International Journal of Parallel Programming, Vol. 34, No. 3, June 2006 (© 2006)
DOI: 10.1007/s10766-006-0015-0

Avoiding Conversion and Rearrangement Overhead in SIMD Architectures

Asadollah Shahbahrami,[1,2,3] Ben Juurlink,[1] Demid Borodin,[1] and Stamatis Vassiliadis[1]

Received May 10, 2006; Accepted June 2, 2006

Single-Instruction Multiple-Data (SIMD) instructions provide an inexpensive way to exploit the Data-Level Parallelism in multimedia applications. However, the performance improvement obtained by employing SIMD instructions is often limited because frequently many overhead instructions are required to bring data in a form amenable to SIMD processing. In this paper, we employ two techniques to overcome this limitation. The first technique, extended subwords, uses four extra bits for every byte in a media register. This allows many SIMD operations to be performed without overflow and avoids packing/unpacking conversion overhead. The second technique, Matrix Register File (MRF), allows flexible row-wise as well as column-wise access to the register file. It is useful for many two-dimensional multimedia algorithms such as the (I) Discrete Cosine Transform, 2×2 Haar Transform, and pixel padding. In addition, we propose a few new media instructions. Experimental results obtained by extending the SimpleScalar toolset show that these techniques improve performance by up to a factor of 4.5 compared to a conventional SIMD instruction set extension.

KEY WORDS: Embedded media processors; multimedia kernels; register file; subword parallelism.

[1]Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, The Netherlands.
E-mail: {shahbahrami,benj,demid,stamatis}@ce.et.tudelft.nl
[2]Department of Electrical and Computer Engineering, Faculty of Engineering, Guilan University, Rasht, Iran.
[3]To whom correspondence should be addressed.

1. INTRODUCTION

Future general-purpose as well as embedded computing systems will exploit Instruction-Level Parallelism (ILP), Thread-Level Parallelism (TLP), as well as Data-Level Parallelism (DLP) to achieve high performance. Due to their area and energy efficiency, Single-Instruction Multiple-Data (SIMD) architectures provide an inexpensive way to exploit the DLP in multimedia applications. One of their main advantages is that the instruction decode and control overhead is amortized over many parallel operations. In general-purpose as well as high-performance embedded processors, SIMD instructions typically operate on 64- or 128-bit registers that contain several narrow data types. For example, a 128-bit SIMD register can be treated either as a vector of 16 bytes, eight 16-bit values, or four 32-bit fixed- or floating-points. Examples of such SIMD instruction set extensions are Intel's MMX,[1] SSE and SSE2,[2,3] Sun's VIS,[4] MIPS' MDMX,[5] and Motorola's AltiVec.[6]

When employing n-way parallel SIMD instructions, the ideal speedup over scalar execution is n. Usually, however, the attained speedup is much smaller. This is due to several reasons. First, the way multimedia data is stored in memory (the *storage format*) is usually too small to represent intermediate results. Consider, for example, the following loop which computes the arithmetic average of two images:

```
unsigned char src1[], src2[], dst[];
for (i=0; i<n; i++)
    dst[i] = (src1[i] + src2[i]) >> 1;
```

Even though the final result `dst[i]` is an 8-bit value, the intermediate result `src1[i]+src2[i]` is 9-bit. The source operands, therefore, need to be unpacked to a larger *computational format* before they can be processed and the results have to be packed again before they can be written back to memory. Obviously, this means loss of performance due to the extra cycles required for unpacking and packing. Furthermore, it also implies a loss of parallelism due to the reduction of the vector length. A second reason why the achieved speedup is usually much smaller than the maximum attainable speedup is that many two-dimensional (2D) multimedia algorithms process data along the rows as well as along the columns. In order to employ SIMD instructions in 2D algorithms, the matrix needs to be transposed frequently. On current SIMD extensions, however, transposition takes a significant amount of time. For example, to implement an 8×8 matrix transposition using MMX/SSE requires 56 instructions if the elements are 8 bits wide. If the elements are two bytes wide, then 88 instructions are required.

In this paper, we employ two techniques to overcome these limitations. The first technique, called *extended subwords*, uses registers that are wider than the size of a packed data type in memory. Specifically, for every byte of data, there are four extra bits. This allows many computations to be performed without overflow and avoids packing/unpacking overhead instructions. The second technique, called the *Matrix Register File* (MRF), allows load and store instructions to access the register file along the rows as well as along the columns. In other words, it allows a view of the register file as a matrix, where each register corresponds to a row of the matrix and corresponding subwords in different registers correspond to a column. In addition, we have designed a few new media instructions which we have found very useful but are not provided in, for example, MMX and SSE.

We have enhanced MMX with extended subwords and the matrix register file. The resulting architecture is called Modified MMX (MMMX, pronounce as triple-MX). Experimental results for many important multimedia kernels have been obtained by extending the SimpleScalar toolset.[7] The results show that MMMX improves performance by factors of 1.08 to 4.47 compared to MMX.

This paper is organized as follows. Related work is discussed in Section 2. Section 3 describes the proposed architecture, i.e., extended subwords, the matrix register file, the instruction set architecture, and estimates the area overhead of the proposed techniques. Section 4 discusses the simulation environment, the benchmarks, and their implementations in MMX and MMMX. The experimental results are provided in Section 5, and conclusions are drawn in Section 6.

2. RELATED WORK

Extended subwords have been previously proposed in Ref. 8, where they are called fat subwords. A register file organization that provides both row- and column-wise accesses has been proposed in Ref. 9. We build on these previous works but significantly extend on them. Specifically, our main contributions are:

- In Ref. 8, extended subwords have been proposed but not evaluated. Our work shows that extended subwords can be employed for many important multimedia kernels and that this technique improves performance significantly.
- We combine extended subwords with the MRF. Our experimental results show that using either of these techniques is insufficient to eliminate all pack/unpack and rearrangement overhead instructions.

– We have designed a few new instructions (see Section 3.3), which have been found very useful for several kernels and allow the elimination of all pack/unpack and rearrangement instructions for the considered kernels.

