# Performance Impact of Misaligned Accesses in SIMD Extensions

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*— **In order to provide the best performance for memory accesses in the multimedia extensions that load or store consecutive subwords from/to memory, the memory access must be correctly aligned. That means that an $n$-byte transfer must be set on an $n$-byte boundary. In most SIMD architectures, unaligned memory accesses have a large performance penalty or are even disallowed. For example, our result shows that for addition of two arrays of size $1024 \times 1024$, whose addresses are either aligned or unaligned, aligned code is 1.47 times faster than unaligned code using SSE instructions. Hence, in this paper we evaluate the advantages and disadvantages of different techniques to avoid misaligned memory accesses such as replication of data in memory, padding of data structures, loop peeling, and shift instructions. Our result shows that the MMX implementation of the FIR filter using replication of data is up to 2.20 times faster than the MMX implementation with misaligned accesses. Furthermore, the MMX and SSE implementations using loop peeling technique are up to 1.45 and 1.66 faster than their implementation for addition of two arrays with different sizes, respectively.**

*Keywords*— **Multimedia extensions, SIMD, Data alignment.**

## I. Introduction

The most common approach to consider the requirements of multimedia applications in the existing General-Purpose Processors (GPPs) has been the extension of the Instruction Set Architecture (ISA) with Single Instruction Multiple Data (SIMD) instructions. These media extensions such as MMX [21] and SSE [22] add a set of SIMD instructions in order to exploit the data parallelism available in multimedia applications. A memory instruction in these SIMD extensions is able to load or store multiple data items. In order to provide the best performance for memory accesses the memory addresses must be naturally aligned.

Some media extensions do not provide any hardware support for unaligned accesses. For example, AltiVec extension [5] is unable to operate on data that is not naturally aligned. A load instruction loads 16-byte contiguous from 16-byte aligned memory. To provide this, hardware does not consider the last 4-bit of the memory address. On the other hand, some other media extentions provide instructions to access unaligned memory addresses but at the expense of a big performance penalty. For instance, the Intel's SSE extension includes both hardware support and unaligned exceptions [25]. The instructions `movdqa` and `movaps` require that the effective addresses to be aligned. While with instructions `movdqu` and `movups` the hardware recognizes the unaligned references and returns the desired memory data. As another example, some Digital Signal Processing (DSPs) for embedded system, like the TigerSharc support accesses to misaligned by hardware units such as data alignment buffer which performs the required aligned loads and shifts [7].

The data access patterns of many applications are inherently misaligned. For example, only 14% of the dynamic accesses in the SPEC95fp and MediaBench benchmark are aligned [16]. As another example, the motion estimation and motion compensation algorithms, which are used in video coding and decoding operate on byte boundaries.

This alignment constraint can significantly impact the effectiveness of SIMD vectorization. For example, our result shows that for addition of two arrays of size $1024 \times 1024$, whose addresses are either aligned or unaligned, aligned code is 1.47 times faster than unaligned code using SSE instructions. If aligned access cannot be guaranteed, the programmer should consider the alignment in software using overhead instructions. This means that the data from two consecutive aligned addresses must explicitly merge.

In this paper we evaluate the advantages and disadvantages of different techniques to avoid misaligned memory accesses. We study some techniques such as replication of data in memory, padding of data structures, and loop peeling. Our results show that the MMX implementation of the Finite Impulse Response (FIR) filter using replication of data is up to 2.20 times faster than the MMX implementation with misaligned accesses. Furthermore, the MMX and SSE implementations using loop peeling technique are up

```
for (i = 0; i< N; i++)
  a[i] = b[i] + c[i];
```

Fig. 1. A loop with aligned accesses.

```
for (i = 0; i< N; i++)
  a[i+1] = b[i+2] + c[i+3];
```

Fig. 2. A loop with misaligned accesses.

to 1.45 and 1.66 faster than their implementation for addition of two arrays with misaligned accesses, respectively.

This paper is organized as follows. Section II describes the alignment restriction placed on SIMD memory operations. In addition, this section discusses the behavior of different multimedia extensions on the aligned and unaligned memory accesses. In Section III different techniques to improve misaligned accesses are discussed. Section IV explains about performance evaluation of MMX and SSE codes that use replication of data in memory and loop peeling techniques to improve misaligned memory accesses. Some related work is indicated in Section V. Finally, conclusions are drawn in Section VI.

## II. BACKGROUND

In this section we describe the alignment problem in detail, the behavior of multimedia extensions on the aligned and unaligned memory accesses, and cache line split.

