Efficient Vectorization of the FIR Filter

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—**The Finite Impulse Response (FIR) filter is one of the most important digital signal processing (DSP) kernels. It performs filtering of speech signals in modern voice coders such as the ETSI GSM EFR/AMR or ITU G.729, as well as in many other signal processing areas. Many contemporary digital signal processors as well as general-purpose microprocessors employ SIMD instructions to exploit the data-level parallelism present in media kernels such as the FIR filter. An important question is, therefore, how can the FIR filter be effectively vectorized to exploit the SIMD capabilities of the architecture?**

In this paper the performance of different methods for vectorizing the FIR filter such as vectorizing the inner loop and vectorizing the outer loop algorithms using C programming and SIMD instructions are compared. Additionally, we present another method to vectorize the FIR filter. It vectorizes the inner loop as well as the outer loop. All of these methods suffer from misaligned memory accesses. To overcome this problem we use four copies of filter coefficients that are aligned to 8-byte in memory. Our results show that the MMX implementation that vectorizes the inner loop is up to 3.34 times faster than the corresponding C implementation. Furthermore, the MMX implementation that vectorizes the inner loop as well as the outer loop and avoids misaligned memory accesses is up to 2.2 times faster than the MMX implementation that only vectorizes the inner loop. Finally, MMX implementation that vectorizes both loops and also avoids misaligned memory accesses is up to 1.69 times faster than the version that does not avoid misaligned memory accesses.

Keywords—**FIR, SIMD, Multimedia applications, data reuse.**

I. INTRODUCTION

One of the most important practical problems in high-performance computing is the design of efficient implementations of Digital Signal Processor (DSP) operations. DSP operations computing problems have grown in size and complexity following the increased capability of hardware. For efficient using of available hardware resources, the designers need a deep understanding of DSPs algorithms as well as a complete knowledge of the capabilities of the specific microarchitecture.

Digital filters are the firmness of most DSP systems. Among the digital filters, the Finite Impulse Response (FIR) filter is the most common DSP function in many multimedia applications such as audio, image, and video processing and pattern recognition. So efficient implementation of this algorithm either in hardware or in software is very important for providing higher performance and reduced energy consumption for multimedia processors and embedded systems which include portable devices such as Personal Digital Assistants (PDAs), digital cameras, and cellular phones. This is because these kind of devices have limitations such as low computational power, low memory capacity, and short battery life.

The FIR filter is a continuing computation over time of the form:

$$output(n) = \sum_{k=0}^{L-1} coef(k) * input(n-k) \qquad (1)$$

Each output value requires L multiplications and $L-1$ additions. The filter coefficients are denoted $coef(k)$ for $k = 0, ..., L-1$, where L is the filter length, and the signal length is N samples. Figure 1 depicts the signal-flow diagram of the FIR filter. It represents the direct calculation of Equation (1) and is called a direct structure [11].

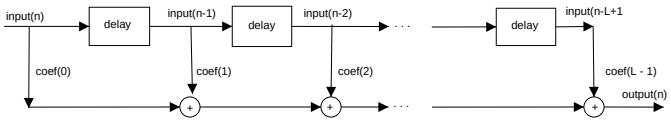

Fig. 1. Direct structure of an FIR fi lter.

The goal of the implementation is to have a minimun cycle time algorithm. This means that to do the filtering as fast as possible in order to achieve the highest sampling frequency. Figure 2 depicts a C implementation of the FIR filter. In all of algorithms in this paper we suppose that input data are stored in reversed order.

Designing high-performance implementations of DSP algorithms is a complex and difficult task. Implementing filters in software is a nontrivial problem because of the complex architecture of modern computers, multilevel

```
void fir()
{
  int *y_ptr, *c_ptr, *x_ptr;
  register int temp;
  int n, k;
  y_ptr = &output[0];
  for (n = 0; n < N; n++) {
    c_ptr = &coef[0];
    x_ptr = &input[n];
    temp = *c_ptr++ * *x_ptr++;
    for (k = 1; k < L; k++)
      temp = temp + *c_ptr++
                  * *x_ptr++;
    *y_ptr++ = temp;
  }
}
```

Fig. 2. C program for FIR fi lter.

memory hierarchy, and deficiencies of modern compilers. Our goal is to implement high-performance FIR filters. We focus on general-purpose processors enhanced with multimedia extensions vectorization issues that arise because of multimedia application characteristics such as Intel's MMX [15], [16].

