On the Use of the MMC Language to Utilize SIMD Instruction Set

Patricio Bulić and Veselko Guštin

University of Ljubljana, Faculty of Computer and Information Science, Slovenia,
patricio.bulic@fri.uni-lj.si,
WWW home page: http://lra-1.fri.uni-lj.si

Abstract. This paper presents the use of the Multimedia C (MMC) language to develop multimedia applications. The MMC language was designed to support operations with multimedia extensions included in all modern microprocessors. Although the idea to extend high programming languages to support vector operations is not novel, we show that integration of multimedia extensions into C is valuable. This is specially true for idiomatic expressions which are difficult for a compiler to identify. The MMC language has been used to develop some of the most frequently used multimedia kernels. The presented experiments on these scientific and multimedia applications have yielded good performance improvements. Although this paper discuses the use of MMC, the key features of the MMC language and implementation of its compiler are also presented.

1 Introduction

Today's computer architectures are very different from those of a few years ago in terms of complexity and the computational availabilities of the execution units within a processor. Practically all modern processors have facilities that improve performance without placing an additional burden on the software developers, as well as those facilities which require support from external entities (i.e. assembler language and compilers) such as multimedia (also called short vector) processing ability (i.e. Intel MMX, Intel SSE, Intel SSE2, Motorola Altivec, SUN VIS, ...). This was reflected in an extension of the assembly languages (extended instruction set).

But a powerful SIMD (*Single Instruction Multiple Data*) multimedia instruction set is worthless without the mean to utilize it. Today, we can utilize SIMD multimedia instruction set in three ways:

1. assembly language - this is the most effective method but it is also more tedious and error prone than any other methods,
2. shared libraries - these libraries are often available from microprocessor manufacturers, but they tend to only cover particular functions and for some particular class of microprocessors,

3. vectorizing compilers - ideally, high level language compiler would be able to automatically identify parallelizable sections of code and generate appropriate SIMD instructions. There have been many proposed methods of automatic SIMD vectorization ([1], [4], [8]) but they have only limited success ([6]).

Programming in high level languages and relying on the compiler to produce the SIMD code is a much easier way to utilize multimedia extensions. But if we want to use them in high-level programming languages such as C, then we have to add these new facilities in some way to the high-level programming languages.

As a consequence of the above we decided to extend the syntax of C and to redefine the existing semantics in such a way that we could use multimedia processing facilities in C. The goal was to provide programmers with the most natural way of using the multimedia processing facilities in the C language. We named this extended C as MMC (MultiMedia C). The MMC was first introduced in the paper [2]. Readers are suggested to refer to this paper for the more extensive description of the language syntax.

This paper is organized as follows: in Section 2 we describe the MMC programming language, in Section 3 we describe the implementation of the MMC compiler, in Section 4 we give real examples from multimedia applications and the performance results.

2 The MMC Language

MMC language is an upward extension of the ANSI C language with multimedia processing facilities. It keeps all the ANSI C syntax plus the syntax rules for vector processing.

2.1 Access to the array elements

To access the elements of an array or a vector we can use one of the following expressions:

1. `expression[expr1]` - with this expression we can access the `expr1`-th element of an array object `expression`. Here, the `expr1` is an integral expression and `expression` has a type "array of type".
2. `expression[expr1:expr2, expr3:expr4]` - with this expression we can access the bits `expr4` through `expr3` of the elements `expr2` to `expr1` of an array object `expression`. Here, the `expr1`, `expr2`, `expr3`, `expr4` are integral expressions and `expression` has a type "array of type". The `expr1` denotes the last accessed element, `expr2` denotes the first accessed element, `expr3` denotes the last accessed bit and `expr4` denotes the first accessed bit.
3. `expression[,expr1:expr2]` - with this expression we can access the bits `expr1` through `expr2` of all the elements of an array object `expression`. Here, the `expr1` and `expr2` are integral expressions and `expression` has a type "array of type". The `expr1` denotes the last accessed bit and `expr2` denotes the first accessed bit.