### A. Address Alignment

A memory address $A$ is aligned if $A \mod n = 0$, where $n$ is the width of the accessed data in bytes. When a memory address is misaligned, the value $A \mod n$ determines the offset from alignment. Figure 1 and Figure 2 show a loop with aligned and misaligned accesses with consideration that their base addresses are aligned, respectively.

Some architectures limit memory references to aligned addresses only. In this case, it is the compiler's responsibility to explicitly merge data in two aligned addresses before and after the unaligned address. This leads to an overhead, which can reduce performance in vectorized code sequences. If the processor directly supports misaligned references, these accesses still need extra cycle, resource usage, or both. This is because to support misaligned accesses by hardware, the processor must implement a rotation or merge operation between memory and registers, potentially increasing the latency of an SIMD load instruction. For example, Figure 3 depicts SIMD vectorization of the code in Figure 2. This figure illustrates that data used in the same iteration can be misaligned when they are loaded into registers. To provide correct alignment, these

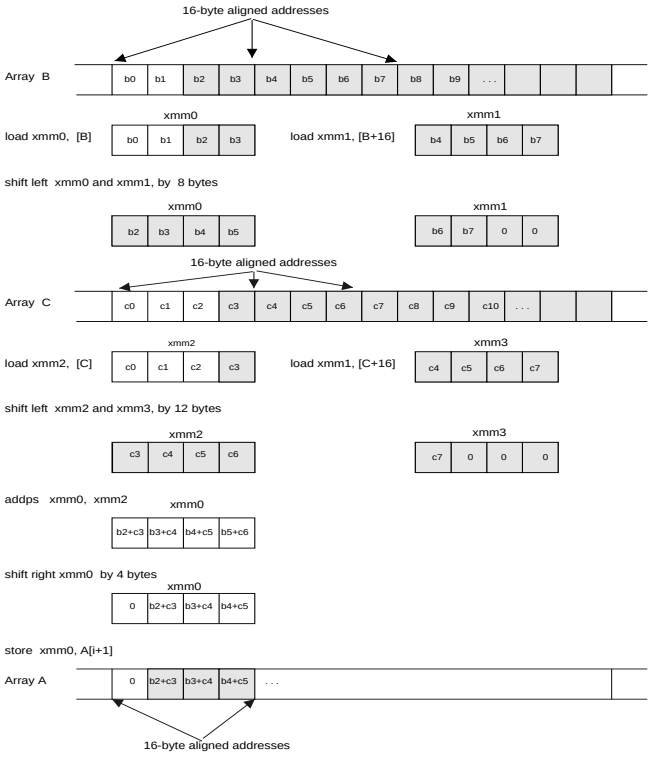


Fig. 3. SIMD vectorization of $a[i+1] = b[i+2] + c[i+3]$ using data reorganizations instructions.

data must be rearranged using data reorganization instructions such as shifting right and left. This means that in the realignment process first, we should read the aligned memory address that is located before the unaligned position. Second, load the aligned data that is placed after the unaligned position. Finally, merge the two previous parts and extract the necessary data. As a result, the processor needs two memory accesses to make an unaligned memory access. On the other hand, aligned accesses require only one memory access.

### B. Multimedia Extensions

Multimedia extensions such as MMX [21], SSE [22], SSE2, and SSE3 [12] support misaligned accesses by hardware and MAX-1/2 [18], VIS [26], 3DNow [1], and VMX [10] support by software. The VMX ISA ignores the low-order bits of an SIMD memory address, to force access to aligned addresses. In contrast to VMX, Intel's IA-32 extensions provide direct hardware support for accessing misaligned data. For instance, SSE/SSE2 distinguishes between aligned and unaligned load instructions. The aligned load instructions are more efficient than unaligned load instructions, but are valid only when the compiler can guarantee alignment addresses. This means that VMX and SSE represent two different points in the de-

```
movq  mm1, [b] ; load 8 bytes from b[]
paddw mm1, [c] ; add mm1 with 8 bytes of c[]
movq  [a], mm1 ; store mm1 in a[]
```

Fig. 4. MMX implementation of the misaligned and aligned loops.

```
movaps xmm0, [b]  ; load 16 bytes from b[]
addps  xmm0, [c]  ; add xmm0 with c[]
movaps [a] , xmm0 ; store xmm0 in the a[]
```

Fig. 5. SSE implementation of a aligned loop.

sign of multimedia memory systems. The 64-bit MMX ISA does not have different instructions for aligned and unaligned load and stores, but still performance penalties may arise for unaligned `movq` instruction.

