Accelerating Color Space Conversion Using Extended Subwords and the Matrix Register File

Asadollah Shahbahrami Ben Juurlink Stamatis Vassiliadis
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.
E-mail: {shahbahrami,benj,stamatis}@ce.et.tudelft.nl

Abstract

Color space conversion is an important kernel in multimedia codecs such as JPEG and MPEG. When implemented using SIMD instructions, however, the performance improvement is often limited due to two reasons. First, corresponding color space components are stored at non-unit strides and, second, intermediate results can be larger than 8 bits. In this paper we show that extended subwords and the Matrix Register File (MRF) can be employed to mitigate these limitations. These techniques avoid rearrangement instructions and increase the number of subwords that are processed in parallel. Experimental results have been obtained by extending the SimpleScalar toolset. The results show that extended subwords and the MRF yield a speedup of up to 2.45x and 1.78x over MMX for the RGB-to-YCbCr and YCbCr-to-RGB kernels, respectively. Compared to C implementations, speedups of up to 10.09x and 6.74x, respectively, are obtained. Additionally, the results show that the speedup over MMX is higher for low issue rates. This means that extended subwords and the MRF are suitable techniques for embedded multimedia systems where high issue rates and out-of-order execution are too expensive. The results also show that using more registers improves performance substantially.

Keywords: Color space conversion, SIMD architectures, multimedia extensions.

1 Introduction

Many color images are represented using the RGB color space. RGB representations, however, are highly correlated, which implies that the RGB color space is not well-suited for independent coding [12]. Compression standards such as JPEG and MPEG, therefore, employ the YCbCr color space and need to transform from the RGB to the YCbCr color space (RGB-to-YCbCr) and vice versa (YCbCr-to-RGB). Color space conversion is a time-consuming operation. It has been reported that it consumes up to 40% of the processing time of a highly optimized decoder [3, 2]. Consequently, the performance of JPEG/MPEG coders/decoders (codecs) can be improved significantly by accelerating color space conversion.

Since color space conversions exhibit significant amounts of data-level parallelism, they could be implemented using Single-Instruction Multiple-Data (SIMD) instructions. Virtually all contemporary processors support such SIMD extensions. Examples include desktop and laptop processors such as the Pentium and the PowerPC, digital signal processors such as the Texas Instruments TMS320C64x families [15] and the Analog Devices TigerSharc processor [5], as well as processors mainly targeted at the embedded market such as ARM. SIMD instructions are particularly suited for embedded processors because they offer high performance at low energy consumption. Color space conversion, however, has certain characteristics which make it difficult to implement it efficiently using existing SIMD extensions such as MMX [8] and SSE [9].

First, the color components are usually stored as unsigned bytes but intermediate results require precision larger than 8-bit. This necessitates conversion overhead and reduces the number of color components that are processed in parallel by a single SIMD instruction by a factor of 2. Second, often the band interleaved format is used where the color components of each pixel are adjacent in memory. This implies that in order to employ SIMD instructions either one of the subwords in a register will be unused or the data has 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 register. In the first case a quarter of the processing capacity will be wasted and in the second case many overhead instructions need to be executed.

In this paper we propose the use of *extended subwords* and the *Matrix Register File* (MRF) to overcome the aforementioned limitations. Extended subwords use registers that are wider than the packed format used to store the data. Specifically, for every byte of data there are four extra bits. For example, four 16-bit values are represented in a register as four 24-bit quantities. The MRF allows to load data stored consecutively in memory to a column of the register file, where a column corresponds to corresponding subwords of different registers.

Given the importance of color space conversion and the difficulty of obtaining high performance using existing SIMD architectures, several (micro)architectural techniques have been proposed to accelerate it. Bensaali and Amira [3] and Sima et al. [13] proposed to employ reconfigurable hardware. Compared to ASICs, however, FPGAs are slower and consume more power. Agostini et al. have proposed a parallel and pipelined architecture for color space conversion from RGB to YCbCr. Kim [7] focused on two architectural enhancements for processing color images. First, a pixel truncation technique is considered that reduces data content in individual pixel word sizes for chrominance and luminance. Second, a color-aware multimedia instruction set extension (CAX) that supports parallel operations on two-packed, 16-bit (6:5:5) YCbCr data on a 32-bit datapath. That means that Kim used pixel-truncation technique. This technique reduces the accuracy of the color space conversions compared to 24-bit implementation (8-bit for each color component). Slingerland and Smith [14] proposed that SIMD architectures implement strided loads and stores to gather non-adjacent data elements as would be useful in color space conversion. Strided memory accesses would eliminate the overhead instructions, but such memory accesses are naturally slower than conventional memory access. Additionally, in [4] it was indicated that one reason for poor VIRAM memory performance for color space conversion is because of the strided memory accesses. The MRF, on the other hand, allows to read and write adjacent data elements.

