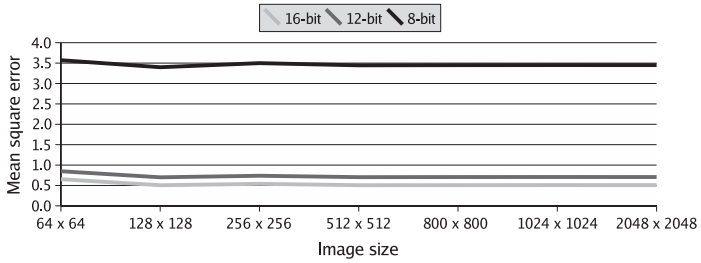

Fig. 9. Mean square error (MSE) for different bit widths in implementation of color space conversion.

In both equations, the coefficients have been rounded to three fractional decimal digits. Color space conversions are defined using floating-point arithmetic, but here, to avoid using floating-point operations, we use fixed-point arithmetic. Specifically, for MMX, we use 16-bit fixed-point numbers; for MMMX, we approximate the CSC using 12-bit fixed-point arithmetic. To determine the accuracy of these approximations, we have performed two tests in a previous paper [Shahbahrami et al. 2006b]. First, we have measured the maximum absolute error by checking all possible RGB values ($0 \leq R, G, B \leq 255$). For both the MMMX implementation (12-bit) and the MMX implementation (16-bit), the maximum absolute error compared to a single-precision floating-point implementation is 1. Second, we have measured the mean square error (MSE) for real images, as well as randomly generated inputs. Figure 9 depicts the MSE of the 8-, 12-, and 16-bit implementations as a function of the image size. It shows that the MSE of the 12- and 16-bit implementations are very close to each other and that the MSE of the 8-bit implementation is much larger.

4.2 Similarity Measurements

Among the different similarity measurements, the sum-of-squared differences (SSD) and the sum-of-absolute differences (SAD) functions have been found

to be the most useful [Zhang and Lu 2003; Wang et al. 2005]. For example, in Zhang and Lu [2003] 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 effective than other functions.

The SSD and SAD cost functions of two $N \times N$ blocks for motion estimation are defined by Equations (4) and (5), respectively. In these equations, $x(m, n)$ represents the current block of N^2 (usually $N = 16$) pixels, $y(m + i, n + j)$ represents the block in the reference frame, and (i, j) is the motion vector.

$$SSD(i, j) = \sum_{m=1}^{N} \sum_{n=1}^{N} (x(m, n) - y(m + i, n + j))^2. \tag{4}$$

$$SAD(i, j) = \sum_{m=1}^{N} \sum_{n=1}^{N} |x(m, n) - y(m + i, n + j)|. \tag{5}$$

The SSD and SAD functions are also used in CBIVR systems, where images and videos are indexed into a database using a vector of features extracted from the image or video. In the retrieval stage, the similarity between the features of the query image and the stored feature vectors is determined. That means that computing the similarity between two images or videos can be transformed into the problem of computing the similarity between two feature vectors [Lee et al. 2004]. Hence, the large computational cost associated with CBIVR systems is related to matching algorithms for feature vectors, because there are many feature vectors from different images and videos in the feature database.

Histogram Euclidean distance (Equation 6) and bin-to-bin difference (b2b) (Equation 7) are common similarity measurements in CBIVR systems [Deb 2005]. 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.

$$d^2(h_1, h_2) = \sum_{i=0}^{2^n - 1} (h_1[i] - h_2[i])^2. \tag{6}$$

$$fd_{b2b}(h_1, h_2) = \frac{\sum_{i=0}^{2^n - 1} |(h_1[i] - h_2[i]|)}{N}. \tag{7}$$

The size of a histogram depends on the number of bits in each pixel. If we suppose a pixel depth of n bits, the pixel values will be between 0 and $2^n - 1$, and the histogram will have 2^n elements.

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

4.3 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 10a

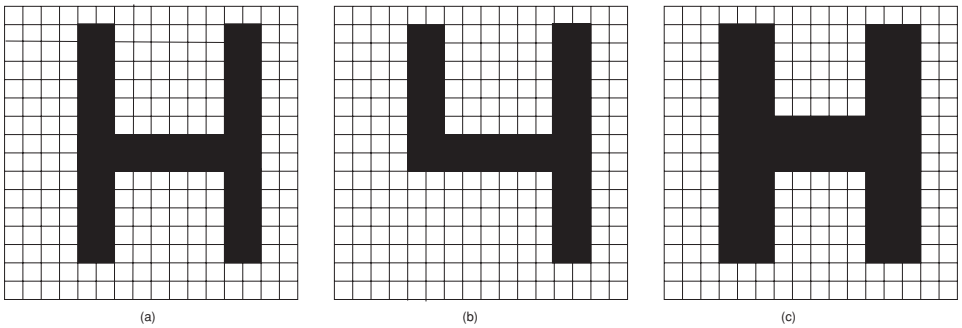

Fig. 10. Similar and dissimilar images.

and b is less than the Euclidean distance of a and c, even though Figure 10a is more similar to Figure 10c 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. For instance, the MPEG-2 encoding offers varieties of block matching, involving half-pixel interpolation. The original MPEG-2 standard first performs interpolation, and then computes the sum of absolute differences. To consider relationships between pixels in this paper, we implement horizontal and vertical interpolation.

5. SIMD IMPLEMENTATION OF KERNELS

In this section, we discuss in detail the SIMD implementations of the color space conversion and similarity measurement functions. The SIMD implementations of other kernels can be found in previous papers [Shahbahrami et al. 2006b, 2006c].