4. `expression[]` - with this expression we can access the whole array object `expression`. Here, the `expression` has a type "array of type".

2.2 Operators

Unary operators. We extended the semantics of the existing ANSI C unary operators `&, *, +, -, ~, !` in the sense that they may now have both scalar- and vector-type operands.

We have also added new reduction unary operators `[+], [-], [*], [&], [|], [^ ]`. These operators are overloaded existing binary operators `+, -, *, &, |, ^` and are only applicable to the vector operands. These operators perform the given arithmetic/logic operation between the components of the given vector. The result is always a scalar value.

We have also added one new vector operator `|/`, which calculates the square root of each component in the vector.

Binary operators. We have extended the semantics of the existing ANSI C binary operators and the assign operators in such a way that they can now have vector operands. Thus, one or both operands can have an array type.

We have overloaded the existing binary operators with 4 new operators:

?	this operator overloads the binary operators in such a way that the given binary operator performs the operation with saturation,
@	this operator overloads the binary add operator in such a way that the given binary operator first performs addition over adjacent vector elements and then averages (shift right one bit) the result.
~	this operator overloads the multiply operator in such a way that the result is the high part of the product,
_	this operator overloads the multiply operator in such a way that the result is the low part of the product.

Besides the existing binary operators we have added one new, binary operator, which we found to be important in multimedia applications. This operator is applicable only on vector operands (if any operand has a scalar type then it is expanded into an appropriate vector strip) and is as follows:

$\lvert - \rvert$	absolute difference (in the grammar denoted as `VEC_SUB_ABS`).

Example 1. As saturated arithmetic is widely used in multimedia programs (especially in image processing) and as there should be a mechanism to efficiently

deal with multiple possible overflows in packed values, the operations that support saturated arithmetic have been added to microprocessors' instruction set. Since C semantics does not support saturated arithmetic as native operators, programmers are forced to express saturated operations in native C operations.

Figure 1 gives such an example (taken from Berkeley Multimedia workload [7], [6]). The code presented in the Figure 1 could not be efficiently vectorized by an automatic vectorizer. Thus, this portion of parallelism could not be efficiently utilized on multimedia extended processors.

```
/* short a, b; int ltmp; */
#define GSM_ADDS(a, b) \
          ((unsigned)((ltmp=(int)(a)+(int)(b)) - MINWORD) > \
          MAXWORD - MINWORD ? (ltmp>0 ? MAXWORD : MINWORD) : ltmp)

#define MAXWORD 127
#define MINWORD -128
#define N 16

//implementation:

  for(i = 0; i<N; i++ ){
    rez[i] = GSM_ADDS(A[i], B[i]);
  }
```

Fig. 1. C Implementation of saturated add operation from Berkeley Multimedia Workload/GSM.

Figure 2 gives the MMC code for the same saturated add operation. The saturation is now easily expressed in native MMC operations.

```
char A[16];
  char B[16];
  char rez[16];

  rez[] = A[] ?+ B[];
```

Fig. 2. Implementation of saturated arithmetic in MMC.

Conditional expression. The conditional operator from ANSI C `'?:'` which is used in the conditional expression can now have array-type operands.

3 Implementation of the MMC Compiler

The laboratory version of the MMC compiler was implemented for Intel Pentium III and Intel Pentium IV processors. It was implemented as a translator to ordinary C code that is then compiled by an ordinary C compiler (in our example with Intel C++ Compiler for Linux [9]).