This paper is organized as follows. Section II elaborates on different methods for vectorization. MMX implementations of the different algorithms are discussed in Section III. In Section IV is explained about performance evaluation of MMX code and C programs. Some related work is indicated in Section V. Finally, conclusions are drawn in Section VI.

II. VECTORIZATION OF THE FIR FILTER

DSP algorithms benefit from the multimedia extensions if and only if they are be vectorized. In [13], it has been indicated that the FIR filter can be vectorized in two ways: by vectorizing the inner loop (VIL), in which case the inner loop calculates several terms of a single output in parallel, or by vectorizing the outer loop (VOL), in which case the inner loop computes one term of several outputs in parallel.

Figure 3 depicts the C implementation of the VIL algorithm. This algorithm has been implemented in [2] using the VIS multimedia extension, where the FIR filter has been implemented as a matrix-vector multiplication on quad-words. The filter coefficients ($coef(k)$) were in a matrix of $N \times N$, and input data was stored in a vector of $N \times 1$.

This method, however, has some disadvantages. First, it is necessary that the filter coefficients are repeated many times in matrix, and some elements of the matrix should

```
for (n=0; n<N; n++)
{
  out_temp0 = out_temp1 = out_temp2
            = out_temp3 = 0;
  for (k=0; k<L ; k+=4) {
    out_temp0 += coef[k]  *input[n+k];
    out_temp1 += coef[k+1]*input[n+k+1];
    out_temp2 += coef[k+2]*input[n+k+2];
    out_temp3 += coef[k+3]*input[n+k+3];
   }
   output[n]  += out_temp0+out_temp1
              +  out_temp2+out_temp3;
}
```

Fig. 3. C program for vectorizing the inner loop (VIL) method.

```
for (n=0; n<N; n+=4)
  for (k=0; k<L; k++) {
    output[n]   += coef[k]*input[n+k];
    output[n+1] += coef[k]*input[n+k+1];
    output[n+2] += coef[k]*input[n+k+2];
    output[n+3] += coef[k]*input[n+k+3];
  }
```

Fig. 4. C program for vectorizing the outer loop (VOL) method.

be padded with zeros and a lot of multiplications are done with zero. Second, it increases the data traffic between the memory hierarchy and the register file of the processors, since the filter coefficients have to be loaded many times. This is because of lack of reusing coefficients for calculations of outputs. Furthermore, based on characteristics of multimedia applications [17] that have small data types this kind of algorithm is not suitable for SIMD architectures.

The C program of the second method (VOL) is depicted in Figure 4. Four outputs are computed in parallel. So four data values of $input(n+k)$ are multiplied with single coefficient $coef(k)$. That means, each filter coefficient is used for calculation of the four outputs at the same time. This algorithm has been implemented in [6], [5]. Although there is reusing coefficients in this algorithm, there is not any reusing of input data. This algorithm like previous method is not good for SIMD processing for multimedia applications.

Here, we use third method to vectorize the FIR filter: by vectorizing the inner and outer loops (VIOL) simultaneously. This method increases data reuse and facilitates efficient vectorization. So this method is suitable for SIMD architectures. In this algorithm the FIR implementation is considered as a multiplication of vector by vector. In

```
for (n=0; n<N; n+=4)
  for (k=0; k<L; k +=4) {
    output[n]   += coef[k]   * input[n+k]   + coef[k+1] * input[n+k+1]
                +  coef[k+2] * input[n+k+2] + coef[k+3] * input[n+k+3];
    output[n+1] += coef[k]   * input[n+k+1] + coef[k+1] * input[n+k+2]
                +  coef[k+2] * input[n+k+3] + coef[k+3] * input[n+k+4];
    output[n+2] += coef[k]   * input[n+k+2] + coef[k+1] * input[n+k+3]
                +  coef[k+2] * input[n+k+4] + coef[k+3] * input[n+k+5];
    output[n+3] += coef[k]   * input[n+k+3] + coef[k+1] * input[n+k+4]
                +  coef[k+2] * input[n+k+5] + coef[k+3] * input[n+k+6];
   }
```

Fig. 5. C program for vectorizing the inner and outer loops (VIOL) method.

each iteration of inner loop 4 filter outputs are calculated. Figure 5 depicts the C program of this algorithm. As this figure illustrates, in the inner loop four output are calculated with same coefficients, so coefficients are the same in each iteration for all of them. Additionally, three of the four input data, which are used for compute one output, are also used for calculation of next output, so reusing of coefficients and input samples are very good in this algorithm.