In Ref. 10, we have proposed using extended subwords to avoid data conversion overhead. For ten important media kernels we have determined the storage format and the maximum number of bits required to represent intermediate results. For seven kernels, a 12-bit data format was found to be sufficient. For the remaining three kernels, 24 bits were sufficient. The efficiency of extended subwords has been evaluated by counting the dynamic number of instructions needed to realize the kernels. The results show that using extended subwords reduces the dynamic number of instructions significantly (up to a factor of 2.7) for eight kernels. For two kernels the dynamic instruction count increases, but this is because in these kernels complex, special-purpose instructions have been emulated using simple, general-purpose instructions.

Although employing extended subwords reduces the dynamic instruction count substantially, most kernel implementations still incur many overhead instructions. This is because many 2D media algorithms process data along the rows as well as along the columns. Since adjacent column elements are not stored consecutively in memory, the data needs to be rearranged to exploit SIMD instructions. To avoid this rearrangement overhead, in Ref. 11, we have combined extended subwords with the MRF. Again, performance was evaluated by calculating the dynamic number of instructions. The results show that extended subwords and the MRF combined reduce the dynamic instruction count by up to a factor of 5.0. Moreover, for the considered kernels, the coalescence of both techniques eliminates the conversion and reorganization overhead completely, while extended subwords alone does not.

The limitation of our previous work is that the effectiveness of the proposed techniques has been evaluated by calculating the dynamic instruction count. Consequently, phenomena such as instruction dependencies, different instruction latencies, and cache effects have not been taken into account. In this paper, we significantly extend our previous work by providing experimental results obtained using a detailed, cycle-accurate simulator.

We briefly summarize other related approaches. In Ref. 12, new subword permutation instructions across multiple registers have been presented, which can perform all permutations of a 2×2 matrix. MIPS' MDMX[5] uses a predefined set of eight 8-bit and eight 16-bit wide shuffles to implement partial shuffle operations. The `Mix` instruction in HP's

MAX[13] can perform any permutation of the four 16-bit elements within a 64-bit register. Motorola's AltiVec[6] includes a three operand instruction (`vperm`) for data rearrangement. Oliver *et al.*[14] propose to include a subword permutation unit (SPU) in the execution pipeline. This SPU allows data permutation operations to be performed before other operations by removing permutation instructions from the instruction stream and instead having the SPU controller schedule the rearrangement instructions. In Ref. 15, a memory-to-memory architecture for 2D vectors is proposed. Control registers are used to specify the size of data in memory and the size of data during computation. If the computational format is larger than the storage format, the input data is automatically unpacked before being processed. A related proposal is the Matrix-Oriented Multimedia (MOM) extension.[16] MOM contains instructions that can be viewed as vector versions of SIMD instruction, i.e., they operate on matrices and each matrix row corresponds to a packed data type. MOM supports a matrix transpose instruction that transposes an 8×8 matrix with a latency of $8 + C$ cycles but this operation cannot be pipelined.

3. PROPOSED ARCHITECTURE

In this section, we describe extended subwords, the matrix register file, and the proposed instruction set. In addition, we briefly discuss the area overhead of the proposed techniques.

3.1. Extended Subwords

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

In previous work, we have proposed MMMX.[10] MMMX features registers that are wider than the size of a packed data type in memory. Specifically, for every byte of data there are four bits of extra precision. Load instructions automatically unpack data and store instructions implicitly pack data. We have shown that four extra bit for every byte in a register is sufficient for many multimedia kernels. This is also supported in Ref. 17, where it was shown that a 12-bit data format is sufficient for 85.7% of the processing in MPEG-4 encoding.

In MMMX, there are eight architectural multimedia registers as in MMX. However, these registers are 96 bits wide instead of 64 bits.

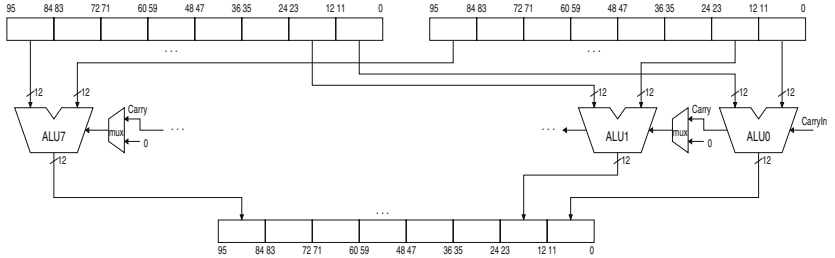

Fig. 1. A 96-bit partitioned ALU.

Figure 1 shown that a 96 bit ALU can be partitioned into eight 12-bit ALUs. Such a partitioned ALU can perform eight 12-bit, four 24-bit, two 48-bit, or a single 96-bit operation. The cost of implementing a partitioned ALU is very small.

3.2. Matrix Register File

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

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

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

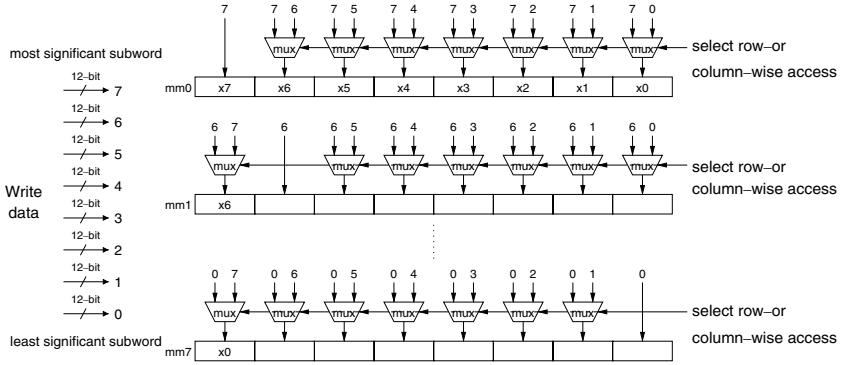

Fig. 2. A matrix register file with 12-bit subwords. For simplicity, write and clock signals have been omitted.