In our previous work [11], we used extended subwords and matrix register file techniques to implement many 2D media kernels such as (I)DCT, pixel padding, and 2×2 Haar transform. Performance was evaluated by calculating the dynamic number of instructions, without using any simulators.

As previously mentioned, color space conversions kernels have certain features compared to other media kernels. Consequently, in this paper we want to implement and evaluate these important kernels using those two techniques. Our work differs from others in the following manner. First, we apply extended subwords and MRF techniques to accelerate color space conversions. We significantly extend to use these techniques by providing experimental results obtained by a detailed, cycle-accurate simulator. Our work shows that combining these two techniques eliminates rearrangement instructions that are needed for color space conversions on the existing SIMD processors. An interesting conclusion from this work is that the MRF can be used to reorganize strided data. Second, we have designed new SIMD instructions and evaluated them using SimpleScalar toolset for implementation of multimedia kernels such as color space conversions.

We refer to MMX enhanced with extended subwords and the MRF as Modified MMX (MMMX, pronounced as triple-MX). In this paper we demonstrate that MMMX provides significant speedups over standard MMX for color space conversions. Specifically, experimental results obtained using the `sim-outorder` simulator of the SimpleScalar toolset [1] show that:

- The MMMX implementation of RGB-to-YCbCr is up to 2.45x faster than the MMX implementation and MMMX is up to 1.78x faster than MMX for YCbCr-to-RGB.
- We also compare the results to C implementations. Compared to C implementations, the MMMX implementations are up 10.09x and 6.74x faster for RGB-to-YCbCr and YCbCr-to-RGB, respectively.
- The speedups achieved by MMMX are higher for low issue rates. This indicates that MMMX is suitable for embedded processors, since high issue rates are inappropriate for embedded processors.
- For historical reasons, MMX has only 8 architectural registers. Consequently, the constants needed for color space conversion cannot be kept in registers. Our simulation results show that adding more registers yields significant performance improvements.

This paper is organized as follows. Section 2 describes the MMMX architecture. Section 3 discusses the color space conversions and their fixed-point implementations in MMX and MMMX. The experimental results are presented in Section 4, and conclusions are drawn in Section 5.

2 MMMX Architecture

In this section we briefly describe the MMMX architecture which features extended subwords and matrix register file (MRF). In addition, we briefly discuss the area overhead and delay of the proposed techniques.

2.1 Extended Subwords and MRF

Multimedia data is typically stored in memory using a narrow data type, for example, 8-bit pixels or 16-bit audio samples. Furthermore, to prevent overflow while processing them, the data elements have to be unpacked after they

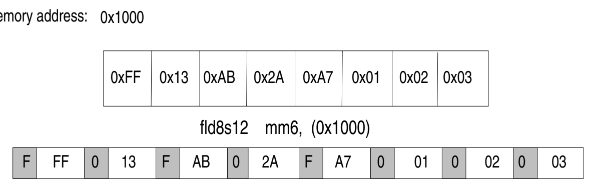
Memory address: 0x1000
0xFF 0x13 0xAB 0x2A 0xA7 0x01 0x02 0x03
fld8s12 mm6, (0x1000)
F FF 0 13 F AB 0 2A F A7 0 01 0 02 0 03

Figure 1. Illustration of the `fld8s12` instruction.

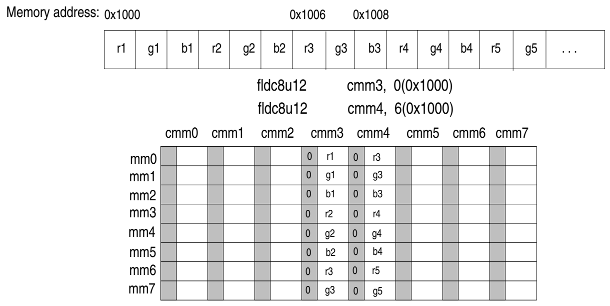
Memory address: 0x1000 0x1006 0x1008
r1 g1 b1 r2 g2 b2 r3 g3 b3 r4 g4 b4 r5 g5 ...
fldc8u12 cmm3, 0(0x1000)
fldc8u12 cmm4, 6(0x1000)
cmm0 cmm1 cmm2 cmm3 cmm4 cmm5 cmm6 cmm7
mm0 mm1 mm2 mm3 mm4 mm5 mm6 mm7
0 r1 0 r3
0 g1 0 g3
0 b1 0 b3
0 r2 0 r4
0 g2 0 g4
0 b2 0 b4
0 r3 0 r5
0 g3 0 g5