The MMC compiler parses input MMC code, performs syntax and semantics analysis, builds its internal representation, and finally translates the internal representation into ANSI C with macros written in a particular assembly language instead of the MMC vector statements. The compilation process is presented in Figure 3. After syntax and semantic analysis of the MMC source code the list

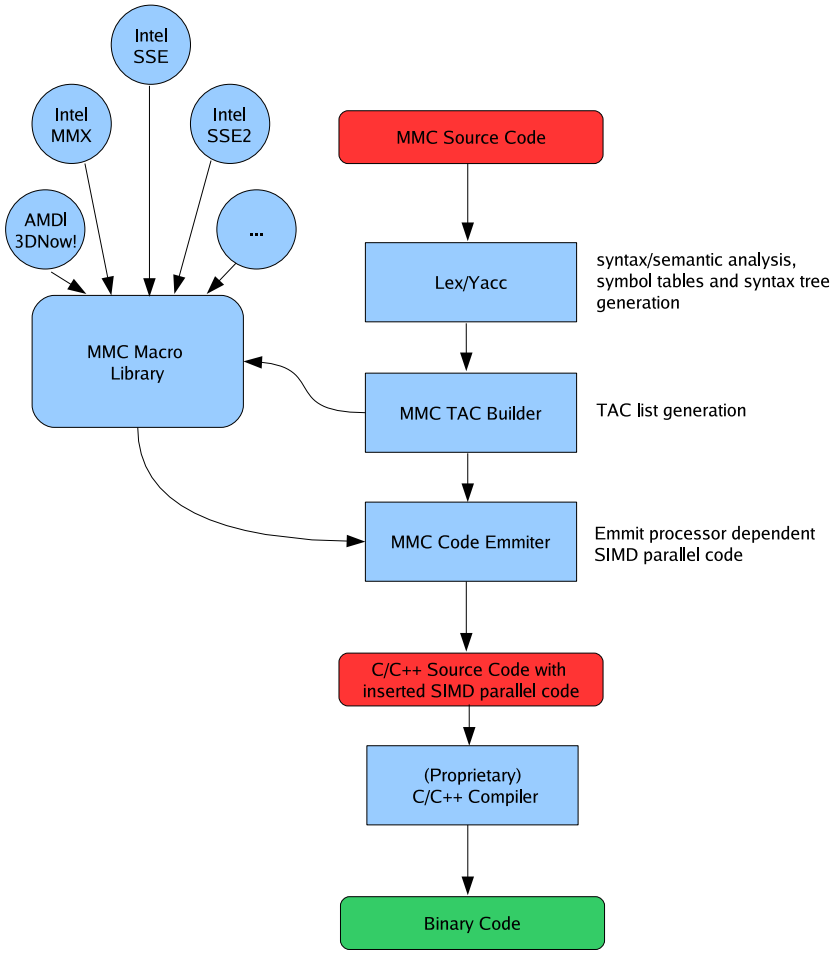

Fig. 3. Compilation process of the MMC source.

of tree-address codes (TAC) for SIMD statements is generated. Then, the MMC Code Emitter inserts SIMD macros for each TAC. The appropriate macro is taken from the MMC macro library. Here we will only show the implementation of one macro for conditional assignment:

```
MMC_QUEST_INT;
*( ($T *)($1) ) = _
MMC_QUEST_SSE2_INT ( *( ($T *)($2) ) ,
                     *( ($T *)($3) ) ,
                     *( ($T *)($4) ));

__m128i _MMC_QUEST_SSE2_INT (__m128i ab,
                             __m128i c,
                             __m128i d )
{
__m128i rez1;
__m128i rez2;
__m128i tmp=_mm_set1_epi32(0);
tmp=_mm_cmpeq_epi32(tmp,ab);

rez1=_mm_and_si128(ab,c);
rez2=_mm_and_si128(tmp,d);

return _mm_or_si128(rez1,rez2);
}
```

The whole macro library, source code of the MMC compiler with Doxygen documentation can be freely downloaded from the MMC web site.

4 Developing Multimedia Kernels

In this section we present the use of MMC language to code some commonly used multimedia kernels. At the end of this section the performance results for the given examples are presented.