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

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

As an example of where column-wise access to the register file is very useful, Fig. 3 depicts the first stage of the LLM algorithm[18] for computing an 8-point 1D Discrete Cosine Transform (DCT). An 8×8 2D DCT can be accomplished by performing a 1D DCT on each row followed by a 1D DCT on each column. Initially, the bytes x_i $(0 \leq i \leq 7)$ are packed consecutively into one 64-bit quadword. It can be seen that this piece of code exhibits subword DLP, but only 4-way. Furthermore, in order to exploit this parallelism using SIMD instructions, the elements $x_0, x_7,\ x_1, x_6,\ x_2, x_5$, and x_3, x_4 have to be in corresponding subwords of different registers. This implies that the high and low doublewords of the quadword have to be split across different registers and that the order of the subwords in one of these registers has to be reversed. An alternative way to realize a 2D DCT is by transposing the matrix so that all the x_i's of different rows are in one register. In other words, we perform several 1D DCTs in parallel rather than trying to exploit the DLP present in a 1D

DCT. If the transposition step can be implemented efficiently, this method is more efficient than the first one. Moreover, it allows to exploit 8-way SIMD parallelism provided the subwords can represent the intermediate results.

We note that matrix transposition not only arises in the DCT but also in many other kernels such as the IDCT, vertical padding, and vertical subsampling. Furthermore, the matrix has to be transposed *twice* in order to exploit 8-way parallelism in these kernels.

3.3. Instruction Set Architecture

In this section, we briefly describe the MMMX instruction set.

Most MMMX instructions are direct counterparts of MMX/SSE instructions. For example, the MMMX instructions `fadd{12,24,48}` (packed addition of 12-, 24-, 48-bit subwords) correspond to the MMX instructions `padd{b,w,d} mm,mm/mem64`. MMMX, however, does not support variants of these instructions that automatically saturate the results of the additions to the maximum value representable by the subword data type. They are not needed because load instructions automatically unpack the subwords and store instructions automatically pack and saturate. For example, the `fld8u12` instruction loads eight unsigned 8-bit elements from memory and zero-extends them to a 12-bit format in a 96-bit MMMX register. Vice versa, the instruction `fst12s8u` saturates the 12-bit signed subwords to 8-bit unsigned subwords before storing them to memory. The instruction `fldc8u12` ("load column 8-bit to 12-bit unsigned") is used to load a column of the MRF.

In the remainder of this section, we describe the novel MMMX instructions which are not supported in MMX. In many media kernels all elements packed in a register need to be summed, while in other kernels adjacent elements need to be added. Rather than providing different instructions for summing all elements and adding adjacent elements, we decided to support adding adjacent elements only but for every packed data type. Whereas summing all elements would probably translate to a multicycle operation, adding adjacent elements is a very simple operation that can most likely be implemented in a single cycle. Figure 4 shows how eight 12-bit subwords can be reduced to a single 96-bit sum or 96-bit difference using the instructions `fsum{12,24,48}` and `fdiff{12,24,48}` respectively. The `fsum` instructions are used to synthesize the special-purpose SSE sum-of-absolute (SAD) instruction, which is not present in MMMX because it is of little benefit if subwords are 12-bit.[10]

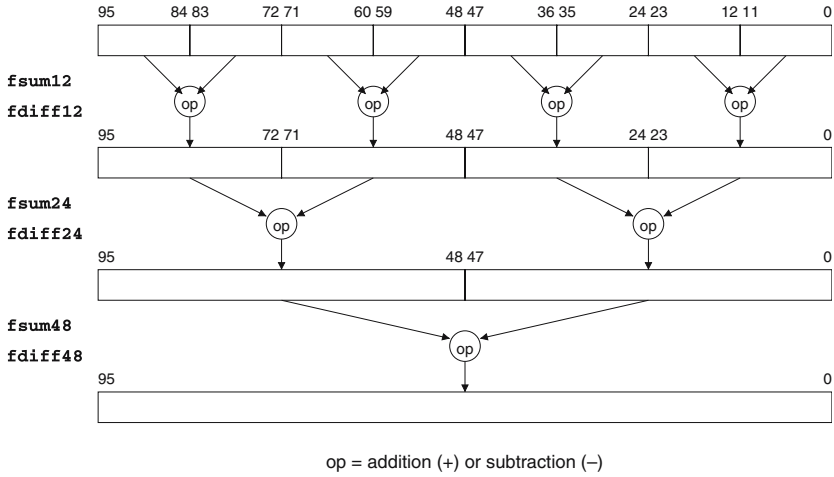

Fig. 4. Reducing eight 12-bit subwords to a single 96-bit sum or 96-bit difference using the instructions `fsum{12,24,48}` and `fdiff{12,24,48}`, respectively.

The instructions `fmadd{12,24} mm,mm/mem64` perform the multiply-add operation on adjacent subwords. Specifically, the instruction `fmadd12` multiplies the eight signed 12-bit subwords of the first operand with the corresponding subwords of the second operand and adds adjacent 24-bit products. The instruction `fmadd24` performs the same operation but on 24-bit subwords and produces two 48-bit results. In the MMX architecture, the multiply–add operation is only supported for the packed word (4×16-bit) data type (`pmaddwd`).

Another operation we have found useful to implement many multimedia kernels such as the (I)DCT kernels is the possibility to negate some or all elements in a packed register. The instructions `fneg{12,24} mm,imm8` negate the 12-bit (24-bit) subwords of the source operand if the corresponding bit in the 8-bit immediate `imm8` is set. If subwords are 24-bit, the four higher order bits in the 8-bit immediate are ignored. Finally, we remark that special-purpose MMX/SSE instructions such as `psadbw`, `pavg {b,w}` and conversion and rearrangement instructions such as `pshufw`, `packss {wb,dw,wb}`, `punpck{hbw,hwd,hdq,lbw,lwd, ldq}` are not supported in the MMMX architecture. In MMMX, the conversion and rearrangement instructions are not needed and the special-purpose instructions can be synthesized using a few general-purpose instructions.