In SIMD extensions like, AltiVec, MIPS-3D, and Alpha, the hardware automatically clear the lower-bit of the effective address and returns an aligned address. In these extensions, it is necessary to access the neighboring aligned locations just before and after the requested address and to shift, rotates, or permute them in order to extract the unaligned data elements as previously discussed in Figure 3.

Table I depicts the misalignment support provided by different multimedia extensions. Column "realign load" shows a merge operation to extract the relevant data elements according to the misalignment of the address. For example, in AltiVec the `vperm` instruction takes three arguments, two aligned vectors and a vector mask generated by the `lvsl` instruction, which is called realignment token by Nuzman and Henderson [20] and to produce the desired unaligned data [5]. The realignment token can be an address, a bit mask, a vector of indices, or any other value that is a function of the unaligned address.

We have implemented both aligned and misaligned loops as Figure 1 and Figure 2 for different data types using MMX and SSE instructions. The MMX implementation of both misaligned and aligned loops is almost the same, as is depicted in Figure 4 for *short* data type.

The SSE implementation of the aligned loop is depicted in Figure 5. In this program we have used `movaps` instruction. This instruction requires the effective address to be aligned. If the data is not 16-byte aligned, a general protection exception will be generated. This means that the data must be 16-byte aligned for packed floating-point operations. SSE provides the `movdqu` (move unaligned double quadword) and `movups` instructions for loading memory from addresses that are not aligned on 16-byte boundaries. If we know that the data is not aligned, we use these instructions to avoid the protection error exception. The unaligned load has a bigger latency than the aligned

```
movups xmm0 , [b+2] ; load 16 bytes from b[]
movups xmm1 , [c+3] ; load 16 bytes from c[]
addps  xmm0 , xmm1  ; xmm0 = xmm0 + xmm1
movups [a+1], xmm0  ; store xmm0 in the a[]
```

Fig. 6. SSE implementation of a misaligned loop.

one. The SSE implementation of the misaligned loop is depicted in Figure 6.

Table II shows the speedup of MMX and SSE implementations of addition of two arrays with aligned accesses over their implementation with misaligned accesses for different data types and different array sizes on the Pentium 3 and Pentium 4 processors. On the other words, we have implemented the codes in Figure 1 and Figure 2 using MMX and SSE instructions set. This table illustrates that implementation using aligned accesses are up to 2.26 and 2.72 times faster than implementation with misaligned accesses for different data types on the Pentium 3 and Pentium 4 processors, respectively.

The programs that use unaligned load instructions frequently encounter situations where the source spans across a cache line boundary. Loading from a memory address that spans across a cache line boundary causes a hardware stall and degrades program performance. Cache line split is discussed in detail in next Section.

### *C. Cache Line Split*

As previously discussed misaligned accesses can incur significant performance penalties. One such case is cache line splits. With existing of the cache line splits, instead of reading all the requested data in one cache line, the processor has to wait until it can access the next cache line to get the remaining data. For example, we assume that the cache line size is 64-byte. There are three possible misaligned accesses that can occur after SIMD implementation of single-precision floating-point operations [3]. As illustrated in Figure 7, every other iteration of the vector loop will cause a cache line split of +4, +8, or +12 bytes into the next cache line.

There is `lddqu` instruction, a 128-bit unaligned load to avoid cache-line splits in Prescott processor [13]. If the address of the load instruction is aligned on a 16-byte boundary, `lddqu` instruction loads requested 16-byte. While if the address of the load instruction is not aligned on a 16-byte boundary, `lddqu` instruction loads a 32-byte starting at the 16-byte aligned address before the address of the requested load instruction. After that, it provides the requested 16-byte. In addition, `lddqu` instruction is used for integer data type [13].

| Media Extension | Unaligned Load | Aligned Load | Realign Load | Realignment Token |
|---|---|---|---|---|
| AltiVec/VMX | | lvx | vperm | lvsl |
| AltiVec/VMX on PPE | lvlx, lvrx | | | |
| SSE/SSE3 | movdqu, lddqu | movdqa | | |
| MIPS-3D | | luxcl | alnv.ps | address |
| MIPS64 | ldl, ldr | | | |
| MVI | | ldq_u | extql, extqh | address |

TABLE I
UNALIGNED LOAD SUPPORT BY DIFFERENT MEDIA EXTENTIONS.