III. MMX Implementation of the Algorithms

For 16-bit data, vector dot-product calculations are efficiently implemented using MMX instructions by loading and processing four data elements at the same time. In the FIR filter, the relative alignment of the input and filter elements changes from one vector dot-product calculation to the next element. Figure 6 shows the relative alignment of the input and filter data. The arrows show elements which are multiplied together. The relative alignment changes by one element for each vector dot-product calculation. This implies that in three out of four vector dot-product calculations, all accesses to one of the vectors will be misaligned (8-byte data accesses which are not on 8-byte-aligned addresses).

The MMX implementation of the first method (VIL) is easy. But its performance is not good because of the following reasons. First, there is not any reusing of input data and coefficients in this algorithm. Second, because of sum and pack sequence in this implementation. After the four multiply-accumulate operation (MAC) (*pmaddwd* instruction in MMX instruction set), each accumulator includes a packed doubleword, each half of which contains half of the result in 32 bits. These halves must be summed together and packed to 16-bit for storing in memory. That means the calculation one result in each iteration is not efficient. These instructions are shown in Figure 7. Additionally, there is misaligned access for both reading input data and storing output data. Consecutive elements are loaded in registers in each iteration. For example, in one iteration *mm1=x15 x14 x13 x12* and *mm2=x14 x13 x12 x11* are loaded. Hence, misalignment is a big problem in this method. The number of dynamic instructions of this MMX implementation is $N(12 + \frac{7L}{4})$.

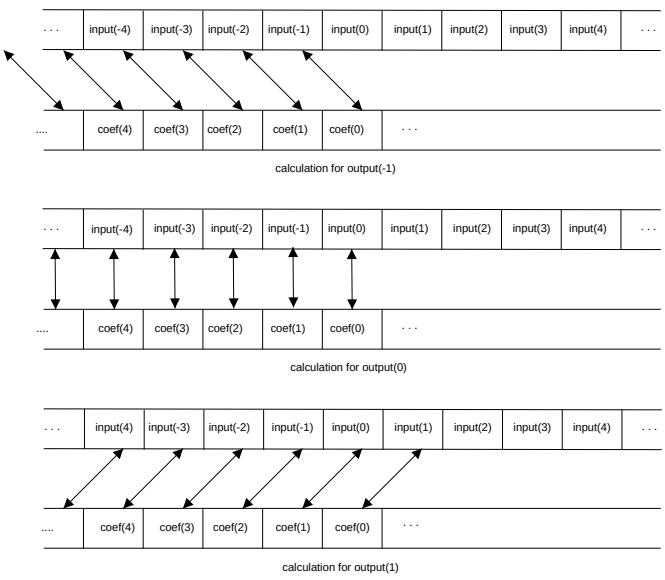

Fig. 6. Relative alignment of input and coeffi cients data for FIR fi lter.

The implementation of the second method (VOL) (Figure 4) using the MMX architecture is more difficult than the first method (VIL) and needs more instructions. Because to put one coefficient in 4 quadwords we have to use many overhead instructions. Additionally, multiplication of one coefficient that has been located in four part of a register with four different input samples is difficult, because there is no full multiplication instruction for 2 16-bit numbers. If we use low/high multiply instructions (*pmulhw* and *pmullw*) the number of executed instruction will be in-

```
;mm7 is an accumulator
movq      mm6    , mm7 ; mm6 = mm7
psrlq     mm7    , 32  ; shift right
paddd     mm7    , mm6 ; add double
packssdw  mm7    , mm7 ; pack
movd      output , mm7 ; store result
```

Fig. 7. Sum and pack operations for one accumulator.

```
; mm4, mm5, mm6, and mm7
; are four accumulators
movq      mm3 , mm7 ; mm3 = mm7
punpckhdq mm3 , mm6 ; unpack high
punpckldq mm7 , mm6 ; unpack low
paddd     mm7 , mm3 ; add doubleword
movq      mm3 , mm5 ; mm3 = mm5
punpckhdq mm3 , mm4 ; unpack high
punpckldq mm5 , mm4 ; unpack low
paddd     mm3 , mm5 ; add doubleword
packssdw  mm7 , mm3 ; pack
movq   output , mm7 ; store results
```

Fig. 8. Sum, pack operation, and combining writes into quadwords for four accumulators.

creased.