3.4. Area Overhead and Delay

In this section, we provide 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 1 Kb, which is very small. In a recent paper,[19] 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 five 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.[20] 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 four (SIMD) instructions per cycle and requires eight read and four 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 $0.05\,\text{mm}^2$ in a $0.18\,\mu$m CMOS process,[20] so a coarse approximation of the area of a 64-bit partitioned ALU is $0.1\,\text{mm}^2$ and of a 96-bit partitioned ALU $0.15\,\text{mm}^2$. A relatively small integrated circuit is $1\,\text{cm}^2$. 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.

4. EXPERIMENTAL SETUP

4.1. Simulation Environment

In order to evaluate MMMX, we have used the `sim-outorder` simulator of the SimpleScalar toolset (revision 3.0d).[21] `sim-outorder` is a

detailed, execution-driven simulator that supports out-of-order issue and execution. New instructions can be synthesized without having to change or recompile the assembler. This is accomplished by adding annotations to instructions in the assembly files as in the following example:

```
add.d/a $f2, $f4, $f6.
```

The annotation `/a` in this example specifies that the first bit of the 16-bit annotation field should be set. The simulator can then be modified to indicate that the instruction above corresponds to, e.g., `paddb` (packed addition of 8 bytes) instead of double-precision floating-point addition. This method, however, is very error-prone.

In order to simplify synthesizing new instructions, two new tools have been developed: the Simple Scalar Instruction Tool (SSIT) and the Simple Scalar Architecture Tool (SSAT).[22] SSIT allows to use human-readable instructions such as `paddb` in the assembly files. It processes assembly files containing readable instructions, replaces them with corresponding annotated instructions, and modifies the source code of `sim-outorder` to support the new instructions. SSAT extends SSIT by providing the possibility to define new registers and to define aliases for existing registers. For example, in MMX as well as many other media extensions, the media registers correspond to the floating-point registers.

We remark that because the SimpleScalar architecture is RISC, 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 does not affect the validity of our simulations 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, in the Pentium 4 MMX instructions involving memory operands are translated to RISC-like micro-operations (μOPs). We also remark that the correctness of the MMX and MMMX codes has been validated by comparing their output to the output of C programs.

The main parameters of the modeled processors are depicted in Table I. We modeled processors with an issue width of 1, 2, and 4 instructions. So, when four SIMD instructions are issued simultaneously, up to 32 data operations are executed in parallel. 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 explained in Section 3.4, the latency and throughput of SIMD instructions are set equal to the latency and throughput of the corresponding scalar instructions.

Table I. Processor Configuration

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

4.2. Benchmarks

In this section, we describe the kernel benchmarks and briefly sketch their MMX and MMMX implementations. These kernels have been studied because they represent the major portion of many workloads. Most kernels process small (e.g., 8×8 or 16×16) sub-blocks but are performed on all sub-blocks of an image or frame. To investigate if this changes their quantitative behavior, we will also consider the case where the kernel is performed on all sub-blocks. Some of the kernels have been taken from the Berkeley Multimedia Kernel Library (BMKL).[23] Others have been found in literature (e.g., Refs. 24 and 25).

4.2.1. Matrix Transpose

Matrix transposition is at the center of many 2D multimedia algorithms. Because of this, we consider it as a kernel benchmark.

The MMMX implementation of matrix transpose is straightforward. Iteratively, we load a vector from memory and write it to a column of the register file. Once a sub-matrix has been transposed in this way, it is written back to memory. The MMX implementation, on the other hand, is more difficult and requires many permutation (pack and unpack) instructions.

The matrix transpose kernel was implemented for two different data types, byte and 12-bit. For brevity, they will be referred to as Transps.

(8) and (12), respectively. A 12-bit data format arises, for example, in the IDCT. In memory, however, these values are stored as 16-bit.

4.2.2. Add Block

For encoding a block or macroblock in Intra-coded mode in standards such as MPEG-4 and H.264, a prediction block is formed based on previously reconstructed blocks. The residual signal between the current block and the prediction is encoded. This residual signal data can be larger than 8-bit. The add block kernel is used in the decoder, during the block reconstruction stage of motion compensation. This kernel requires 9 bits of intermediate precision. Consequently, the MMX implementation needs to unpack the input data from 8- to 16-bit and to pack the 16-bit result to 8-bit. In the MMMX implementation this overhead is not required. Figures 5 and 6 depict the MMX and MMMX implementations of the inner loop of the add block kernel.

4.2.3. DCT and IDCT

The DCT and its inverse (IDCT) are widely used in many image and video compression applications. JPEG and MPEG partition the input image into 8×8 blocks and perform the 2D DCT on each block. The

```
1   pxor      mm7 , mm7
2   movq      mm0 , Blk1
3   movq      mm1 , mm0
4   punpcklbw mm0 , mm7
5   punpckhbw mm1 , mm7
6   paddsw    mm0 , Blk2
7   paddsw    mm1 , Blk2+8
8   packuswb  mm0 , mm1
9   movq      Blk1, mm0
```

Fig. 5. MMX implementation of inner loop of add block kernel.

```
1   fld8u12  mm0 , Blk1
2   fld16s12 mm1 , Blk2
3   fadd12   mm0 , mm1
4   fst12s8u Blk1, mm0
```

Fig. 6. MMMX implementation of inner loop of add block kernel.

input elements are either 8- or 9-bit, and the output is an 8×8 block of 12-bit 2's complement data.

Our implementations are based on the LLM algorithm,[18] which performs a 1D DCT and is performed on every row and column. The MMX/SSE implementation of the 2D DCT is due to Slingerland and Smith.[23] Because the input data is either 8- or 9-bit, they have used 16-bit functionality (4-way parallelism). Furthermore, they have vectorized the 1D DCT. As explained in Section 3.2, to do so requires a significant amount of overhead. The MMMX implementation performs several 1D row DCTs in parallel instead of parallelizing each 1D row DCT. This allows to exploit 8-way parallelism but requires that the matrix is transposed prior to the row and column DCTs.