Figure 2. Illustration of the `fldc8u12` instruction.

have been loaded into a register. The conversion overhead and the fact that the unpacked data no longer fits in a single register limit the performance improvement that can be obtained by applying SIMD instructions. To avoid this conversion overhead and to increase parallelism, we employ extended subwords. 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 treaded either as a vector of eight 12-bit subwords, four 24-bit subwords, or two 48-bit quantities.

When loading data into an MMMX register, the subwords are automatically unpacked. For example, as illustrated in Figure 1, 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.

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, $MRF[i, j]$ corresponds to the jth subwords of register i. "Load-column" instructions load data elements stored consecutively in memory into a column of the MRF.

Figure 2 illustrates how the MRF can be used to reorganize the band interleaved data so that each register contains either red, green, or blue data. With eight load column instructions (`fldc8u12`) eight red, eight green, and eight blue values are loaded into each register. Each load column instruction loads eight bytes (three red, three green, and two blue) values as is shown in Figure 2. To provide correct arrangement of RGB values for each pixel in different subwords of the registers an offset, which is multiple of 6 bytes for each `fldc8u12` instruction is used.

Most MMMX instructions are direct counterparts of MMX instructions. For example, the MMMX instructions `fsub{12,24,48}` (subtraction of 12-, 24-, 48-bit subwords) and `fadd{12,24,48}` (addition of 12-, 24-, 48-bit subwords) correspond to the MMX instructions `psub{b,w,d}` and `padd{b,w,d}`, respectively. In contrast to MMX, however, MMMX does not support variants of these instructions that automatically saturate the results to the maximum value representable by the subword data type. They are not needed because, as explained above, load instructions automatically unpack and store instructions automatically pack and saturate. Another difference with MMX is that MMMX supports multiplication instructions for the smallest subword data type (12-bit). Specifically, the instructions `fmul12{l,h}` multiply the eight corresponding subwords of the source and destination operands and write the low-order (`fmul12l`) or high-order (`fmul12h`) 12 bits of the 24-bit product to the destination operand. In MMX/SSE, the packed multiply does not support the packed byte data type but only packed word (16-bit). We have used the `fmul12h` instruction in the fixed-point MMMX implementation of color space conversion.

	MMX/SSE (Int. part)	MMMX
Datapath	64-bit	96-bit
Size of register file	8 x 64-bit	8 x 96-bit
Shared with	Floating point reg.	Dedicated
Access to register file	row-wise	row- & col.-wise
# of partitioned ALU	8	8
Size of the integer sub.	8-, 16-, and 32-bit	12-, 24-,and 48-bit
High and low mul. ins.	16-bit	12-, 24-,and 48-bit
The size of MAC ope.	16-bit	12-, 24-,and 48-bit
MAC instruction	pmaddwd	fmadd{12,24,48}
Special purpose ins.	No/pavg{b,w},psadbw	No
Saturate Add/Sub.	Yes	No
Overhead instructions	packsswb,packssdw packuswb,punpckhbw punpckhwd,punpckhdq punpcklbw,punpcklwd punpckldq/pshufw	funpckl12 funpckl24 funpckh12 funpckh24

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

The main differences between MMX/SSE and MMMX architectures in integer part are in Table 1. As this table depicts there are SIMD instructions for different data types in the MMMX ISA. There are `funpckl{12,24}` and `funpckh{12,24}` instructions in the MMMX ISA for reshuffling the lower and higher packed subwords.

2.2 Area Overhead and Delay

In this section we discuss coarse estimates of the area overhead of extended subwords and wide partitioned ALUs using area estimates found in literature. Providing accurate estimates is beyond the scope of this paper and will be the subject of future work. We also briefly discuss the latency and throughput of SIMD instructions.