5.1 SIMD Implementation of Color Space Conversion

In this section, we discuss in detail the SIMD implementation of RGB-to-YCbCr color space conversion using the MMX and MMMX architectures.

The RGB values are usually in the band-interleaved format. Because of this, a straightforward MMX implementation of the RGB-to-YCbCr kernel is not efficient for the following reasons. First, image pixels must be unpacked from unsigned byte to 16 bits and vice versa, because of the mismatch between the storage and the computational format. Second, there are four 16-bit subwords in each MMX register and three R, G, and B values for each pixel. This implies that one of the subwords (a quarter of the processing capacity) will be unused. Third, there is no instruction in the MMX ISA that adds adjacent pixels. To synthesize this operation, we have to use many shift instructions and basically perform scalar addition. In addition, there are unaligned memory accesses, because, in each loop iteration, two pixels are processed and their starting address is not necessarily a multiple of 8.

To efficiently implement this kernel, we must first change from the band-interleaved to the band-separated format using rearrangement instructions, since experimental results on an actual machine show that the MMX implementation using the band-separated format is 4.20 times faster than the straightforward MMX implementation for an image of size 576×768. We use the faster method as the reference.

The MMX implementation using the band-separated format consists of the following stages:

1. Load the RGB values of eight pixels into the media register file (three instructions).
2. Conversion from band-interleaved to band-separated format using rearrangement instructions (35 instructions).
3. Unpack the packed byte data types to packed 16-bit word data types (six instructions).
4. Shift the RGB values to the left by 7 bits (six instructions).
5. Convert from RGB to YCbCr using 16-bit packed multiplication and addition instructions (51 instructions).
6. Truncate the results by shifting them to the right by 6 bits (six instructions).
7. Pack the unpacked results and store in memory (12 instructions).

This MMX implementation is also not very efficient, because many rearrangement and data-type conversion instructions are required. For instance, 35 instructions are needed to convert 8 pixels from the band-interleaved to the band-separated format. Figure 11 shows a part of the MMX code that achieves this rearrangement. It can be seen that many unpack, shift, and data-transfer instructions are required to achieve this. As a result, both MMX implementations are inefficient.

In the MMMX implementation of the RGB-to-YCbCr kernel, because of the MRF, changing from the band-interleaved to the band-separated format is not needed, as was illustrated in Figure 5. In addition, the data-type conversion instructions are avoided and eight-way parallelism is provided using the extended-subwords technique.

5.2 SIMD Implementation of Similarity Measurements

In this section, we explain the SIMD implementations of the SAD and the SAD with interpolation functions.

As mentioned in Section 1, there are some SPIs for the SAD function, for example, the `psadbw` instruction [Raman et al. 2000]. A 64-bit `psadbw` instruction consists of three steps: (1) calculate eight 8-bit differences between the elements, (2) calculate the absolute value of the differences, and (3) perform three cascaded summations. The code in Figure 12 depicts the MMX/SSE implementation of the motion-estimation kernel for two 16×16 blocks using the `psadbw` instruction.

One of the reasons why the `psadbw` instruction provides a significant performance benefit is that the 9-bit differences cannot be stored in the 8-bit subwords.

movq	mm0, (RGB)	; mm0 =	g3	r3	b2	g2	r2	b1	g1	r1
movq	mm2,8(RGB)	; mm2 =	r6	b5	g5	r5	b4	g4	r4	b3
movq	mm1,16(RGB)	; mm1 =	b8	g8	r8	b7	g7	r7	b6	g6
movq	mm3, mm0	; mm3 =	g3	r3	b2	g2	r2	b1	g1	r1
movq	mm4, mm0	; mm4 =	g3	r3	b2	g2	r2	b1	g1	r1
psrlq	mm3, 24	; mm3 =	0	0	0	g3	r3	b2	g2	r2
punpcklbw	mm4, mm3	; mm4 =	r3	r2	b2	b1	g2	g1	r2	r1
movq	mm6, mm2	; mm6 =	r6	b5	g5	r5	b4	g4	r4	b3
movq	mm7, mm2	; mm7 =	r6	b5	g5	r5	b4	g4	r4	b3
psrlq	mm3, 24	; mm3 =	0	0	0	0	0	0	g3	r3
psllq	mm6, 16	; mm6 =	g5	r5	b4	g4	r4	b3	0	0
psrlq	mm7, 8	; mm7 =	0	r6	b5	g5	r5	b4	g4	r4
por	mm6, mm3	; mm6 =	g5	r5	b4	g4	r4	b3	g3	r3
punpcklbw	mm6, mm7	; mm6 =	r5	r4	b4	b3	g4	g3	r4	r3
movq	mm3, mm4	; mm3 =	r3	r2	b2	b1	g2	g1	r2	r1
punpcklwd	mm4, mm6	; mm4 =	g4	g3	g2	g1	r4	r3	r2	r1
punpckhwd	mm3, mm6	; mm3 =	r5	r4	r3	r2	b4	b3	b2	b1
movq	mm0, mm2	; mm0 =	r6	b5	g5	r5	b4	g4	r4	b3
movq	mm6, mm1	; mm6 =	b8	g8	r8	b7	g7	r7	b6	g6
psrlq	mm2, 32	; mm2 =	0	0	0	0	r6	b5	g5	r5
psrlq	mm0, 56	; mm0 =	0	0	0	0	0	0	0	r6
psllq	mm6, 8	; mm6 =	g8	r8	b7	g7	r7	b6	g6	0
por	mm0, mm6	; mm0 =	g8	r8	b7	g7	r7	b6	g6	r6
punpcklbw	mm2, mm0	; mm2 =	r7	r6	b6	b5	g6	g5	r6	r5
movq	mm0, mm1	; mm0 =	b8	g8	r8	b7	g7	r7	b6	g6
psrlq	mm1, 16	; mm1 =	0	0	b8	g8	r8	b7	g7	r7
psrlq	mm0, 40	; mm0 =	0	0	0	0	0	b8	g8	r8
punpcklbw	mm1, mm0	; mm1 =	0	r8	b8	b7	g8	g7	r8	r7
movq	mm6, mm2	; mm6 =	r7	r6	b6	b5	g6	g5	r6	r5
punpcklwd	mm2, mm1	; mm2 =	g8	g7	g6	g5	r8	r7	r6	r5
punpckhwd	mm6, mm1	; mm6 =	0	r8	r7	r6	b8	b7	b6	b5
movq	mm1, mm4	; mm1 =	g4	g3	g2	g1	r4	r3	r2	r1
punpckldq	mm1, mm2	; mm1 =	r8	r7	r6	r5	r4	r3	r2	r1
punpckhdq	mm4, mm2	; mm4 =	g8	g7	g6	g5	g4	g3	g2	g1
punpckldq	mm3, mm6	; mm3 =	b8	b7	b6	b5	b4	b3	b2	b1