| Array size | Char | | Short | | Integer | | Float | |
|---|---|---|---|---|---|---|---|---|
| | Pen. 3 | Pen. 4 | Pen. 3 | Pen. 4 | Pen. 3 | Pen. 4 | Pen. 3 | Pen. 4 |
| 256 | 1.46 | 1.64 | 1.58 | 1.8 | 1.05 | 1.71 | 3.37 | 4.73 |
| 512 | 1.58 | 1.78 | 1.66 | 2.01 | 1.1 | 1.91 | 3.72 | 5.46 |
| 1024 | 1.66 | 2.02 | 1.74 | 2.24 | 1.08 | 2.04 | 4 | 3.96 |
| 32768 | 1.13 | 2.7 | 1.12 | 2.1 | 1.1 | 1.91 | 1.41 | 2.47 |
| 64000 | 1.28 | 2.71 | 1.23 | 2.2 | 1.16 | 1.04 | 1.45 | 1.13 |
| 1048576 | 1.15 | 1.83 | 1.11 | 1.99 | 1.1 | 1.25 | 1.36 | 1.53 |
| 4194304 | 1.15 | 1.94 | 1.11 | 2 | 1.08 | 1.15 | 1.38 | 1.24 |
| 16777216 | 1.14 | 2.24 | 1.12 | 1.89 | 1.09 | 1.15 | 1.37 | 1.23 |
| Average | 1.32 | 2.11 | 1.33 | 2.03 | 1.09 | 1.52 | 2.26 | 2.72 |

TABLE II
SPEEDUP OF THE MMX AND SSE ALIGNED CODES OVER THE MMX AND SSE MISALIGNED CODES FOR ADDITION OF TWO ARRAYS FOR DIFFERENT DATA TYPES ON THE PENTIUM 3 AND PENTIUM 4 PROCESSORS. PEN. 3 = PENTIUM 3 AND PEN. 4 = PENTIUM 4.

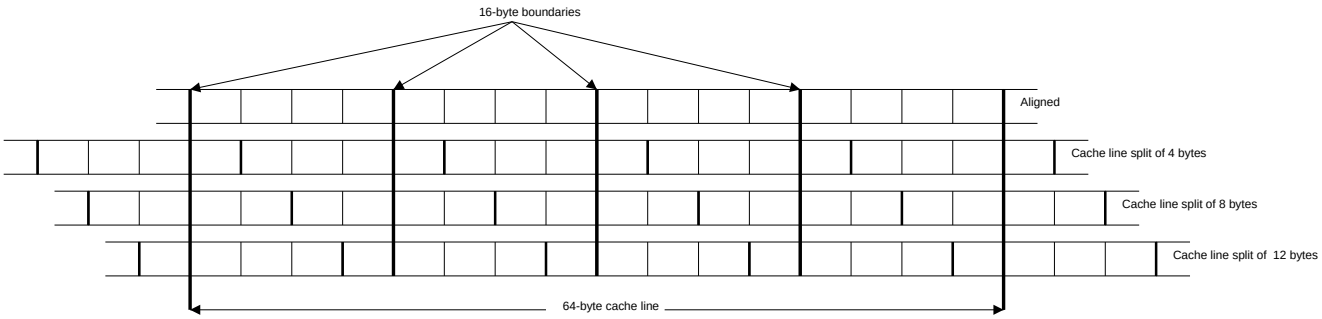


Fig. 7. Cache line splits in vector access.

## III. TECHNIQUES TO REMOVE MISALIGNMENTS

In this section, different techniques to reduce misalignment overhead are explained.

### A. Loop Peeling

Loop peeling is a technique that is used to execute a few iterations of the misaligned loop until the memory address within the loop reaches a known alignment. When the loop reaches an aligned address, we can exit the pre-loop code and start to execute the main loop. Figure 8 depicts one example of this technique. As can be seen to execute one iteration of the misaligned loop, an aligned loop is provided. This is compiler responsibility that statically peel off one iteration to align all access patterns in the main loop.

A misalignment address $A$ can be resolved by $(16 - A)/n$ times loop peeling with considering to a 16-byte boundary alignment and element size of $n$ bytes. We assume that arrays are aligned at least at an element size boundary.

```
double a[N], b[N];
                    a[1] = b[1] * 3;
for (i=1; i<N ;i++) for (i=2; i<N; i++)
  a[i] = b[i] * 3;    a[i] = b[i] * 3;

(a) misaligned loop (b) providing a aligned
                    loop using loop peeling
```

Fig. 8. (a) A misaligned loop. (b) using loop peeling technique to provide an aligned loop.