The MMX implementation of the third method (VIOL) (Figure 5) is more efficient than the two previous methods. As we mentioned above, reusing of coefficients in this algorithm is possible. In each iteration of outer loop four outputs are calculated. Summing and packing results of the MAC instructions are more efficient than their implementation in previous algorithms. Because the calculated results are in four accumulators, they can be packed with one *packssdw* instruction and the result for all four can be stored with one *movq* instruction. These instructions are shown in Figure 8. The number of executed instructions in this implementation is $\frac{N}{4}(20 + \frac{19L}{4})$. However, there is misaligned access for just reading input data in this MMX code. This is the most important disadvantage of this method.

To avoid misaligned data accesses we use different copies of the filter data. That means there are four copies of the filter data, each one with a different alignment relative to an 8-byte boundary. Figure 9 shows this structure. This method was implemented in [10], [8], for example. In [10] the filter length is 13 and in [8] a 16-tap FIR filter is used. In both implementations the filter data is copied four times and padded with zeros and aligned differently, as in Figure 9.

The performance of the MMX implementation of this

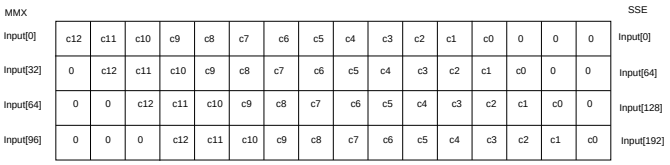

Fig. 9. Multiple copies of filter data for filter length of 13 (ci = coef(i)).

method which has no any misaligned access neither in reading input data nor storing result in memory, is better than three the previous methods as will be discussed in next section. We refer to this algorithm as vectorizing the inner and outer loops without misaligned (VIOLWM) access. The number of dynamic instructions of its MMX implementation is $\frac{N}{4}(20 + \frac{16L}{4})$.

IV. Performance Evaluation

All programs have been implemented using C programming and in line assembly language for MMX code. They will be referred to as C_VIL, C_VOL, and C_VIOL for C programs, and MMX_VIL, MMX_VIOL, and MMX_VIOLWM for MMX programs. C programs were compiled using the gcc compiler with optimization level *-O2*.

In the MMX implementation, input samples and coefficients are represented as 16-bit values, using the `short` data type. But for C programs we use `int` data type. According to ANSI semantics, all short integers are automatically promoted to register-length integers before conducting any arithmetic operations. This is known as *integral promotion* [14] and is implemented in most commercial compilers. That means in most cases the implementation that uses `int`s is faster than the program that employs `short`s. Accordingly, we compare our results to the C programs that represents coefficients and input data as `int`s.

Performance was measured using the cycle counters [9]. Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program [1]. The IA-32 counter is accessed with the *rdtsc* (read time stamp counter) assembly instruction. In order to eliminate the effects of context switching and compulsory cache misses, the *K-best* measurement scheme and a *warmed up* cache have been used, as explained in [1].

Figure 10 depicts the speedup of the MMX implementation of the VIL and VIOL algorithms over their C programs. As this figure shows, the MMX_VIL and MMX_VIOL programs are up to 3.34 and 4.37 faster than

C_VIL and C_VIOL, respectively. As the number of input samples are increased from 20 to 16384 the speedup increases. That means for larger input sample there is higher speedup compare to small input sample. This is because reading and storing data are more efficient in MMX implementation, where short data type is used compared to integer data type which is used in C programs.