Fig. 11. The MMX instructions needed to convert RGB values from the band-interleaved to the band-separated format.

```
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     psadbw   mm1 , mm2
9     psadbw   mm3 , mm4
10    paddd    mm1 , mm3
11    paddd    mm5 , mm1
12    add      blk1, 16
13    add      blk2, 16
14    dec      eax
15    jnz      .loop
```

Fig. 12. The MMX/SSE program of the SAD function.

Furthermore, there are no instructions to sum all the elements in a register or to add adjacent elements. In the MMMX architecture the 9-bit differences can be stored in the 12-bit subwords. Moreover, it provides instructions to add adjacent, elements, which can most lightly be performed in a single cycle. In other words, we have implemented SIMD instructions to replace the `psadbw` instruction, which are more general purpose and can be used in many multimedia kernels and also in other similarity measurements. The `psadbw` instruction can be synthesized using a small number of such general-purpose SIMD instructions with only a small performance degradation. Figure 13 shows how the SAD function can be implemented using MMMX instructions.

The two `psadbw` instructions have been synthesized by the SIMD instructions `fsub12`, `fneg12`, `fmax12`, `fadd12`, and `fsum{12,24,48}`, which are more general purpose than them. In order to provide eight-way parallelism, we divided a 16×16 block into two 8×16 blocks. In the first iteration of the outer loop, the SAD function of the first 8×16 block is calculated and in the next iteration, the SAD function of the other 8×16 block is performed. Finally, the results are accumulated into one register using the `fsum{12,24,48}` instructions.

As already mentioned is Section 4.3, one way to consider the relationship between image pixels is averaging. For this, the SSE ISA provides a special averaging instruction `pavgb` for 8-bit subwords. This instruction averages two pixels; unsigned values are rounded up to the nearest integer. However, averaging four pixels using horizontal and vertical interpolation may produce an error of 1 when performing three average operations as follows $pavgb(x, y, z, t) = pavgb[pavgb(x, y), pavgb(z, t)]$. To avoid this error in the MMX/SSE implementation, we use 16-bit operations using pack/unpack instructions. Figures 14 and 15 show a part of the MMX/SSE and MMMX implementations of the SAD function with horizontal and vertical averaging, respectively.

The sum of four neighboring pixels is larger than 8 bits. Hence, in the MMX/SSE implementation, we unpack the 8-bit data type to 16 bits. This means that four-way parallelism is provided, as depicted in Figure 14. The MMMX

```
1     mov        ecx , 2
2   loop2:
3     fxor       3mx5, 3mx5
4     mov        eax , 8
5   loop1:
6     fld8u12s   3mx1, [blk1]
7     fld8u12s   3mx2, [blk2]
8     fld8u12s   3mx3, [blk1+8]
9     fld8u12s   3mx4, [blk2+8]
10    fsub12     3mx1, 3mx2
11    fneg12     3mx7, 3mx1
12    fmax12     3mx1, 3mx7
13    fsub12     3mx3, 3mx4
14    fneg12     3mx7, 3mx3
15    fmax12     3mx3, 3mx7
16    fadd12     3mx1, 3mx3
17    fadd12     3mx5, 3mx1
18    add        blk1, 16
19    add        blk2, 16
20    dec        eax
21    jnz        .loop1
22    fsum12     3mx5
23    fsum24     3mx5
24    fsum48     3mx5
25    fadd96     3mx6 , 3mx5
26    dec        ecx
27    jnz        .loop2
```

Fig. 13. The MMMX implementation of the SAD function.

implementation, on the other hand, employs eight-way parallelism, because 12 bit is sufficient for the sum of four pixels.

6. PERFORMANCE EVALUATION

In this section, we evaluate the MMMX architecture by comparing the performance obtained for the MMMX implementation to the performance of scalar and MMX implementations on different out-of-order processors. For the RGB-to-YCbCr kernel, the performance of the band-interleaved MMMX code is compared to the band-separated MMX code.