The IDCT is the inverse of the DCT and can be accomplished using the same algorithm except that the stages are reversed.

4.2.4. Repetitive Padding

An important new feature in MPEG-4 is *padding*. Profiling results, reported in Refs. 26–28, indicate that padding is a computationally demanding process.

MPEG-4 defines Video Object Planes (VOPs) as arbitrarily shaped regions of a frame which usually correspond to objects. Motion estimation is defined over VOPs instead of frames. The padding process defines the color values of pixels outside the VOP. It consists of two steps. First, each horizontal line of a block is scanned. If a pixel is outside the VOP and between an end point of the line and an end point of a segment inside the VOP, then it is replaced by the value of the end pixel of the segment inside the VOP. Otherwise, if the pixel is outside the VOP and between two end points of segments inside the VOP, it is replaced by the average of these two end points. In the second step, the same procedure is applied to each vertical line of the block. Figure 7 shows horizontal and vertical repetitive padding for an 8×8 pixel block. In this figure, VOP boundary pixels are

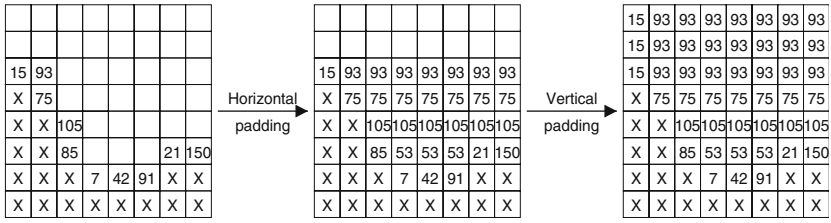

Fig. 7. Repetitive padding for VOP boundary blocks.

indicated by a numerical value, interior pixels are denoted by X, and pixels outside the VOP are blank.

Our implementations are based on the algorithm described in Ref. 25, where special instructions have been proposed for both horizontal as well as vertical repetitive padding. If column-wise access to the register file is supported, however, then both steps can be performed identically and efficiently using SIMD instructions, and special instructions for horizontal and vertical repetitive padding are not needed.

4.2.5. 2×2 *Haar Transform*

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

$$\begin{bmatrix} x0 & x1 \\ x2 & x3 \end{bmatrix} \underset{\text{transform}}{\overset{2\times2 \text{ Haar}}{\rightarrow}} \begin{bmatrix} x0+x1 & x0-x1 \\ x2+x3 & x2-x3 \end{bmatrix}, \qquad \begin{bmatrix} x0+x1+(x2+x3) & x0-x1+(x2-x3) \\ x0+x1-(x2+x3) & x0-x1-(x2-x3) \end{bmatrix}. \tag{1}$$

The MMX code for the inner loop of the 2×2 Haar transform is depicted in Fig. 8. The `punpcklbw` and `punpckhbw` instructions (instructions 9, 10, 12, and 13) expand the data to two bytes. Because both operands are in the same register and because MMX does not have an instruction that adds or subtracts adjacent elements, the instruction `pmaddwd` with some multiplicands set to 1 and others to -1 is used for the final addition or subtraction. The MMMX code for the 2×2 Haar transform is depicted in Fig. 9. MMMX reduces the number of instructions in the inner loop by almost a factor of 2 (from 20 to 11) compared to MMX. This reduction is due to two reasons. First, 8 pixels can be processed in parallel because 12 bits are sufficient for adding or subtracting 4 pixels. Second, as described in Section 3.3, MMMX includes instructions for adding and subtracting adjacent elements in a register.

4.2.6. Vector/Matrix Multiply

Vector/matrix multiplication is another important kernel with many applications. For example, it has been used to implement finite impulse response filters,[29] the discrete wavelet transform,[30] and, of course, matrix/matrix multiplication. The naive vector/matrix multiply algorithm traverses the matrix along the columns. Matrices are typically stored in

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

Fig. 8. MMX implementation of 2×2 Haar Transform.

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

Fig. 9. MMMX implementation of 2×2 Haar Transform.

row-major order leaving the columns scattered in memory. This kernel will be referred to as V × M.

In Ref. 31, two MMX implementations of vector/matrix multiplication have been explained. In the first implementation the matrix is split into 4×2 sub-matrices and the vector is also split into sub-vectors of two elements. This algorithm processes four columns of the matrix in parallel and accumulates results in a set of four accumulators. However, this algorithm exhibits poor cache utilization. In the second method, the outer loop of the vector/matrix multiply algorithm is unrolled four times. This algorithm processes 16 columns of the matrix in each iteration of the inner loop, and each iteration of the outer loop calculates 16 elements of the output vector. We compare our MMMX code to the second method, because it performs better than the first method. For this kernel a block size of 8×16 is used and the elements are assumed to be 12 bits wide (stored as 16-bit) as is the case in, for example, the Hadamard Transform in image processing and in edge detection using different masking. This kernel mainly benefits from the 8×12-bit multiply-add operation provided in MMMX (cf. Section 3.3).

4.2.7. Paeth Prediction

Paeth prediction is used in the PNG standard.[24] The Paeth predictor returns the pixel a, b, or c which is closest to the initial prediction $a+b-c$, where a is the element to the east of the current pixel, b the element to the north, and c is the element to the northeast. A pseudo-code description of the Paeth predictor (for one pixel) is given in Fig. 10.

Because a, b, and c are unsigned bytes, $p = a + b - c$ is in the range $-255 \ldots 510$ (10 bits), $pa = |p - a| = |b - c|$ is in the range 0–255 (8 bits), $pb = |p - b| = |a - c|$ is also in the range 0–255, and $pc = |p - c| =$

```
int Paeth_predict(a, b, c)
{
   p = a+b-c     /* initial prediction */
   pa = |p-a|
   pb = |p-b|
   pc = |p-c|
   if (pa<=pb AND pa<=pc) return a
   else if (pb<=pc) return b
   else return c
}
```

Fig. 10. Pseudo-code description of the Paeth predictor.