### *B. Dynamic Loop Peeling*

Using pointers in some programming languages such as C is usuall and a programmer can use them anywhere and pass pointers to arbitrary locations in memory. Consequently, alignment of pointers at compile time will be difficult. In such cases, the compiler must be ready to deal with each possible alignment. Some compilers such as Intel C++ use *dynamic loop peeling* technique to deal with pointer alignment [3], [2], [17]. Figure 9 illustrates one example of dynamic loop peeling. In this example, we assumes that data types are aligned to their data width. As can be seen in this figure the parameters of *mult* function that is called from *main* function are not aligned to 16-byte boundaries. The offsets of the arguments are different in each call. A pre-loop code has been written in part (b), which executes some iterations to align $x[i]$ to a 16-byte boundary.

### *C. Padding multidimensional arrays*

Using multidimensional arrays in programming can cause a misalignment. Misaligned addresses occur for multidimensional arrays when the size of the low-order dimension is not a multiple of the vector length. Figure 10 shows an example. In part (a), the offset of $a[i][j]$ depends on the $i$ loop iteration. For instance, reference $a[0][0]$ is 16-byte aligned, but reference $a[1][0]$ has an offset of 12 bytes. Part (b) shows the array's layout in memory. In some cases, the compiler can fix an array's offset to its column dimension by padding [17]. Adding an extra element to the column dimension of the array in Figure 10 (a) produces the layout in part (c), where all accesses $a[i][0]$ are aligned.

### *D. Multi-Version Code*

One approach to uncover alignment information is to use runtime tests to identify aligned addresses dynamically [2], [15] as depicted in Figure 11. There are four conditions in this code. Within each test, the compiler can vectorize the loop for a specific set of conditions. Obtaining the exact offset of a misaligned address is almost unimportant. It will be sufficient to determine, which memory references are aligned. For example, in VMX the instruction sequence for accessing misaligned data does not depend on the offset. While for IA-32 processors, offset information is useful.

```
void mult(float *x,float *y,float *z,int M)
{
   int i;
   for (i=0; i<M; i++)
     x[i] = y[i] * z[i];
}

void  main()
{
 float x[N], y[N], z[N];
 ...
 mult(x+1, y+1, z+1, C);
 ...
 mult(x+3, y+3, z+3, C+50);
}
 (a)

void mult(float *x,float *y,float *z,int M)
{
 ...
//Pre-loop to align x[i] to 16-byte boun.
 unsigned offset_x = (unsigned) &x[0] & 15;
 peel = offset_x ? (16 - offset_x) / 4 : 0;
 for (i=0; i < min(M, peel); i++) {
   x[i] = y[i] * z[i];
}
   // x[i] guaranteed to have offset 0
   for (; i<M; i++) {
     x[i] = y[i] * z[i];
 }
 (b)
```

Fig. 9. To enforce runtime alignment using dynamic loop peeling. (a) The *main* and *mult* functions. (b) Using a pre-loop in the *mult* function to enforce alignment for x[i].

Two most likely offsets are $x \mod 16 = 0$ and $y \mod 16 = 8$. In general, multi-version code that tests $m$ different offsets with respect to a certain base for $n$ different memory references gives rise to $m^n$ separate branches as well as one $else$-branch that is required if not all possible offsets are covered.

### *E. Duplicating Constant Tables*

Many multimedia kernels contain references to arrays of constants. These constants ate often accessed in non uniform manner, to make their dynamic alignment difficult. These tables of constants are almost small. This means that we can duplicate them for each possible alignment that