6.1 Evaluation Environment

In order to evaluate the MMMX architecture, we have used the `sim-outorder` simulator of the SimpleScalar toolset [Austin et al. 2002]. `sim-outorder` is a detailed, execution-driven simulator that supports out-of-order issue and execution. We have used the PISA ISA, which consists of 64-bit instructions. Each instruction contains a 16-bit annotate field, which can be used to synthesize new instructions without having to change and recompile the assembler. We have synthesized MMX/SSE and MMMX instructions using this annotate field. More detail about our extension to the SimpleScalar toolset can be found in Juurlink et al. [2007].

```
1     mov         eax, 16
2   loop:
3     ; Pixels 0..7
4     movq        mm1, [blk1]
5     movq        mm3, [blk1+16]
6     movq        mm2, mm1
7     movq        mm4, mm3
8     punpcklbw   mm1, mm0
9     punpcklbw   mm3, mm0
10    movd        mm5, [blk1+1]
11    movd        mm6, [blk1+17]
12    punpcklbw   mm5, mm0
13    punpcklbw   mm6, mm0
14    .
15    .
16    packuswb    mm1, mm2
17    psadbw      mm1, [blk2]
18    ; Pixels 8..F
19    movq        mm1, [blk1+8]
20    movq        mm3, [blk1+24]
21    movq        mm2, mm1
22    movq        mm4, mm3
23    punpcklbw   mm1, mm0
24    punpcklbw   mm3, mm0
25    movd        mm5, [blk1+9]
26    movd        mm6, [blk1+25]
27    punpcklbw   mm5, mm0
28    punpcklbw   mm6, mm0
29    .
30    .
31    add         blk1, 16
32    add         blk2, 16
33    dec         eax
34    jnz         .loop
```

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

The main objective is to compare the performance of an SIMD architecture without extended subwords and the MRF to the same architecture with these features. We 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 IV. We modeled processors by varying the issue width from 1 to 4 instructions per cycle. 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. Register renaming and reordering of instructions are done using this unit. The goal of the RUU is to always have enough instructions ready to feed the available functional units. The default value of 16 is insufficient to find many independent instructions. We, therefore, used an RUU size of 64 instead.

The latency and throughput of SIMD instructions are set equal to the latency and throughput of the corresponding scalar instructions. This is a conservative

```
1     mov       eax , 8
2   loop:
3     fld8u12   3mx1, [blk1]
4     fld8u12   3mx2, [blk1+8]
5     fld8u12   3mx3, [blk1+16]
6     fld8u12   3mx4, [blk1+24]
7     fadd12    3mx1, 3mx3
8     fadd12    3mx2, 3mx4
9     fld8u12   3mx3, [blk1+1]
10    fld8u12   3mx4, [blk1+9]
11    fld8u12   3mx5, [blk1+17]
12    fld8u12   3mx6, [blk1+25]
13    fadd12    3mx3, 3mx5
14    fadd12    3mx4, 3mx6
15    fadd12    3mx1, 3mx3
16    fadd12    3mx2, 3mx4
17    fsra12    3mx1, 2
18    fsra12    3mx2, 2
19    fld8u12   3mx3, [blk2]
20    fld8u12   3mx4, [blk2+8]
21    .
22    .
23    add       blk1, 16
24    add       blk2, 16
25    dec       eax
26    jnz       .loop
```

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

Table IV. Processor Configuration

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

assumption given that the SIMD instructions perform the same operation, but on narrower data types. In addition, both latency and throughput of the `fsum` instructions are set to 1, while the latency and throughput of the `psadbw` instruction are set to 4 and 1, respectively—the same as in the Pentium 4 processor.

In the experiments, three programs have been implemented and simulated using the SimpleScalar simulator for each kernel. Each program

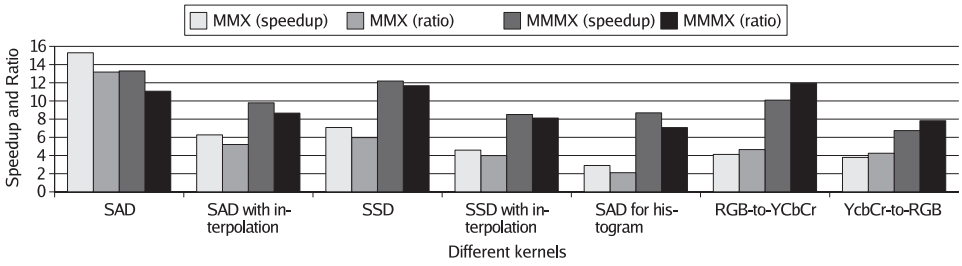

Fig. 16. Speedup of MMX and MMMX over the C implementation as well as the ratio of committed instructions (C implementation to MMX and MMMX) for different kernels on the single issue processor.

consists of three parts. One part is for reading the image, the second part is the computational kernel, and the last part is for storing the transformed image. One program is completely written in C. It was compiled using the gcc compiler targeted to the SimpleScalar PISA with optimization level $-O2$. The reading and storing parts of the other two programs were also written in C, but the second part was implemented using MMX/SSE and MMMX. These programs will be referred to as C, MMX, and MMMX for each kernel.

In addition, a whole image size has been used as input for some kernels. For example, we have implemented the full search algorithm for motion estimation on an image size of quarter common intermediate format (QCIF). The QCIF has a size of 144×176. In order to determine the motion vectors for the reference blocks in the current frame, we have used a macroblock of 8×8 pixel region as the basic block and a search range of ± 16 in the process of motion estimation.

6.2 Performance Evaluation Results

Figure 16 depicts the speedup of MMX and MMMX over the scalar implementation, as well as the ratio of committed instructions (C implementation to MMX and MMMX). For all kernels, the speedup of MMX and MMMX is significantly larger than one. This is because of the following reasons. First, MMX, as well as MMMX, exploit DLP. Specifically, eight-way parallelism is used in the MMMX code because of the extended subwords technique. In the MMX code, the eight-way parallel `psadbw` instruction is employed for the SAD function and four-way parallelism is employed in other workloads. Second, the number of loop-overhead instructions has been reduced. Both MMX and MMMX reduce a significant number of loop-overhead instructions, which increment or decrement index and address values. Third, both MMX and MMMX codes use short vector load and store instructions (8 bytes) compared to the C implementation that load one unsigned char in each load instruction.