MMX and MMMX have only eight architectural SIMD registers, but we assume 32 64-bit physical (renaming) registers. Under this assumption, the total area overhead for extended subwords is 1Kb, which is very small. In a recent paper [16], an area breakdown of the TM3270 media processor, the latest TriMedia VLIW processor, has been presented. The register file constitutes about 12% of the total area. The TriMedia register file is relatively large, however, because it consists of 128 32-bit registers and has 10 32-bit read and 5 32-bit write ports. The area of a register file is the product of the number of registers, the number of bits per register, and the size of a register cell [10]. Furthermore, the size of a cell is proportional to $(3 + p)(4 + p)$, where p is the total number of ports. The most aggressive superscalar processor we have simulated issues at most 4 (SIMD) instructions per cycle and requires 8 read and 4 write ports. Since we assume 32 64-bit physical registers and require at most 12 ports, the MMX register file would constitute at most 4.2% of the total area. Under these assumptions, implementing extended subwords would require less than 2.1% of the total area.

A 32-bit ALU requires less than 0.05mm^2 in a 0.18μm CMOS process [10], so a coarse approximation of the area of a 64-bit partitioned ALU is 0.1mm^2 and of a 96-bit partitioned ALU 0.15mm^2. A relatively small integrated circuit is 1cm^2. Therefore, four 64-bit SIMD ALUs, as is assumed in the most aggressive MMX-enhanced superscalar, require less than 0.4% of the total area and four 96-bit SIMD ALUs take less than 0.6% of the total area. In other words, the area overhead of 96-bit SIMD units instead of 64-bit SIMD units is very small.

In our simulations we assume that the latency and throughput of SIMD instructions are equal to the latency and throughput of the corresponding scalar instructions. This is a conservative assumption given that the SIMD instructions perform the same operation but on narrower data types.

3 Color Space Conversion

In this section we discuss the RGB-to-YCbCr and YCbCr-to-RGB color space conversions and describe their implementations in MMX and MMMX.

Conversion between the YCbCr and RGB formats and vice versa can be accomplished with the following transformations.

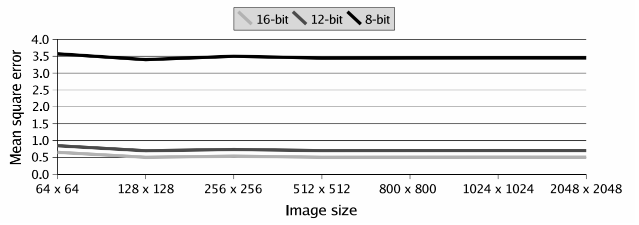

Figure 3. Mean square error for different bit widths in implementation of color space conversion.

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.256 & 0.502 & 0.098 \\ -0.148 & -0.290 & 0.438 \\ 0.438 & -0.366 & -0.071 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 16.5 \\ 128.5 \\ 128.5 \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.164 & 0.000 & 1.596 \\ 1.164 & -0.392 & -0.813 \\ 1.164 & 2.017 & 0.000 \end{pmatrix} \begin{pmatrix} Y - 16.5 \\ Cb - 128.5 \\ Cr - 128.5 \end{pmatrix} \tag{2}$$

In both equations, the coefficients have been rounded to three fractional decimal digits.

Color space conversions are usually defined using floating-point arithmetic but here, to avoid long-latency floating-point operations, we use fixed-point arithmetic. Specifically, for MMX we use 16-bit fixed-point numbers and for MMMX we approximate the color space conversion using 12-bit fixed-point arithmetic. To determine the accuracy of these approximations, we have performed two tests. 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. An 8-bit fixed-point implementation, on the other hand, has a maximum absolute error of 3. This indicates that the MMX implementation cannot employ 8-bit fixed-point arithmetic, since the accuracy is too low. Second, we have measured the mean square error (MSE) for real images such as “Lena” as well as randomly generated inputs. Figure 3 depicts the MSE of the 8-, 12-, and 16-bit implementations as a function of the image size. It shows that 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.

We now briefly sketch the MMX and MMMX implementations of the color space conversion kernels. We have developed two MMX implementations. In both implementations, the coefficients are represented as 16-bit fixed-point values because, as shown above, 8 bits are insufficient. In the first implementation, referred to as MMX-InnerProduct, color space conversion is carried out by computing the inner product of every row of the transformation matrix with the red, green, and blue data of every pixel. This implementation uses the packed multiply-add (pmaddwd) instruction to multiply the RGB values with the corresponding constants and to sum the two high-order and the two low-order words. Hereafter, however, the two doublewords packed in one register needed to be added. Since MMX does not provide such an instruction, unpack instructions are needed to place them in different registers. Another disadvantage of this implementation is that one of the subwords will be unused, because a pixel has three color components and there are four 16-bit subwords in an MMX register. In other words, 25% of the processing capability is wasted. Overall, this implementation needs to execute 312 instructions to process eight pixels.