$|a + b - 2c|$ is in the range $-510 \ldots 510$ (10 bits). This implies that 12 bits are sufficient to perform this kernel without overflow. MMX, on the other hand, needs to unpack to 16-bit values and, therefore, can only exploit 4-way parallelism.

Because the MMX implementation computes the prediction for 3 pixels in a single iteration and MMMX for 7 pixels, the number of columns must be divisible by both 3 and 7 to implement this kernel conveniently. We, therefore, assumed a block size of 7×21.

5. EXPERIMENTAL RESULTS

In this section, we evaluate MMMX by comparing the performance obtained for an MMMX implementation of a kernel to the performance of an MMX implementation of the same kernel.

Figure 11 depicts the speedup of MMMX over MMX *for one execution of each kernel on a single block*. Consequently, the measured running times include many compulsory instruction and data cache misses. It can be seen that for most kernels MMMX achieves a speedup between 1.5 and 2.5. The two kernels for which a speedup smaller than 1.5 is obtained are the 2D IDCT and the Padding kernel. The input data of the 2D IDCT is 12-bit and some intermediate results are larger than 12-bit. Therefore, the MMMX implementation cannot employ 12-bit functionality (8-way parallel SIMD instructions) all the time but sometimes has to convert to 4×24-bit packed data types. The MMX implementation, on the other hand, is able to use 16-bit functionality all the time. The reason why MMMX improves performance by just 20% for the Padding kernel is that the MMX implementation employs the special-purpose `pavgb` instruction, which computes the arithmetic average of eight pairs of bytes. (More precisely, `pavgb` is supported in the SSE integer extension to MMX.) We

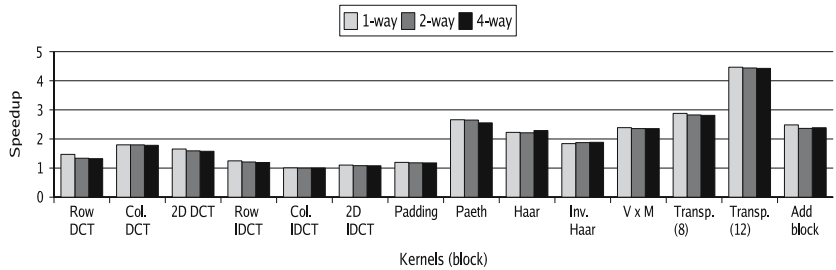

Fig. 11. Speedup of MMMX over MMX for one execution of each kernel on a single block.

decided not to support the `pavgb` instruction in MMMX because with extended subwords it offers little extra functionality since it can be synthesized using the more general-purpose instructions `fadd12` and `fsar12` (shift arithmetic right on extended subwords). Nevertheless, because the matrix needs to be transposed between horizontal and vertical padding MMMX provides a speedup.

The two kernels for which the highest speedups are obtained are the 8×8 matrix transpose (on 8-bit as well as 12-bit data) and Paeth. If the matrix elements are 8-bit, MMMX can use the MRF to transpose the matrix, while MMX requires many pack and unpack instructions to realize a matrix transposition. Furthermore, if the elements are 12-bit (but stored as 16-bit data types), MMMX is able to employ 8-way parallel SIMD instructions, while MMX can only employ 4-way parallel instructions. As a result, MMMX improves performance by more than a factor of 4.4. The speedup obtained for the Paeth kernel is approximately 2.6. The reason is that intermediate data is at most 10 bits wide and MMMX can, therefore, calculate the prediction for 7 pixels in each loop iteration while MMX computes the prediction for three pixels. Finally, it can also be observed in Fig. 11 that, in general, the speedup of MMMX over MMX slightly decreases when the issue width is increased. This can be expected because MMMX collapses several MMX instructions into a single instruction, so generally it will decrease the distance between dependent instructions.

As explained before, the results presented in Fig. 11 are for one execution on a single block. In most cases, however, the kernels are executed on all blocks of an image or frame. To investigate if this changes the results fundamentally, Fig. 12 depicts the image-level speedups (i.e., the speedups obtained when the kernels are executed on all blocks).

In general, the image-level speedups are higher than the block-level speedups. For example, the block-level speedup for the 2D DCT is between 1.58 (4-way) and 1.66 (1-way) while the image-level speedup is between 2.20 and 2.26. As another example, the block-level speedup for

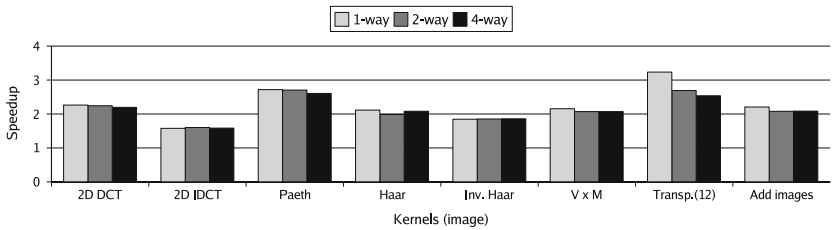

Fig. 12. Image-level speedup of MMMX over MMX.

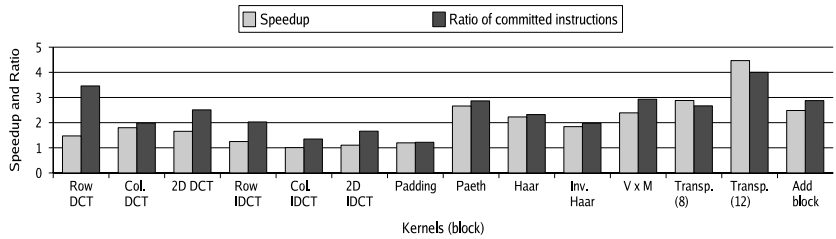

Fig. 13. Ratio of committed instructions (MMX to MMMX) versus speedup.

the Paeth kernel is between 2.55 and 2.66 whereas the image-level speedup is between 2.61 and 2.72. The reason for this behavior is as follows. Although executing the kernels on all blocks of an image does not reduce the number of data cache misses (because the images are too large to be kept in cache), it does reduce the number of instruction cache misses, since the kernels are relatively small and can be kept in the instruction cache.