Example 2. Finite impulse response (FIR) filters are used in many aspects of present-day technology because filtering is one of the basic tools of information acquisition and manipulation. FIR filters can be expressed by the equation:

$$y(n) = \sum_{k=0}^{N-1} h(k) \cdot x(n-k) \tag{1}$$

where N represents the number of filter coefficients $h(k)$ (or the number of delay elements in the filter cascade), $x(k)$ is the input sample and $y(k)$ is the output sample.

Structurally, FIR filters consist of just two things: a sample delay line and a set of coefficients. To implement the filter one has to:

1. Put the input sample into the delay line.
2. Multiply each sample in the delay line by the corresponding coefficient and accumulate the result.
3. Shift the delay line by one sample to make room for the next input sample.

The MMC implementation of the above algorithm for the FIR filter is as follows:

```
#define FILTER_LENGTH 1024
#define SIGNAL_LENGTH 8192
int j;
double h[FILTER_LENGTH];
double delay_line[FILTER_LENGTH];
double x[SIGNAL_LENGTH];
double y[SIGNAL_LENGTH];

for (j=0; j<SIGNAL_LENGTH; j++) {
  delay_line[0] = x[j];

  //calculate FIR:
  y[j] = [+] ( h[] * delay_line[] );

  //shift delay line:
  delay_line[] = delay_line[] << 1;
}
```

This MMC code is translated by the MMC compiler into C code with inserted macros. So, after strip-mining and macro insertion, which is done by the MMC compiler, we have C code like in the Figure 4. The compiled code can now be further compiled into binary code by the use of C/C++ compiler for desired processor family.

Example 3. An Infinite Impulse Response (IIR) filter produces an output, $y(n)$, that is the weighted sum of the current and the past inputs, $x(n)$, and past outputs. IIR filters can be expressed by the equation:

$$y(n) = \sum_{k=0}^{N-1} h(k) \cdot x(n-k) + \sum_{p=1}^{M-1} h'(p) \cdot y(n-p) \tag{2}$$

where N represents the number of forward-filter coefficients $h(k)$ (or the number of delay elements in the forward-filter cascade) and M represents number of backward-filter coefficients $h'(k)$ (or the number of delay elements in the backward-filter cascade), $x(k)$ is the input sample and $y(k)$ is the output sample.

To implement the IIR filter one has to:

1. Put the input sample into the input delay line, and the output sample into the output delay line.

```
#include <mmintrin.h>
#include <xmmintrin.h>
#include <emmintrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void main()
{
  int j;
  __declspec(align(16))  float h[1024];
  __declspec(align(16))  float delay_line[1024];
  __declspec(align(16))  float x[8192];
  __declspec(align(16))  float y[8192];

  for(j=0; j<8192; j++)
  {
    delay_line[0] = x[j];
  float __mmc_internal_symbol_2 = 0;
  __declspec(align(16)) float __mmc_internal_symbol_1[1024] = {0};

  int __mmc_internal_symbol_3;
  for(__mmc_internal_symbol_3=0; __mmc_internal_symbol_3<1024; __mmc_internal_symbol_3+=4)
  {
    *((__m128 *)(__mmc_internal_symbol_1 + __mmc_internal_symbol_3 + 0)) = _mm_mul_ps( *((__m128 *)(h +
__mmc_internal_symbol_3 + 0)), *((__m128 *)(delay_line + __mmc_internal_symbol_3 + 0)) );
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3 + 0];
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3 + 1];
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3 + 2];
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3 + 3];
    y[j] = __mmc_internal_symbol_2;
  }

  int __mmc_internal_symbol_6;
    for(__mmc_internal_symbol_6=22; __mmc_internal_symbol_6>=0; __mmc_internal_symbol_6--)
  {
    delay_line[__mmc_internal_symbol_6 + 1] = delay_line[__mmc_internal_symbol_6 + 0];
  }

  }
}
```

Fig. 4. Compiled MMC source of the FIR filter.