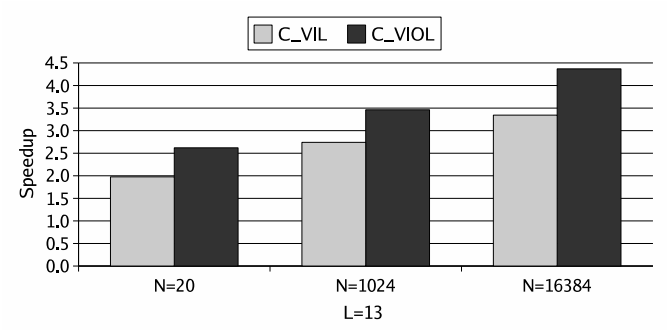

Fig. 10. Speedup of MMX implementation over C programs in VIL and VIOL algorithms.

Figure 11 shows the speedup of MMX_VIOLWM program over other MMX implementations. As this figure shows, MMX implementation of the vectorizing the inner loop as well as the outer loop and avoids misaligned memory accesses is up to 2.2 and 1.69 times faster than the MMX implementation that only vectorizes the inner loop and the version that does not avoid misaligned memory accesses, respectively. That means alignment access to memory is very important for getting higher performance in SIMD implementation of algorithms.

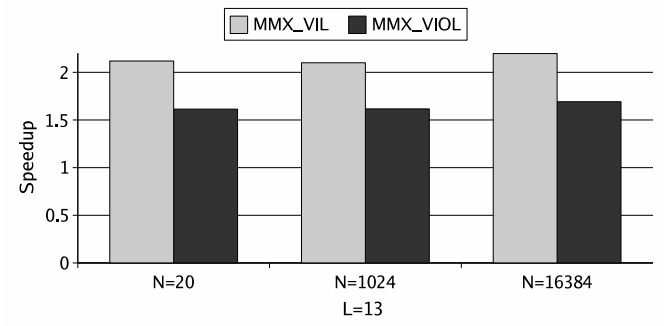

Fig. 11. Speedup of MMX implementation of the VIOLWM algorithm over MMX implementations of the VIL and VIOL methods.

Although using four copies of filter data can solve the memory alignment, it uses four times more memory than other methods. Furthermore, the coefficients and input data are both not reused in each iteration. That means four copies of four coefficients should be read in each iteration. It increases the data traffic between the memory hierarchy and the register file of the processors, as filter coefficients have to be loaded many times. This is because of lack of reusing coefficients for calculations of outputs.

V. Related Work

As in previous section it was indicated vectorizing inner loop algorithm was implemented in [2] using the VIS multimedia extension. The second method (VOL) has been implemented in [6], [5]. Although there is reusing coefficients in this algorithm, but there is not any reusing of input data. In [4], for calculation of each output, it is splited into four sequences. After the calcution of four sequences, they are added each other and provide the final result of the one output of the filter. For all of outputs of the filter this algorithm is repeated.

In [3], [12], [7] have been explained one method for reusing the coefficients of FIR filter by using vector pointers. So all of elements of a vector pointer point to the same coefficient in a register and are incremented by one for next coefficient. For the input data, each item is used in the four equations at a different place in the Figure 4. Therefore, all of elements in a vector pointer point to consecutive entries in a register and like vector pointer to coefficients are incremented by one after each use is suitable for addressing the data. Figure 12 illustrates this algorithm. As this figure shows, V_{p0} index the coefficient $coef[0]$ and V_{p1} indices the input data. As output filter processing continues, the vector values will be updated by the indicated stride one, so that sequential coefficients and input values are used as inputs to the vector multiply-accumulate operations.

In this method, there are overhead both on hardware and on software. Because in hardware, vector pointer registers and a vector pointer unit are needed. In software, the programmer or the compiler has to use initializations instructions for loading pointers to two source vector pointers and a destination vector pointer.

VI. Conclusions

The performance of different methods for vectorizing the FIR filter has been evaluated using C programming and MMX instructions. These algorithms are vectorizing the inner loop, vectorizing the outer loop, and vectorizing the inner loop as well as the outer loop with and without misalignment access. Based on our results MMX implementation of vectorizing the inner loop is up to 3.34 faster than the corresponding C implementation. Additionally, MMX implementation of vectorizing the inner and outer loops without misaligned access is up to 2.2 and 1.69 faster than MMX implementation of the vectorizing the inner loop and vectorizing the inner and outer loops algorithms, respectively. That means that aligned access to memory is an