In the second implementation, referred to as MMX-BandSeparated, we first rearrange the subwords contained in several registers so that the red data is contained in one register, the green data in another register, and the blue data in a third register. In other words, we first change from the band interleaved to the band separated format. Although this requires many rearrangement instructions, the main advantage of this scheme is that hereafter the packed multiply high instruction can be used to multiply, e.g., four red values with the same constant. This instruction simply ignores the 16 low-order bits of the 32-bit products, but these bits are not needed. This implementation needs to execute 124 instructions to process eight pixels.

We have measured the execution times of both the MMX-InnerProduct implementation and the MMX-BandSeparated implementation on the Pentium 4 using cycle counters [6]. For an image of size 576×768, the MMX-BandSeparated implementation is faster by a factor of 4.10x. This is mainly due to three reasons. First, as mentioned before, in the MMX-InnerProduct implementation a quarter of the processing capability is wasted. Second, the packed multiply-add produces two 32-bit results but of the 32 bits only the 16 higher-order bits are really needed. In the MMX-BandSeparated implementation the packed multiply high instruction is used, which simply discards the 16 lower-order bits. Because of this, the MMX-BandSeparated implementation can exploit more data-level parallelism during subsequent processing. Third, the MMX-InnerProduct implementation uses unaligned memory accesses because the color components of one pixel may be stored in memory locations which cross an address that is a multiple of 8. Although this could be avoided, this would incur significant rearrangement overhead. Because MMX-BandSeparated is much faster, we will use this implementation for our comparison with MMMX. We note, however, that this implementation is also not very efficient because many instructions are needed to go from the band interleaved to the band separated format. For example, to do this for 8 pixels requires 38 instructions.

The MMMX implementation is similar to the second MMX implementation. However, due to the MRF, changing from the band interleaved to the band separated format can be done with zero cost. Furthermore, in the MMMX implementation 12-bit arithmetic is used, which allows to perform 8 operations in a single SIMD instruction. The 12-bit constant coefficients are stored in memory as 16-bit values. During execution these coefficients are packed to 12-bit values. The total number of instructions needed to process 16 pixels using MMMX is 86, while the MMX-BandSeparated implementation requires 248 instructions.

4 Experimental Evaluation

In this section we evaluate MMMX by comparing the performance of the MMMX implementations of the color space conversion kernels to the performance of the C and MMX implementations.

4.1 Simulation Environment and Methodology

In order to evaluate MMMX, we have used the `sim-outorder` simulator of the SimpleScalar toolset [1], which 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, however, because our main objective is to compare the performance of an SIMD architecture without extended subwords and the MRF to the same architecture with these features. Furthermore, MMX instructions involving memory operands are translated to RISC-like micro-operations (μOPs) in the Pentium 4. We also remark that the correctness of the MMX and MMMX codes has been validated by comparing their output to the output of corresponding C programs.

Table 2 depicts the main parameters of the modeled processors. We modeled processors with issue widths varying 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 integer ALUs and SIMD ALUs 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. As remarked before, the latency and throughput of SIMD instructions are assumed to be equal to the latency and throughput of the corresponding scalar instructions. This is a realistic, even conservative

Parameter	
Issue width	1/2/4
Integer & SIMD ALU	1/2/4
Integer & SIMD MULT	1/2/4
L1 Instruction cache	32KB,direct-mapped, 64-byte lines, 1-cycle hit time
L1 Data cache	32KB,4-way set associative, 64-byte lines, 1-cycle hit time, LRU replacement
L2 Unified cache	256KB, 4-way set associative, 64-byte lines, 6-cycle hit time, LRU replacement
Main memory latency	18 cycles for the first chunk, 2 thereafter
Memory bus width	16 bytes
RUU entries	64
Load-store queue size	8
Execution	out-of-order

Table 2. Processor configuration.

assumption given that the SIMD instructions perform the same operation but on narrower data types.