2. Multiply each sample in the delay line(s) by the corresponding coefficient and accumulate the result.
3. Shift the delay lines by one sample to make room for the next input or output sample.

The MMC implementation of the above algorithm for the IIR filter is as follows (note that for simplicity in implementation we use the $h'(0)$ coefficient, which is always zero):

```
int j;
float hf[FILTER_LENGTH_F];
float hb[FILTER_LENGTH_B];
float in_delay[FILTER_LENGTH];
float out_delay[FILTER_LENGTH];
float x[SIGNAL_LENGTH];
float y[SIGNAL_LENGTH];
```

```
for (j=0; j<SIGNAL_LENGTH; j++) {
  in_delay[0] = x[j];

  //calculate FIR:
  y[j] = [+] ( hf[] * in_delay[] );

  out_delay[0] = y[j];

  //calculate IIR:
  y[j] += ([+]( hb[] * out_delay[]))

  //shift delay lines:
  in_delay[] = in_delay[] << 1;
  out_delay[] = out_delay[] << 1;
}
```

Example 4. The MPEG audio standard uses Discrete Cosine Transformation (DCT) to transform samples from one domain into another. DCT is defined as a linear transformation of N input samples, $s[k]$, and N DCT samples , $x[i]$ where $k = 0 \ldots K-1$ and $i = 0 \ldots K-1$ (see Equation 3).

$$x(i) = \sum_{k=0}^{N-1} s(k) \cdot \cos \frac{(2k+1) \cdot i \cdot \pi}{2N} \tag{3}$$

The DCT formula can also be expressed in matrix form as:

$$\boldsymbol{x} = \mathbf{D} \cdot \boldsymbol{s} \tag{4}$$

where $\boldsymbol{x}$ is the vector of N DCT samples and $\boldsymbol{s}$ is the vector of N input samples. $\mathbf{D}$ is an N by N matrix with the elements presented in Equation 5.

$$D_{i,j} = \cos \frac{(2j+1) \cdot i \cdot \pi}{2N} \tag{5}$$

The matrix representation is used for practical implementation. The matrix representation of the DCT algorithm is well suited for MMC code implementation since the regular structure of matrix multiplication fits the SIMD nature. The MMC implementation of the DCT algorithm is as follows:

```
int j;
float D[N*N];
float v[N];
float s[N];
float D_row[N];

for (j=0; j<N; j++) {
  D_row[] = D[j*N : j*N+(N-1)];
```

```
  //calculate j-th DCT sample:
  v[j] = [+] ( D_row[] * s[] );
}
```

This MMC code is translated by the MMC compiler into C code with inserted macros. So, after strip-mining and macro insertion, which is done by the MMC compiler, we have C code like in the Figure 5. The compiled code can now be further compiled into binary code by the use of C/C++ compiler for desired processor family.

```
#include <mmintrin.h>
#include <xmmintrin.h>
#include <emmintrin.h>
#include "MMC_SSE2_MACROS.H"

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void main()
{
  int j;
  __declspec(align(16))  float D[16900];    /* D matrix */
  __declspec(align(16))  float v[130];      /* DCT samples vector */
  __declspec(align(16))  float s[130];      /* output samples vector */
  __declspec(align(16))  float D_row[130];  /* D matrix row */

  for (j=0; j<130; j++) {
    /* D_row[] = D[j* : j*+(N-1)];   */ /* store matrix row into a vector */



    /* calculate j-th DCT sample: */
  float __mmc_internal_symbol_2 = 0;
  __declspec(align(16)) float __mmc_internal_symbol_1[130] = {0};

  int __mmc_internal_symbol_3;
  for(__mmc_internal_symbol_3=0; __mmc_internal_symbol_3<130; __mmc_internal_symbol_3+=4)
  {
    *( (__m128 *)(__mmc_internal_symbol_1 + __mmc_internal_symbol_3 + 0) ) =
            _mm_mul_ps ( *( (__m128 *)(D_row + __mmc_internal_symbol_3 + 0) ) ,
                         *( (__m128 *)(s + __mmc_internal_symbol_3 + 0) ) );
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3 + 0];
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3 + 1];
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3 + 2];
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3 + 3];
    v[j] = __mmc_internal_symbol_2;
  }

  for(__mmc_internal_symbol_3=128; __mmc_internal_symbol_3<130; __mmc_internal_symbol_3++)
  {
    __mmc_internal_symbol_1[__mmc_internal_symbol_3] =
                  D_row[__mmc_internal_symbol_3] * s[__mmc_internal_symbol_3];
    __mmc_internal_symbol_2 += __mmc_internal_symbol_1[__mmc_internal_symbol_3];
    v[j] = __mmc_internal_symbol_2;
  }
  }
}
```