The figure also shows that MMMX performs better than MMX for all kernels except SAD. The speedup of MMMX is between 8.51 and 13.30, while the speedup for MMX is between 4.59 and 15.30. The most important reason why MMMX improves performance compared to MMX is that it needs to execute fewer instructions than MMX. In the SAD kernel, on the other hand, MMMX

Table V. Ratio of Total Instructions, SIMD Instructions, Scalar, and SIMD ld/st Instructions of the MMX to the MMMX Implementation[a]

Kernels	Ratio of total instructions	Ratio of SIMD instructions	Ratio of scalar instructions	Ratio of SIMD ld/st instructions
SAD	$\frac{6499}{8037} = 0.81$	$\frac{3808}{5346} = 0.71$	$\frac{1314}{1314} = 1$	$\frac{1377}{1377} = 1$
SAD with interpolation	$\frac{23346}{12978} = 1.8$	$\frac{17415}{8343} = 2.1$	$\frac{1314}{1314} = 1$	$\frac{4617}{3321} = 1.4$
SSD	$\frac{14517}{6741} = 2.2$	$\frac{11583}{3807} = 3$	$\frac{1557}{1557} = 1$	$\frac{1377}{1377} = 1$
SSD with interpolation	$\frac{28206}{13302} = 2.1$	$\frac{21951}{8343} = 2.6$	$\frac{1638}{1638} = 1$	$\frac{4617}{3321} = 1.4$
SAD for histogram	$\frac{136}{40} = 3.4$	$\frac{88}{16} = 5.5$	$\frac{32}{16} = 2$	$\frac{16}{8} = 2$
RGB-to-YCbCr	$\frac{300}{115} = 2.6$	$\frac{208}{71} = 2.93$	$\frac{8}{4} = 2$	$\frac{84}{40} = 2.1$
YCbCr-to-RGB	$\frac{178}{92} = 1.9$	$\frac{124}{56} = 2.21$	$\frac{6}{6} = 1$	$\frac{48}{30} = 1.6$

[a]The numbers have been obtained for either to search one current block in a window search of the reference frame in the full-search algorithm or to process 16 pixels in the CSCs and a 16 elements array for SAD that is used for histogram similarity.

needs to execute more instructions than MMX. As Figure 16 shows, the ratio of committed instructions for the SAD kernel is 13.18 and 11.07 using MMX and MMMX, respectively. An SPI has been used in the MMX implementation of the SAD function, while in the MMMX implementation this SPI has been synthesized by a few general-purpose SIMD instructions. Thus, the SPI `psadbw` provides little benefit if extended subwords are supported.

Table V depicts the ratio of total instructions, SIMD instructions, scalar, and SIMD load/store (ld/st) instructions of the MMX implementation to the MMMX implementation. SIMD instructions consist of SIMD ALU/MULT and SIMD overhead, data-type conversions, and rearrangement instructions. For SAD, SAD with interpolation, SSD, and SSD with interpolation, these numbers correspond to a full search of the best matching block in a search window of 16×16 pixels. For the CSCs, these numbers correspond to processing 16 pixels and, for the histogram SAD, the histogram size is 16 elements. As this table shows, the MMMX architecture reduces the dynamic number of instructions by up to 3.40 over the MMX architecture. This is the main reason why MMMX provides a speedup of up to 3.00 over MMX.

This reduction of the dynamic instruction count is because of extended subwords and the MRF. The MMMX implementation can employ eight-way parallel SIMD instructions, while MMX can employ only four- or two-way parallelism in all kernels, except the SAD. In other words, MMMX can pack more arithmetic and logical operations into a single SIMD instruction. In addition, MMMX avoids SIMD data-type conversion and rearrangement instructions.

Table V shows that most of the reduction is because of the reduction of the number of SIMD instructions. This means that MMMX improves the performance of SIMD instructions more than the performance of other parts. Reduction of the scalar operations and SIMD ld/st instructions is much less than the

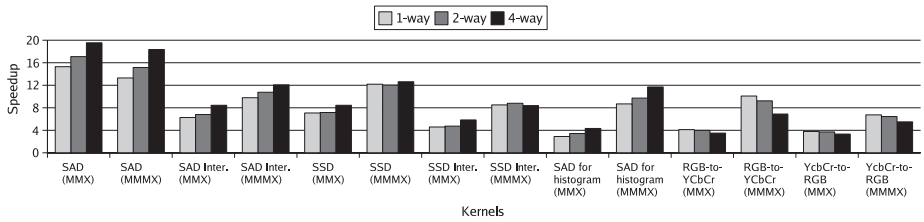

Fig. 17. Speedup of MMX and MMMX over C implementation for different issue widths using out-of-order execution. The speedup is relative to the time taken by the scalar implementation when executed on the processor with the same issue width.

reduction of SIMD instructions. This is because some parts of the algorithms cannot be vectorized and, therefore, scalar instructions have to be used. The scalar instructions are used for conditional operations, boundary checking, updating the pointers, and incrementing or decrementing index and address values. In addition, both architectures load or store 8 bytes simultaneously. This means that, in some cases, the number of ld/st instructions is the same in both architectures.

Figure 17 depicts the effect of increasing the issue width. It shows the speedup of the MMX and MMMX implementations on out-of-order processors with different issue widths. The speedup is relative to the time taken by the scalar implementation when executed on the processor with the same issue width. For the similarity measurements, increasing the issue width increases the relative speedup. For the CSCs, however, increasing the issue width decreases the speedup over the scalar implementation for the same issue width. The main reason is that there are many more scalar instructions in the similarity measurements functions than in the CSCs, as shown in Table V. As depicted in Table IV, when the issue width is doubled, the number of SIMD and scalar functional units is scaled accordingly. This means that with increasing the issue width for similarity measurements more instructions can be executed than for CSCs.