The main reason why MMMX improves performance is because it needs to execute fewer instructions than MMX. To illustrate this reduction, Fig. 13 depicts the ratio of committed instructions, i.e., the ratio of the number of committed instructions for the MMX implementation to the number of committed instructions for the MMMX implementation. As Fig. 11, the results are for one execution of each kernel on a single block. For comparison, the speedup achieved by MMMX for the 1-way processor is depicted also.

Figure 13 shows that for most benchmarks MMMX reduces the dynamic number of instructions by a factor larger than 2. This is due to three reasons. First, when intermediate results are larger than 8 bits but smaller than 13 bits (or larger than 16 but smaller than 25 bits), MMMX can perform more operations in a single SIMD instruction than MMX. Second, employing extended subwords avoids conversion overhead. Third, for the kernels that process two-dimensional data the MRF reduces the cost of matrix transposition significantly. For some kernels, in particular Padding, the instruction count reduction is smaller than 2. As explained before, this is because the MMX implementation of this kernel employs the instruction `pavgb` which is not supported in MMMX and must be synthesized using two general-purpose instructions.

Figure 13 also shows that for all but two kernels the attained speedup is smaller than the instruction count reduction. For example, MMMX reduces the dynamic number of instructions by factors of $2.51\times$, $2.86\times$, and $2.93\times$ for the 2D DCT, Paeth, and Vector/Matrix Multiplication kernels, respectively, but the speedups achieved for these kernels are 1.66,

2.66, and 2.39, respectively. This is expected since the measured running times include memory stall cycles. Since MMMX only improves the performance of the computational part of an algorithm, the speedup will generally be smaller than the instruction count reduction. However, for the matrix transposition kernel (for both data formats) the attained speedup is larger than the instruction count reduction. A possible explanation is that the MMMX implementation of this kernel exhibits better cache performance than the MMX implementation.

To further illustrate the instruction count reduction, Figures 14 and 15 depict the mix of instructions in the MMX and MMMX implementations of the kernels, respectively. They break down the instruction mix into load/store instructions, overhead instructions (mainly pack and unpack) due to mismatch between the computational and storage formats, loop overhead instructions, multiplication operations, and other. Figure 14 shows that many (up to 38% of the total instruction count) overhead instructions are required in the MMX implementations to convert between the computational and storage formats. MMMX, on the other hand, eliminates this overhead completely, as shown in Fig. 15. Because MMMX reduces the instruction count compared to MMX, relatively it performs more load/store instructions.

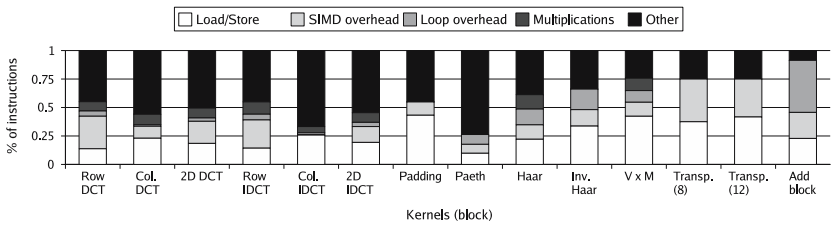

Fig. 14. Instruction mix in the MMX implementations of the kernels.

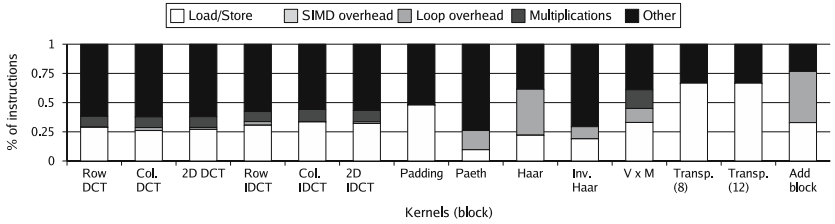

Fig. 15. Instruction mix in the MMMX implementations of the kernels.

6. CONCLUSIONS

In this paper, we have evaluated two techniques aimed at reducing data conversion and reorganization overhead. The first technique, extended subwords, uses four extra bits for every byte in a media register. This allows many SIMD operations to be performed without overflow and avoids packing/unpacking conversion overhead because of mismatch between the storage and computational formats. Furthermore, this implies that frequently MMMX can exploit 8-way parallelism in cases where MMX must resort to 4-way parallel SIMD instructions. The second technique is the MRF, which allows flexible row-wise as well as column-wise access to the register file. This eliminates the costly transposition steps which are required for many kernels that process 2D images.

We have shown that there are many important media kernels that can benefit from these techniques. Furthermore, simulation results obtained by extending the `sim-outorder` simulator of the SimpleScalar toolset show that MMMX improves performance significantly compared to MMX. The block-level speedups (i.e., for one execution on a single block) range from 1.1 to 4.47 with an arithmetic average of 2.03 and a geometric mean of 1.86 for a 1-way processor and from 1.08 to 4.43 with an arithmetic average of 1.99 and a geometric mean of 1.82 for a 4-way processor. The image-level speedups (i.e., the speedups obtained when the kernels are executed on all blocks of an image) range from 1.58 to 3.24 for a 1-way processor and from 1.58 to 2.54 for a 4-way processor. As can be expected, in general the speedup is smaller than the reduction of the dynamic number of instructions, due to memory stall cycles.

Future high-performance computing systems will exploit ILP, TLP, and DLP. An excellent example of this is the Cell processor[32,33] developed by a partnership of Sony, Toshiba, and IBM. Cell is a chip multiprocessor consisting of a PowerPC core that controls eight so-called Synergistic Processing Elements (SPEs). Each SPE is an in-order dual-issue statically scheduled processor that can issue two 128-bit SIMD instructions per cycle: one arithmetic instruction and one memory operation. Interestingly, the SIMD architecture does not support the packed 8-bit data type because such a narrow data type would degrade computation results.