Fig. 5. Compiled MMC source of DCT.

Example 5. This example demonstrates how to implement saturated operations in MMC. Saturated addition of two vectors (i.e. bitmaps) can be expressed in MMC as: :

```
char bits1[SIZE];
char bits2[SIZE];
char bitsDest[SIZE];

...

bitsDest[] = bits1[] ?+ bits2[];

...
```

This MMC code is translated by the MMC compiler into C code with inserted macros. The compiled code is:

```
int __mmc_internal_symbol_2;
for(__mmc_internal_symbol_2=0;
    __mmc_internal_symbol_2<SIZE;
    __mmc_internal_symbol_2+=8)
{
   *((__m64 *)(destBits + __mmc_internal_symbol_2 + 0)) =
      _mm_adds_pi8( *( (__m64 *)(bits1 + __mmc_internal_symbol_2 + 0) ) ,
                    *( (__m64 *)(bits2 + __mmc_internal_symbol_2 + 0) ) );
}
```

Example 6. This example demonstrates how to implement averaging operations in MMC. We can describe an average operation as:

```
A[]@+[] = ((A[0]+[0])>>1)|((A[0]+[0])&1),
                          ...

          ((A[N-1]+[N-1])>>1)|((A[N-1]+[N-1])&1)
```

This is an idiomatic expression that is ussually hard to detect by compilers. The MMC implementation of an average operation is straightforward:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void main()
{
  char x[1024];
  char y[1024];
  char z[1024];

  z[] = x[] @+ y[];
}
```

This MMC code is translated by the MMC compiler into C code with inserted macros. So, after strip-mining and macro insertion, which is done by the MMC compiler, we have C code like in the Figure 6. The compiled code can now be further compiled into binary code by the use of C/C++ compiler for desired processor family.

```
#include <mmintrin.h>
#include <xmmintrin.h>
#include <emmintrin.h>
#include "MMC_SSE2_MACROS.H"

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void main()
{
  __declspec(align(16))  char x[1024];
  __declspec(align(16))  char y[1024];
  __declspec(align(16))  char z[1024];
  __declspec(align(16)) char __mmc_internal_symbol_1[1024] = {0};