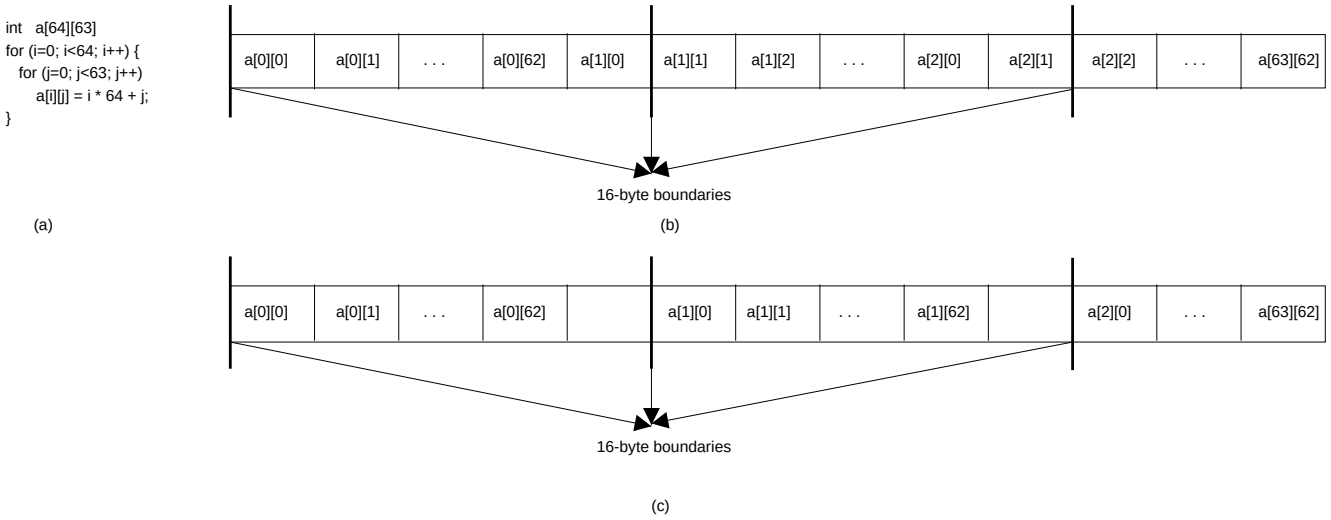


Fig. 10. (a) Accessing a two-dimensional array, which has 63 4-byte elements in each row. (b) Data layout of the array in the memory without padding technique. (c) Data layout after padding, all references a[i][0] are aligned to a 16-byte boundary.

```
void copy(double *x, double *y) {
...
// Offset of x and y relative to 16 bytes
unsigned offx = (unsigned) x & 15;
unsigned offy = (unsigned) y & 15;
if (offx == 0 && offy == 0)
  // Both x and y aligned
   for (i=0; i<N; i++)  x[i] = y[i];
else if (offx == 0 && offy == 8)
      // x aligned, y misaligned
      for (i=0; i<N; i++)  x[i] = y[i];
     else if (offx == 8 && offy == 0)
          // x misaligned, y aligned
          for (i=0; i<N; i++) x[i] = y[i];
         else // x and y misaligned
          for (i=0; i<N; i++) x[i] = y[i];
 }
```

Fig. 11. Multi-version code to detect alignment at runtime.

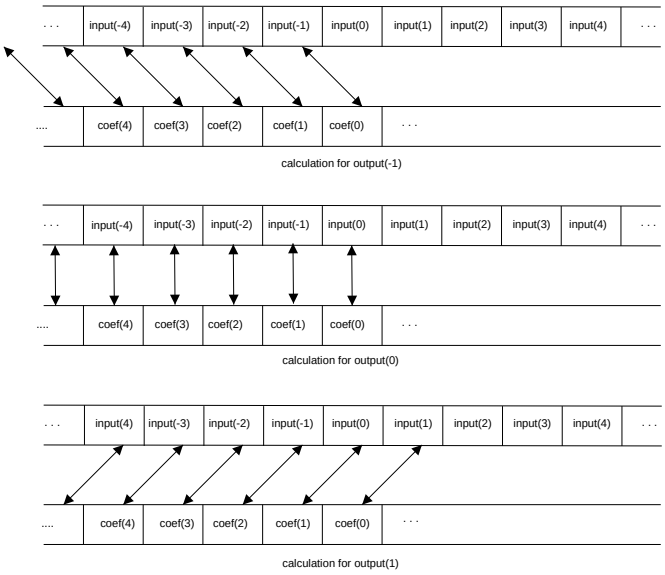


Fig. 12. Relative alignment of input and coeffi cients data for FIR fi lter.

we want. For example, 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 12 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).

To avoid misaligned data accesses we use different copies of the filter data. That means that there are four copies of the filter data, each one with a different alignment relative to an 8-byte boundary. Figure 13 shows this structure. This method was implemented in [14], [11].

## IV. Performance Evaluation

We have implemented two techniques, duplicating constant values and loop peeling. We used the former method for MMX implementation of the FIR filter and the later technique for MMX and SSE implementations of the C code in Figure 2. In the MMX implementation of the FIR filter, input samples and coefficients are represented as 16-bit values, using the `short` data type. For 16-bit data, vector dot-product calculations are efficiently imple-

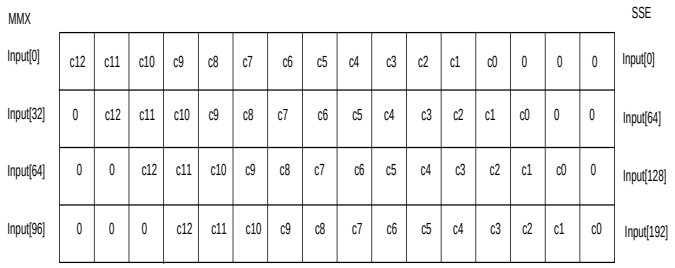


Fig. 13. Multiple copies of fi lter data for fi lter length of 13 (ci = coef(i)).

mented using MMX instructions by loading and processing four data elements at the same time. We have implemented three different implementation of the FIR filter. First, vectorizing the inner loop (VIL), in which case the inner loop calculates several terms of a single output in parallel. Second, by vectorizing the inner and outer loops (VIOL). Finally, vectorizing the inner and outer loops without misaligned (VIOLWM) access using duplicating coefficients values [24]. They will be referred to as MMX_VIL, MMX_VIOL, and MMX_VIOLWM for MMX programs.