Figure 17 also shows that a slightly higher speedup is achieved on the higher issue width processors than on the lower-issue width processors for the SAD function without interpolation compared to the SAD with interpolation and SSD functions. This is because, as Table V shows, many more ld/st instructions have been used for pixel averaging in the SSD and SAD functions with interpolation than in the SAD function without interpolation. MMMX mainly reduces the number of SIMD instructions and not the number of ld/st instructions.

The speedup of the higher-issue width processors is generally not significantly higher than the speedup of the lower-issue width processors for the similarity measurements algorithms. Again we emphasize that the speedup is relative to the scalar implementation executed on the same issue width. In addition, for the CSCs the speedup on the four-way processor is slightly lower than the speedup on the one- and two-way processors for both architectures. For instance, on the one-way processor, MMMX obtains an average speedup of 8.42 over the C implementations for CSCs. On the four-way processor, the average speedup over the C implementation running on the four-way processor is 6.18.

The most important reason is that the scalar implementations achieve higher instructions per cycle. In other words, the scalar implementations benefit more from a higher-issue width than the MMX and MMMX implementations. Because MMX and MMMX pack several independent operations in a single SIMD instruction (MMMX even more than MMX), the distance between dependent instructions decreases. In other words, when the C implementation is executed the available DLP is exploited as instruction-level parallelism. Furthermore, because the C implementation executes more loop iterations than the MMX and MMMX implementations, the branch prediction accuracy is higher.

As already discussed, MMMX reduces the number of scalar and SIMD ld/st instructions much less than the number of SIMD instructions. On one hand, the scalar instructions are necessary, while on the other, these instructions prevent larger performance improvements. SIMD ld/st instructions consume an average of 21 and 28% of the total instructions in the MMX and MMMX implementations, respectively. In other words, the percentage usage of SIMD ld/st instructions in MMMX is higher than MMX. In the following section, we reduce the number of SIMD ld/st instructions by increasing the number of registers.

6.3 Impact of the Number of Registers

It is well-known that for ISA legacy reasons, MMX has only eight architectural registers. Because of this, the constants needed for performing CSCs and the 8×8 block of the current frame that is used by the full-search algorithm cannot be kept in registers, but have to be reloaded from memory in each loop iteration. Although the constants and the current block will be found in the cache most of the time, the number of SIMD ld/st instructions is relatively large compared to the number of total instructions. A larger register file would allow to keep intermediate and constant values in the media registers during the entire execution of a program. Therefore, in this section, we consider the effect of adding more registers to the MMMX architecture.

We have found that 13 extra media registers are sufficient to keep the critical data in the register file. In the RGB-to-YCbCr kernel, 11 of these registers are used to hold constants and 2 to hold intermediate results. Since two of the constant coefficients in the YCbCr-to-RGB kernel are zero and three of them are the same, for this kernel only 9 additional registers are needed. For the similarity measurements kernels we employ 8 extra media registers. Thus, an entire 8×8 candidate block can be stored in these 8 extra media registers, as depicted in Figure 18.

Figure 19 illustrates the speedup of MMMX with 8 registers (MMMX-8) and MMMX with 13 extra registers (MMMX-13) over MMX, as well as the ratio of committed instructions (MMX implementation to MMMX) on the single-issue processor. MMMX-13 yields speedups ranging from 1.37 to 3.64. Furthermore, the performance improvement of MMMX-13 over MMMX-8 ranges from 1.38 to 1.57, and the ratio of committed instructions (MMMX-8 implementation to MMMX-13) ranges from 1.22 to 1.56. Again, the main reason for these performance improvements is the reduced number of instructions that need to be

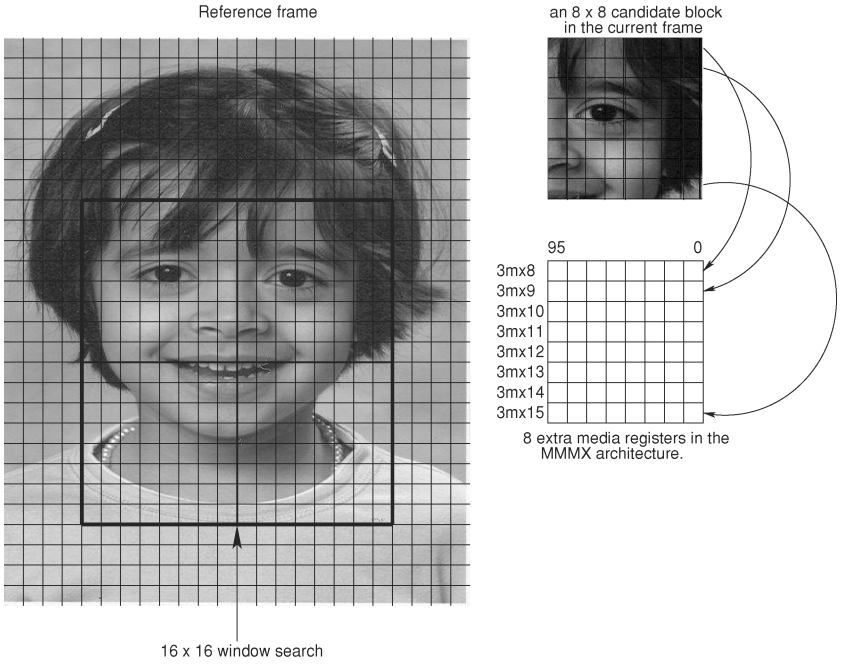

Fig. 18. The candidate block of the current frame can be stored in eight media registers to calculate the motion vector at each 16 × 16 window search of the reference frame.