The cost of implementing extended subwords and the MRF appears to be small. To validate this, we intend to develop RTL models of both techniques. Furthermore, we intend to identify other sources of inefficiency (e.g., unaligned memory accesses) and to develop (micro) architectural techniques to circumvent them.

REFERENCES

1. A. Peleg, S. Wiljie, and U. Weiser, Intel MMX for Multimedia PCs, *Commun. ACM*, 25–38 (1997).
2. S.K. Raman, V. Pentkovski, and J. Keshava, Implementing Streaming SIMD Extensions on the Pentium 3 Processor *IEEE Micro*, 47–57 (2000).
3. S. Thakkar and T. Huff, The Internet Streaming SIMD Extensions, *Intel. Technol. J.*, vol.1–8 (1999).
4. M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He, VIS Speeds New Media Processing, *IEEE Micro*, 10–20 (1996).
5. M. T. Inc., MIPS Extension for Digital Media, with 3D, www.mips.com.
6. K. Diefendorff, P. K. Dubey, R. H., and H. Scales, AltiVec Extension to PowerPC Accelerates Media Processing *IEEE Micro*, pp. 85–95 (2000).
7. D. Burger and T. M. Austin, The SimpleScalar Tool Set, Version 2.0, www.simplescalay.com.
8. N. Slingerland and A. J. Smith, Measuring the Performance of Multimedia Instruction Sets, *IEEE Trans. Comput.*, **51**(11):1317–1332 (2002).
9. Y. Jung, S. G. Berg, D. Kim, and Y. Kim, A Register File with Transposed Access Mode, in *Proc. Int. Conf. on Computer Design*, pp. 559–560 (2000).
10. B. Juurlink, A. Shahbahrami, and S. Vassiliadis, Avoiding Data Conversions in Embedded Media Processors, in *Proc. 20th Annual ACM Symp. on Applied Computing*, pp. 901–902 (2005).
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* (2005).
12. R. B. Lee, Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures, in *Proc. IEEE Int. Conf. on Application-Specific Systems Architectures and Processors*, pp. 9–23 (2000).
13. R. B. Lee, Subword Parallelism with MAX-2, *IEEE Micro*, 51–59 (1996).
14. J. Oliver, V. Akella, and F. Chong, Efficient Orchestration of Sub-Word Parallelism in Media Processors, in *Proc. Symp. on Parallel Algorithms and Architecture* (2004).
15. D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff, The CSI Multimedia Architecture, *IEEE Trans. VLSI Syst*, **13**(1):1–13 (2005).
16. J. Corbal, M. Valero, and R. Espasa, Exploiting a New Level of DLP in Multimedia Applications, in *Proc. Int. Symp. on Microarchitecture* (1999).
17. A. Dasu and S. Panchanathan, Reconfigurable Media Processing, *Parallel comput*, **28**(7):(2002).
18. C. Loeffler, A. Ligtenberg, and G. S. Moschytz, Practical Fast 1-D DCT Algorithms With 11 Multiplications, in *Proc. Int. Conf. on Acoustical and Speech*, vol. 2, pp. 988–991 (1989).
19. 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).
20. S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, Register Organization for Media Processing, *in Proc. 6th Int. Symp. on High-Performance Computer Architecture*, pp. 9–23 (2000).
21. T. Austin, E. Larson, and D. Ernst, SimpleScalar: An Infrastructure for Comput System Modeling, *IEEE Computer*, **35**(2):59–67 (2002).

22. B. Juurlink, D. Borodin, R. J. Meeuws, G. T. Aalbers, and H. Leisink, The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT), available via `http://ce.et.tudelft.nl/~shahbahrami/`.
23. N. Slingerland and A. J. Smith, Design and Characterization of the Berkeley Multimedia Workload, *Multimedia Syst*., **8**:315–327 (2002).
24. G. Roelofs, *PNG: The Definitive Guide*, Ph.D. thesis, O'Reilly and Associates (1999).
25. M. Berekovic, H. J. Stolberg, M. B. Kulaczewski, and P. Pirsch, Instruction Set Extensions for MPEG-4 Video, *J VLSI Signal Process*, **23**:27–49 (1999).
26. H. C. Chang, L. G. Chen, M. Y. Hsu, and Y. C. Chang, Performance Analysis and Architecture Evaluation of MPEG-4 Video Codec System, *in IEEE Int. Symp. on Circuits and Systems*, vol. 2, pp. 449–452 (2000).
27. H. C. Chang, Y. C. Wang, M. Y. Hsu, and L. G. Chen, Efficient Algorithms and Architectures for MPEG-4 Object-Based Video Coding, in *Proc. IEEE Workshop on Signal Processing Systems* (2000).
28. S. Vassiliadis, G. Kuzmanov, and S. Wong, MPEG-4 and the New Multimedia Architectural Challenges, in *Proc. 15th Int. Conf. on Systems for Automation of Engineering and Research*, pp. 24–32 (2001).
29. W. Chen, H. J. Reekie, S. Bhave, and E. A. Lee, Native Signal Processing on the Ultrasparc in the Ptolemy Environment, in *Proc. IEEE Conf. on Signals Systems and Computers*, vol. 2, pp. 1368–1372 (1996).
30. A. Shahbahrami, B. Juurlink, and S. Vassiliadis, Performance Comparison of SIMD Implementations of the Discrete Wavelet Transform, in *Proc. 16th IEEE Int. Conf. on Application Specific Systems Architectures and Processors* (*ASAP*), pp. 393–398 (2005).
31. Intel, *An Efficient Vector/Matrix Multiply Routine using MMX Technology*, Technical report, Intel Developer Services (2004).
32. B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. J. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. A. Brokenshire, M. Peyravian, V. To, and E. Iwata, The Microarchitecture of the Synergistic Processor for a Cell Processor, *IEEE J. Solid-State Circ*., **41**:63–70 (2006).
33. H. P. Hofstee, Power Efficient Processor Architecture and the Cell Processor, in *Proc. 11th IEEE Int. Symp. on High-Performance Computer Architectur*, pp. 258–262 (2005).