We implemented C code in Figure 2 using MMX and SSE instructions set. In both MMX and SSE implementations *char* and *float* data types were used, respectively. In order to improve alignment in the C code we used loop peeling technique for array $a$. After using this technique we also implemented both MMX and SSE codes. This means that we implemented both MMX and SSE codes with misaligned accesses and MMX and SSE programs with aligned accesses for array $a$.

Performance was measured using the cycle counters [12]. Cycle counters provide a very precise tool for measuring the time that elapses between two different points in the execution of a program [4]. 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 [4].

Figure 14 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 using duplicating coefficients values is up to 2.20 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.

Figure 15 depicts the speedup of the MMX and SSE implementation that use loop peeling technique in the array addition program over their implementation with misaligned accesses. As this figure shows loop peeling techniques improve performance significantly up to 1.45 and 1.66 in the MMX and SSE codes, respectively.

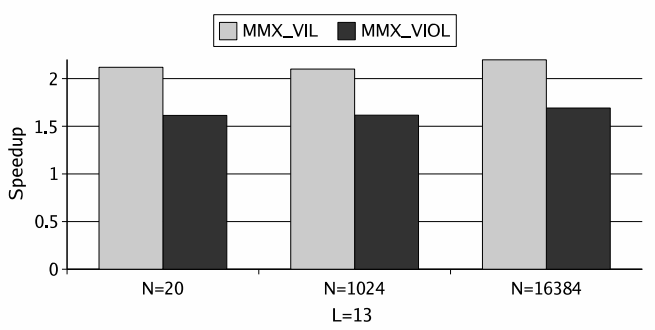


Fig. 14. Speedup of MMX implementation of the VIOLWM algorithm over MMX implementations of the VIL and VIOL methods.

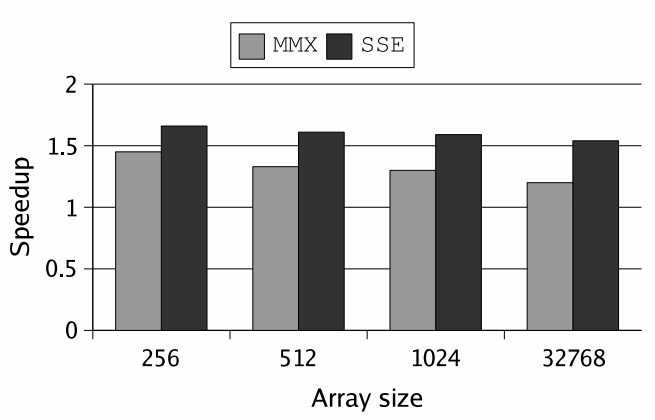


Fig. 15. Speedup of MMX and SSE implementations using loop peeling over the MMX and SSE implementations with misaligned accesses in the array additions program.

## V. Related Work

Eichenberger et al. [6] have proposed a systematic method to simdizing loops with misaligned stride one memory references for SIMD architecture with alignment constraints. In their method data reorganization instructions are automatically generated during the simdization to align data in registers. These instructions are inserted into the simdized code to satisfy the actual alignment constraints. They called vectorization for SIMD architectures as simdization. They have introduced a new data reorganization operator, *vshiftstream(c1, c2)*, which shifts all values of a register stream from offset $c1$ to offset $c2$. In general, they focused on generating optimized SIMD codes in the presence of misaligned references.

Eichenberger et al. have investigated several policies

to generate data reorganization instructions using *vshift-stream* nodes such as zero-shift policy, eager-shift policy, lazy-shift policy, and dominant-shift policy. The zero-shift policy shifts each misaligned register to a stream offset of 0 immediately after it is loaded from memory. After that it shifts each register to the alignment of the store address just before it is stored to memory. While eager-shift policy shifts each misaligned load directly to the alignment of the store instruction. Lazy-shift policy is almost based on the eager-shift policy and dominant-shift policy reduces the number of stream shifts by shifting registers to the most dominant stream offset.