We have implemented three versions of each kernel: one in C, one in assembly using MMX, and one using MMMX. The different versions of each kernel employ the same algorithm and data types. Each program consists of three parts, for reading the image, for performing color space conversion, and for storing the transformed image. Only the color space conversion has been implemented in MMX and MMMX and we report only the time taken by this part. All programs have been compiled using the Simplescalar compiler with optimization level *-O2*.

4.2 Experimental Results

Figure 4 depicts the speedup of the MMX and MMMX implementations of the RGB-to-YCbCr kernel for out-of-order processors with different issue widths. The speedup is relative to the time taken by the C implementation when executed on the processor with the same issue width. Figure 5 depicts the results for the YCbCr-to-RGB kernel. The results show that MMMX outperforms MMX significantly for both kernels. Specifically, for RGB-to-YCbCr, the speedup of MMMX over MMX is between 1.96x for the 4-way processor and 2.45x for the 1-way processor. For the YCbCr-to-RGB kernel, the speedup of MMMX over MMX is between 1.65x (4-way) and 1.78x (1-way). The reason that the speedups for RGB-to-YCbCr are higher than for YCbCr-to-RGB is that in the first kernel, the input is in the band interleaved format and needs to be changed to the band separated format, which requires many rearrangement instructions using MMX, while MMMX uses the MRF. In the second kernel, the input is already in the band separated format so that MMX does not incur this overhead.

An interesting observations is that MMMX exhibits higher speedup for lower issue rates. For instance, on the 1-way processor, MMMX obtains an average speedup of 8.42x over the C implementation. On the 4-way processor, the average speedup over the C implementation running on

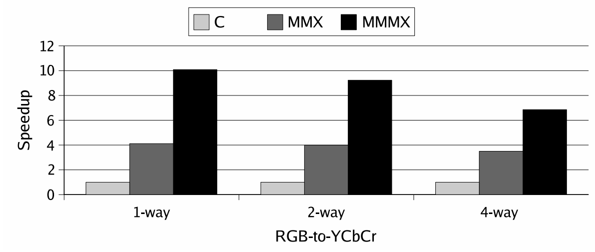

Figure 4. Speedup of MMX and MMMX over the C implementation of the RGB-to-YCbCr kernel for different issue widths using out-of-order execution.

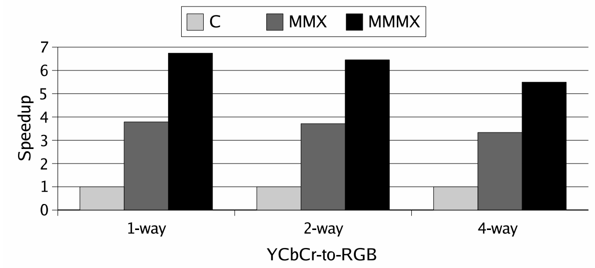

Figure 5. Speedup of MMX and MMMX over the C implementation of the YCbCr-to-RGB kernel for different issue widths using out-of-order execution.

the 4-way processor is 6.18x. This result demonstrates that MMMX is a suitable candidate for embedded multimedia systems where high issue rates and out-of-order execution are too expensive. The reason that MMMX exhibits higher speedup for lower issue rates is that the C implementation achieves higher IPCs, as shown in Figure 6 for the RGB-to-YCbCr kernel. 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 data-level parallelism 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.

The main reason why MMMX improves performance compared to MMX is that it needs to execute fewer instructions than MMX. To illustrate this, Figure 7 depicts both the speedup of MMX and MMMX over the C implementation as well as the instruction ratio, i.e., the ratio of the number of instructions committed by the C implementations of both kernels to the number of instructions committed by the MMX and MMMX implementations. These results are for the 1-way out-of-order processor. It can be seen that in general, the speedup is close to the instruction count reduction. In all cases, the speedup is slightly smaller than the ratio

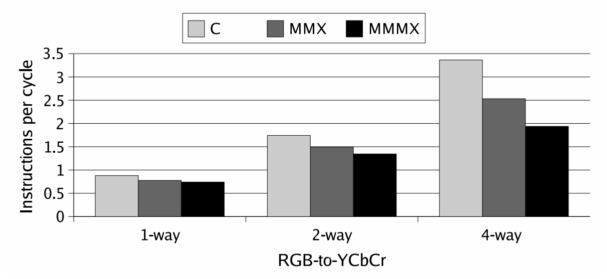

Figure 6. Instructions per cycle for RGB-to-YCbCr on out-of-order processors with different issue widths.