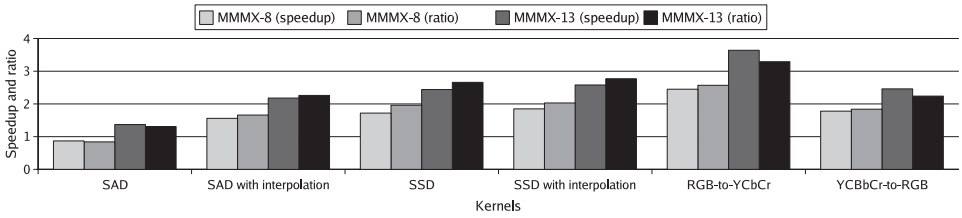

Fig. 19. Speedup of MMMX with 8 registers (MMMX-8) and MMMX with 13 extra registers (MMMX-13) over MMX (8 registers) as well as the ratio of committed instructions (MMX implementation to MMMX) on the single-issue processor.

executed. Because the data that is needed very often can be kept in registers, fewer ld/st instructions need to be executed. The largest performance improvement is achieved for the SAD function. For this kernel the speedup is 0.87 and 1.37 using MMMX-8 and MMMX-13, respectively. Although the MMX code that uses the SAD SPI is faster than the MMMX-8 implementation, the MMMX-13 code yields more performance.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have shown that the MMMX architecture, which features extended subwords and a matrix register file (MRF), can also be used to accelerate color space conversions and different similarity measurement functions.

The MRF was proposed to improve the efficiency of 2-D block-based algorithms, which are used in many media applications. In this paper, it is shown, however, that it can also be used to rearrange strided data, as is needed in color space conversion. In this way, many rearrangement instructions needed by conventional SIMD extensions are eliminated. In addition, it was shown that if extended subwords are supported and a few general SIMD instructions, then special-purpose instructions, such as the SAD instruction, provide little additional benefit. Specifically, the speedup of the MMX/SSE implementation of the SAD kernel that uses the SPI `psadbw` is only 15% faster than the MMMX implementation. Furthermore, the usefulness of the SPI `psadw` is limited, since it can only be used for the SAD kernel. It can, for example, not be used to calculate the SAD of two histograms, since the histogram elements are wider than 8 bits. In other words, we have shown that the MMMX architecture is very versatile. We remark that we do not claim that SPIs are not useful. However, when extended subwords are supported, their usefulness is limited.

Results have been obtained by synthesizing the MMX and MMMX instruction sets in the `sim-outorder` simulator of the SimpleScalar toolset. They show that MMMX improves performance compared to MMX by a factor of up to 3.0. The main reason for this performance improvement is the reduction of the dynamic number of instructions. The use of extended subwords avoids conversion overhead between different packed data types and, furthermore, allows more operations to be packed in a single SIMD instruction and the MRF avoids rearrangement overhead. The results also show that using, at most, 13 extra media registers yields an additional performance improvement ranging from 1.38 to 1.57. Although it is well-known that the small number of architectural registers is a limitation of the MMX architecture, it is somewhat surprising that using a larger register file can provide a speedup of up to 1.57.

As future work, we consider investigating ways to either reduce the number of scalar instructions or to overlap their execution with the execution of SIMD instructions. In addition, data accesses in many multimedia kernels, for example, motion estimation and compensation, are inherently misaligned. Therefore, new techniques are needed to improve the efficiency of misaligned accesses.

REFERENCES

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Comput. 35,* 2, 59–67.

BARON, M. 2005. Cortex-A8: High speed, low power. *Microprocessor Rep. 11,* 14, 1–6.

BARTKOWIAK, M. 2001. Optimizations of color transformation for real time video decoding. In *Proceedings of the EURASIP Conference on Digital Signal Processing for Multimedia Communications and Services*.

BENSAALI, F. AND AMIRA, A. 2005. Accelerating colour space conversion on reconfigurable hardware. *Image Vision Comput. 23*, 935–942.

CHATTERJI, S., NARAYANAN, M., DUELL, J., AND OLIKER, L. 2003. Performance evaluation of two emerging media processors: VIRAM and Imagine. In *Proceedings of the 14th IEEE International Symposium on Parallel and Distributed Processing*. 229–235.

DEB, S. 2005. *Video Data Management and Information Retrieval*. IRM Press, Hershey, Pennsylvania, USA.

DIEFENDORFF, K., DUBEY, P. K., HOCHSPRUNG, R., AND SCALES, H. 2000. AltiVec extension to powerPC accelerates media processing. *IEEE Micro 20,* 2, 85–95.

FLACHS, B., ASANO, S., DHONG, S. H., HOFSTEE, H. P., GERVAIS, G., KIM, R., LE, T., LIU, P., LEENSTRA, J., MICHAEL, J. L. B., OH, H. J., MUELLER, S. M., TAKAHASHI, O., HATAKEYAMA, A., WATANABE, Y., YANO, N., BROKENSHIRE, D. A., PEYRAVIAN, M., VANDUNG, T., AND IWATA, E. 2006. The microarchitecture of the synergistic processor for a cell processor. *IEEE J. Solid-State Circuits 41*, 63–70.

GOODACRE, J. AND SLOSS, A. N. 2005. Parallelism and the ARM instruction set architecture. *IEEE Comput. 38,* 7, 42–50.

GSCHWIND, M., HOFSTEE, H. P., FLACHS, B., HOPKINS, M., WATANABE, Y., AND YAMAZAKI, T. 2006. Synergistic processing in cell's multicore architecture. *IEEE Micro 26,* 2, 10–24.