Larsen et al. [16] have concentrated on the detection of memory alignments and with techniques to increase the number of aligned references in a loop. They used loop peeling to align accesses. The loop peeling method is equivalent to the eager-shift policy with the restriction that all memory references in the loop must have the same misalignment.

Bik et al. [3] have described a code sequence of aligned loads and shuffle to load memory references that cross cache line boundaries. This scheme was implemented in Intel's compiler for SSE2.

Fridman [7] has explained three solutions for data alignment. First, maintaining multiple replication of coefficients. This approach used by Intel for MMX and SSE [14], [11] implementation of the FIR filter. Second method depends on memory system support for misaligned accesses, as for example, the MIPS [19] and SSE [22] memory systems. Third, accessing the aligned memory addresses before and after the misaligned memory address and providing misaligned subwords using rearrangement instructions. This approach is used in Motorola AltiVec [5] using a permutation unit. These techniques have some limitations. For example, the replication of the coefficients has two drawbacks. First, It needs large memory for replication of the $N$ filter coefficients. Second, This method cannot be applied to algorithms, which use dynamic data, as in convolution and correlation [7]. In the memory system that support misaligned accesses, however, a single misaligned access is significantly slower than an aligned access. Finally, using a permutation unit and shifter in the third method causes the extra execution time and larger code size of the program.

A dedicated hardware resource called Data Alignment Buffer (DAB) is used in TigerSHARC Digital Signal Processing (DSP) [9], [8]. A DAF is provided between the memory banks and the computation elements. The data alignment buffer allows unaligned accesses to specified operands that are stored in different memory rows. A DAF is managed directly by software. It uses muxes as well as temporary memory for a complete aligned memory word.

The VIS instructions set added instructions for partial load and store as well as instructions for merging partial loaded data [26]. This means that to provide data in a form that the VIS instructions can operate, subword rearrangement and alignment instructions should be used. This results in extra overhead that limits the performance benefits from VIS instructions. For instance in [23], Ranghanathan et. al. observe that for MPEG/JPEG codecs, on average, 41% of the executed VIS instructions are instructions associated with subword rearrangement and alignment operations.

## VI. Conclusions

Our results showed that unaligned memory accesses have a large performance penalty. For example, the MMX and SSE aligned codes for addition of two arrays are up to 2.26 and 2.72 times faster than their implementations using misaligned accesses on the Pentium 3 and Pentium 4 processors, respectively. To improve the misaligned memory accesses several techniques have been discussed in this paper. These techniques are following as, loop peeling, padding multidimensional arrays, multi-version code, and duplicating constant tables. We have implemented loop peeling and duplicating constant tables techniques using MMX and SSE instructions. Based on our results MMX implementation using replication of data is up to 2.20 times faster than the MMX implementation with misaligned accesses in the FIR implementation. Furthermore, the MMX and SSE implementations using loop peeling technique are up to 1.45 and 1.66 faster that their implementation for addition of two arrays with different sizes, respectively.

## Acknowledgment

This research was supported in part by the Netherlands Organisation for Scientific Research (NWO).

## References

[1] *3DNow Technology Manual*, 2000.

[2] A. J. C. Bik. *The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance*. Intel Press, Hillsboro, 2004.

[3] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic Intra-Register Vectorization for the Intel Architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

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

[5] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, pages 85–95, March-April 2000.

[6] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. In *Proc. ACM*

*SIGPLAN Conf. on Programming Lnguage Design and Implementation*, volume 39, pages 82–93, May 2004.

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

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

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

[10] IBM Corporation. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments*, 2005.

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

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

[13] Intel Corporation. *Prescott New Instructions Software Developer's Guide*, 2004. Order Number: 252490-004.

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

[15] A. Krall and S. Lelait. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*, 28(4):347–361, August 2000.

[16] S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for Increasing and Detecting Memory Alignment. Technical Report LCS-TM-621, MIT/LCS, November 2001.

[17] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proc. 11th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 18–29, September 2002.

[18] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, pages 51–59, August 1996.

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

[20] D. Nuzman and R. Henderson. Multi-platform Auto-vectorization. In *Proc. IEEE Int. Conf. on Code Generation and Optimization*, 2006.

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

[22] 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.

[23] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of Image and Video Processing with General Purpose Processors and Media ISA Extensions. In *Proc. Int. Symp. on Computer Architecture (ISCA 26)*, pages 124–135, 1999.

[24] A. Shahbahrami, B.H.H. Juurlink, and S. Vassiliadis. Efficient Vectorization of the FIR Filter. In *Proc. 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, pages 432–437, November 2005.

[25] S. Thakkar and T. Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, pages 1–8, 1999.

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