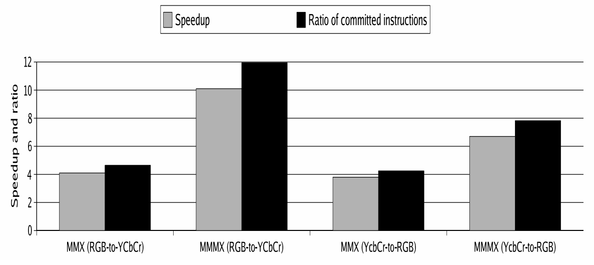

Figure 7. Ratio of committed instructions (C implementation to MMX and MMMX) against speedup for both kernels on 1-way out-of-order processor.

of committed instructions. This can be attributed to memory stall cycles, since MMX and MMMX do not reduce the time waiting for memory.

On average, MMMX reduces the dynamic number of instructions by a factor of 2.22 compared to MMX. This reduction comes from four factors. First, due to extended subwords, the MMMX implementation can employ 8-way parallel SIMD instructions while MMX can employ only 4-way parallelism. Second, the MRF allows to reorganize the data from band interleaved to band separated with zero cost, while MMX requires a significant amount of rearrangement overhead. Third, because MMMX processes more values in parallel, it reduces the loop overhead. Although the loop overhead incurred in the MMX implementation can be reduced by loop unrolling, this would increase the instruction footprints of the kernels. Finally, because 8 elements fit in a single MMMX register, the number of register copy instructions (`movq`) is reduced from 22 in each iteration of the MMX implementation of the RGB-to-YCbCr kernel to 15 in each iteration of the MMMX implementation.

4.3 Impact of the Number of Registers

It is well-known that for ISA legacy reasons, MMX has only 8 architectural registers. Because of this, the constants needed for performing color space conversion cannot be kept in registers but have to be reloaded from memory (cache) in each loop iteration. Although the constants will be found in cache most of the times, the number of load/store instructions is relatively large compared to the number of arithmetic instructions. In this section we consider the effect of adding more registers to the MMMX architecture.

First, we consider the effect of adding two registers to the MMMX architecture, so 10 in total. These registers are used to hold the fixed-point representations of the additive constants 16.5 and 128.5. Figure 8 shows that the speedup of MMMX with two additional registers over MMX (with 8 registers) is 2.97x for the RGB-to-YCbCr kernel and 1.89x for the YCbCr-to-RGB kernel. With 8 registers the speedups are 2.45x and 1.78x, respectively. So adding two registers improves performance by factors of 1.21x and 1.06x, respectively.

Next, we consider the effects of employing 13 extra media registers for 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 we need to employ only 9 additional registers. As can be seen in Figure 8, in this case the speedup of MMMX over MMX is 3.64x for RGB-to-YCbCr and 2.24x for YCbCr-to-RGB. So using 13 (resp. 9) extra registers provides an additional performance improvement by a factor of 1.49x (resp. 1.38x). Again, as also shown in Figure 8, the main reason for these performance improvements is the reduced number of instructions that need to be executed. It is interesting to observe, however, that in most cases the speedup is larger than the reduction of the dynamic instruction count. This is because keeping the constants in registers also reduces the memory stall cycles. Although the processor with only 8 media registers will mostly find the constants in the cache, sometimes it will not due to cache conflicts in which case it has to stall waiting for data to arrive from memory.

5 Conclusions

Because e.g. the red data of adjacent pixels are spaced 3 bytes apart and because intermediate results are wider than 8 bits, it is difficult to vectorize color space conversion efficiently using a conventional SIMD architecture such as MMX. In this paper we have shown that extended subwords and the matrix register file (MRF) are suitable techniques to overcome these limitations. With extended subwords the subwords of the media registers are wider than the subwords stored in memory. Extended subwords technique allows to perform many computations without overflow and, therefore, avoids packing/unpacking conversion overhead and increases the number of operations that can be performed in parallel by a single SIMD instruction. The MRF allows to view the register file as a matrix in which corresponding subwords in different registers correspond to a column of