GWENNAP, L. 1996. Digital, MIPS add multimedia extensions. *Microprocessor Rep. 10,* 15, 24–28.

HUANG, L., LAI, M., DAI, K., YUE, H., AND SHEN, L. 2007. Hardware support for arithmetic units of processor with multimedia extension. In *Proceedings of the IEEE International Conference on Multimedia and Ubiquitous Engineering*. 633–637.

IBM 2007. *Synergistic Processor Unit Instruction Set Architecture*. IBM. Version 1.2.

JENNINGS, M. D. AND CONTE, T. M. 1998. Subword extensions for video processing on mobile systems. *IEEE Concurrency 6,* 3, 13–16.

JUURLINK, B., BORODIN, D., MEEUWS, R. J., AALBERS, G. T., AND LEISINK, H. 2007. The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT). Available via `http://ce.et.tudelft.nl/~shahbahrami/`

KOZYRAKIS, C., GEBIS, J., MARTIN, D., WILLIAMS, S., MAVROIDIS, I., POPE, S., JONES, D., PATTERSON, D., AND YELICK, K. 2000. Vector IRAM: A media-oriented vector processor with embedded DRAM. In *Proceedings of the 12th International Conference on Hot Chips*.

KUHN, P. 1999. *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*. Kluwer Academic Publ. Boston, MA.

LARSEN, S. AND AMARASINGHE, S. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 145–156.

LEE, A. J. T., HONG, R. W., AND CHANG, M. F. 2004. An approach to content-based video retrieval. In *Proceedings of the IEEE International Conference on Multimedia and Expo*. Vol. 1. 273–276.

LEE, J., VIJAYKRISHNAN, N., IRWIN, M. J., AND WOLF, W. 2004. An architecture for motion estimation in the transform domain. In *Proceedings of the 17th IEEE International Conference on VLSI Design*.

LEE, R. B. AND SMITH, M. D. 1996. Media processing: A new design target. *IEEE Micro 16,* 4, 6–9.

MORENO, J. H., ZYUBAN, V., SHVADRON, U., NEESER, F. D., DERBY, J. H., WARE, M. S., KAILAS, K., ZAKS, A., GEVA, A., BEN-DAVID, S., ASAAD, S. W., FOX, T. W., LITTRELL, D., BIBERSTEIN, M., NAISHLOS, D., AND HUNTER, H. 2003. An innovative low-power high-performance programmable signal processor for digital communications. *IBM J. Res. Develop. 47,* 2/3, 299–326.

Motorola Inc. 1998. *AltiVec Technology Programming Environments Manual*. Motorola Inc. Rev.0.1.

NAISHLOS, D., BIBERSTEIN, M., DAVID, S. B., AND ZAKS, A. 2003. Vectorizing for a SIMdD DSP Architecture. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 2–11.

PELEG, A., WILJIE, S., AND WEISER, U. 1997. Intel MMX for Multimedia PCs. *Commun. ACM 40,* 1, 24–38.

POYNTON, C. 1996. *A Technical Introduction to Digital Video*. Wiley, New York.

RABBANI, M. AND JONES, P. W. 1991. *Digital Image Compression Techniques*. Bellinghan, Washington.

RAMAN, S. K., PENTKOVSKI, V., AND KESHAVA, J. 2000. Implementing streaming SIMD extensions on the Pentium 3 processor. *IEEE Micro 20,* 4, 47–57.

SESHAN, N. 1998. High VelociTI Processing. *IEEE Signal Processing Mag. 15,* 2, 86–101.

SHAHBAHRAMI, A., JUURLINK, B., BORODIN, D., AND VASSILIADIS, S. 2006a. Avoiding conversion and rearrangement overhead in SIMD architectures. *Intern. J. Parallel Programming 34,* 3, 237–260.

SHAHBAHRAMI, A., JUURLINK, B., AND VASSILIADIS, S. 2006b. Accelerating color space conversion using extended subwords and the matrix register file. In *Proceedings of the 8th IEEE International Symposium on Multimedia*. 37–46.

SHAHBAHRAMI, A., JUURLINK, B., AND VASSILIADIS, S. 2006c. Limitations of special-purpose instructions for similarity measurements in media SIMD extensions. In *Proceedings of the ACM*

International Conference on Compilers, Architecture and Synthesis for Embedded Systems. 293–303.

SHANABLEH, T. AND GHANBARI, M. 2000. Heterogeneous video transcoding to lower spatio-temporal resolutions and different encoding formats. *IEEE Trans. Multimedia 2,* 2, 101–110.

SLINGERLAND, N. AND SMITH, A. J. 2002. Measuring the performance of multimedia instruction sets. *IEEE Trans. Comput. 51,* 11, 1317–1332.

TAMHANKAR, A. AND RAO, K. R. 2003. An overview of H.264/MPEG-4 Part 10. In *Proceedings of the 4th International Conference on Video and Image Processing and Multimedia Communications*. 1–51.

Texas Instruments 2007. *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*. Texas Instruments. Literature Number: SPRU732D.

TREMBLAY, M., 0'CONNOR, J. M., NARAYANAN, V., AND HE, L. 1996. VIS speeds new media processing. *IEEE Micro 16,* 4, 10–20.

WANG, L., ZHANG, Y., AND FENG, J. 2005. On the euclidean distance of images. *IEEE Trans. Pattern Anal. Machine Intell. 27,* 8, 1334–1339.

ZHANG, D. AND LU, G. 2003. Evaluation of similarity measurement for image rretrieval. In *Proceedings of the IEEE International Conference on Neural Networks and Signal Processing*. Vol. 2. 928–931.

Received March 12, 2007; revised August 8, 2007; accepted November 30, 2007