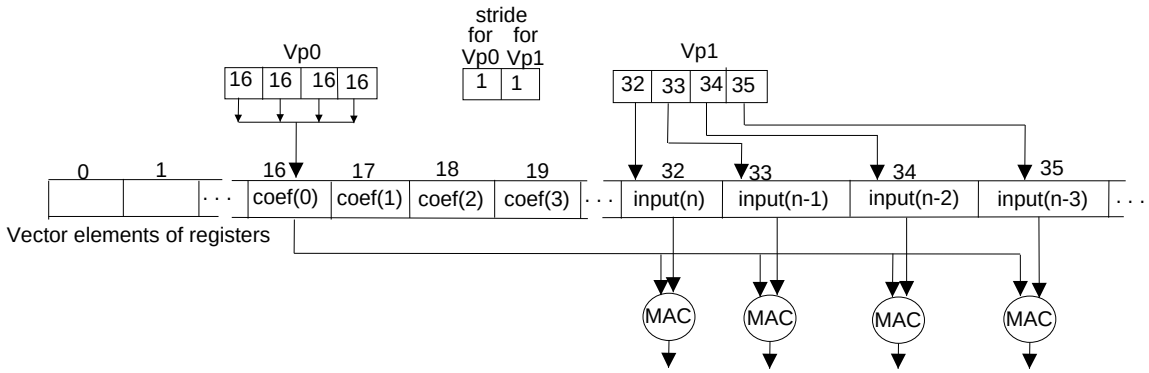

Fig. 12. Using vector pointers for reusing coeffi cients of the FIR fi lter and input data.

important factor for getting higher performance in SIMD implementation of DSPs algorithms.

REFERENCES

[1] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.

[2] W. Chen, H. J. Reekie, S. Bhave, and E. A. Lee. Native Signal Processing on the Ultrasparc in the Ptolemy Environment. In *IEEE Conf. on Signals Systems and Computers*, volume 2, pages 1368–1372, November 1996.

[3] J. H. Derby and J. H. Moreno. A High-Performance Embedded DSP Core With Novel SIMD Features. In *Proc. IEEE Int. Conf. on Acoustics Speech and Signal Processing*, volume 2, pages 301–304, April 2003.

[4] J. Fridman. Data Alignment for Sub-Word Parallelism in DSP. In *Proc. IEEE Workshop on Signal Processing Systems*, pages 251–260, October 1999.

[5] J. Fridman. Sub-Word Parallelism in Digital Signal Processing. *IEEE Signal Processing Magazine*, 17:27–35, March 2000.

[6] J. Fridman and Z. Greenfi eld. The TigerSHARC DSP Architecture. *IEEE Micro*, 20:66–76, January-February 2000.

[7] H. C. Hunter and J. H. Moreno. A New Look at Exploiting Data Parallelism in Embedded Systems. In *Proc. IEEE Int. Conf. on Compilers Architectures and Synthesis for Embedded Systems*, pages 159–169, 2003.

[8] Intel Corporation. *Real and Complex FIR Filter Using Streaming SIMD Extensions*, 1999. Order Number: 243643-002.

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

[10] Intel Corporation. *Using MMX Technology Instructions to Compute a 16-Bit FIR Filter*, 2004. www.intel.com/IDS.

[11] S. M. Kuo and W. S. Gan. *Digital Signal Processors Architectures, Implementations, and Applications*. Prentice Hall, 2005.

[12] J. H. Moreno, V. Zyuban, U. Shvadron, F. D. Neeser, J. H. Derby, M. S. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. W. Asaad, T. W. Fox, D. Littrell, M. Biberstein, D. Naishlos, and H. Hunter. An Innovative Low-power High-performance Programmable Signal Processor for Digital Communications. *IBM Journal of Research and Development*, 47(2/3):299–326, March/May 2003.

[13] D. Naishlos, M. Biberstein, S. B. David, and A. Zaks. Vectorizing for a SIMdD DSP Architecture. In *Int. Conf. on Compilers Architectures and Synthesis for Embedded Systems*, volume 2, pages 2–11, November 2003.

[14] International Standard Organization. Programming Languages - C. ISO/IEC 9899, 1999.

[15] A. Peleg, , and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, pages 42–50, August 1996.

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

[17] A. Shahbahrami, B.H.H. Juurlink, and S. Vassiliadis. A Comparison Between Processor Architectures for Multimedia Applications. In *Proc. 15th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, pages 138–152, November 2004.