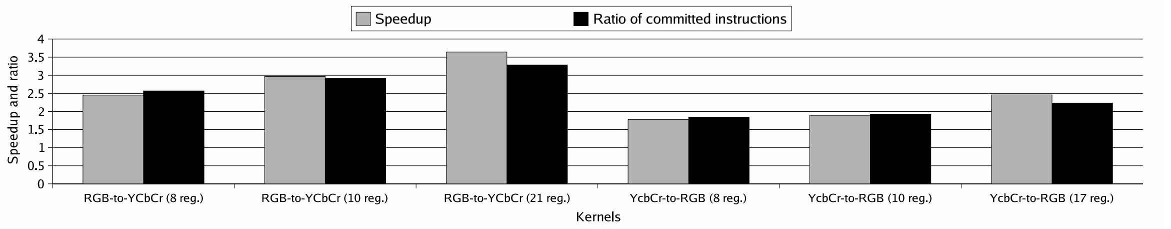

Figure 8. Speedup of MMMX over MMX and ratio of committed instructions (MMX to MMMX) on the 1-way out-of-order processor.

the matrix. Specifically, it is possible to load data stored consecutively in memory into a column of the MRF. Although the MRF has been proposed for block-based multimedia kernels such as the (I)DCT and motion estimation, in this paper we have shown that it can also be used for other permutations than matrix transposition.

We have considered the RGB-to-YCbCr and YCbCr-to-RGB kernels because these kernels consume a significant fraction of the total processing time of JPEG and MPEG encoders and decoders. Experimental results have been obtained using an extended version of the `sim-outorder` simulator of the SimpleScalar toolset. The results show that MMMX, which features extended subwords and the MFR, improves performance compared to a C implementation by up to 10.09x for the RGB-to-YCbCr kernel and by up to 6.74x for the YCbCr-to-RGB kernel. Compared to MMX, MMMX improves performance by up to 2.45x and 1.78x, respectively. Additionally, the results show that MMMX exhibits higher relative performance for lower-issue rates. This indicates that extended subwords and the MRF are suitable techniques for embedded multimedia systems where high issue rates and out-of-order execution are too expensive. The results also show that additional media registers would improve performance significantly, since it would allow to hold the constant coefficients in registers during the entire execution of the color space conversion kernels. For example, the results show that using at most 21 media registers provides an additional speedup over MMX of up to 1.49x and 1.38x for the RGB-to-YCbCr and YCbCr-to-RGB kernels, respectively.

Our future work will focus on considering the impact of out-of-order execution for larger register files and memory behavior of the color space conversions. In addition, we want to use extended subwords and the matrix register file techniques to implement other color space conversions such as RGB to CMYK conversion to develop new SIMD instructions.

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] M. Bartkowiak. Optimisations of Color Transformation for Real Time Video Decoding. In *Digital Signal Processing for Multimedia Communications and Services*, September 2001.

[3] F. Bensaali and A. Amira. Accelerating Colour Space Conversion on Reconfigurable Hardware. *Image and Vision Computing*, 23:935–942, 2005.

[4] S. Chatterji, M. Narayanan, J. Duell, and L. Oliker. Performance Evaluation of Two Emerging Media Processors: VIRAM and Imagine. In *Proc. 14th IEEE Int. Symp. on Parallel and Distributed Processing*, April 2003.

[5] J. Fridman and Z. Greenfield. The TigerSHARC DSP Architecture. *IEEE Micro*, 20(1):66–76, January-February 2000.

[6] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual Volume 3 System Programming Guide*, 2004. Order Number: 253668.

[7] J. Kim. *Architectural Enhancements for Color Image and Video Processing on Embedded Systems*. PhD thesis, Georgia Institute of Technology, March 2005.

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

[9] 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.

[10] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register Organization for Media Processing. In *In Proc. 6th Int. Symp. on High-Performance Computer Architecture*, pages 9–23, November 2000.

[11] 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*, May 2005.

[12] G. Sharma, M. J. Vrhel, and H. J. Trussell. Color Imaging for Multimedia. In *Proc. IEEE*, volume 86, 1998.

[13] M. Sima, S. Vassiliadis, S. Cotofana, and J. T. J. Eijndhoven. Color Space Conversion for MPEG Decoding on FPGA Augmented TriMedia Processor. In *Proc. 14th IEEE Int. Conf. on Application Specific Systems Architectures and Processors (ASAP)*, June 2003.

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

[15] TMS320C64x DSP Technical Brief. www.ti.com.

[16] J. W. Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J. P. Itegem, D. Amirtharaj, K. Kalra, P. Rodriguez, and H. Antwerpen. The TM3270 Media-Processor. In *Proc. 38th IEEE/ACM Int. Symp. on Microarchitecture*, 2005.