  int __mmc_internal_symbol_2;
  for(__mmc_internal_symbol_2=0; __mmc_internal_symbol_2<1024; __mmc_internal_symbol_2+=16)
  {
    *( (__m128i *)(__mmc_internal_symbol_1 + __mmc_internal_symbol_2 + 0) ) =
         _mm_avg_epu8 ( *( (__m128i *)(x + __mmc_internal_symbol_2 + 0) ) ,
                        *( (__m128i *)(y + __mmc_internal_symbol_2 + 0) ) );
    *((__m128i *)(z + __mmc_internal_symbol_2 + 0)) =
         *((__m128i *)(__mmc_internal_symbol_1 + __mmc_internal_symbol_2 + 0));
  }
}
```

Fig. 6. Compiled MMC source of an averaging operation.

In Figure 7 we can see the performance improvement of some typical multimedia cores when using the MMC language instead of C. We have implemented these cores in MMC and C. Then, the MMC code was compiled into C code with the MMC compiler and into binary code with Intel C/C++ compiler. Sequential C sources are compiled into binaries with Intel C/C++ compiler with vectorization switched on.

In cases where vectorization was successful we reached slightly better performance with the Intel vectorizing compiler. This is because the Intel vectorizing compiler performs some additional optimizatins on the vectorized loop [1], [9]. But in cases where vectorization failed (SATURATION, AVERAGE, SUM OF ABS. DIFF.), better performance is reached with MMC. This is because in these three cases idiomatic expressions where used, which were very difficult for a compiler to identify.

5 Conclusion

We have developed a MMC programming language which is able to use hardware-level multimedia execution capabilities. The MMC language is an upward extension of ANSI C and it saves all the ANSI C syntax. In this way it is suitable for

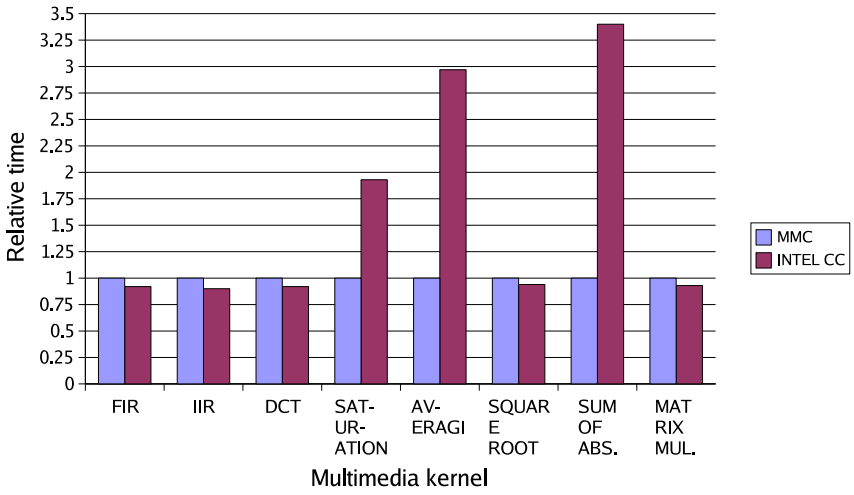

Fig. 7. Speedup on an Intel Pentium IV using MMC.

use by programmers who want to extract SIMD parallelism in a high-level programming language and also by programmers who do not know anything about multimedia processing facilities and who are using the C language.

We have shown the ease with which it is possible to express some common multimedia kernels with MMC. With MMC we can express these kernels in a more straightforward or 'natural' way. The presented extension to C also preserves the interchangeability of arrays and pointers and adds as few as possible new operators. All added operators have an analogue in ordinary C. The declarations of arrays are left unchanged and also no new types have been added.

We obtained good performance for several application domains. Experiments on representive scientific and multimedia applications have significant performance improvements. We are currently rewriting Berkely Multimedia Workload with the MMC language. In such a way we will be able to fully evaluate the performance improvement of widely used multimedia applications. We will also be able to evaluate how difficult is for people to use the MMC language.

References

1. Bik A.J.C., Girkar M., Grey P.M., Tian X.M. Automatic Intra-Register Vectorization for the Intel (R) Architecture. *International Journal of Parallel Programming.* Vol 30., No. 2, pp. 65-98. 2002.
2. Bulić P., Guštin V. An Extended ANSI C for Processors with a Multimedia Extension. *International Journal of Parallel Programming.* Vol 31., No. 2, pp. 107-136. 2003.
3. Ferretti M., Rizzo D. Multimedia Extensions and Sub-Word Parallelism in Image Processing: Preliminary Results. *Lecture Notes in Computer Science*, No. 1685, pp. 977-986, 1999.

4. Krall A., Lelait S. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*. Vol. 28, No. 4, pp. 347-361, 2000.
5. Lee R., Smith M.D. Media Processing: A New Design Target. *IEEE Micro*, Vol. 16, No. 4, pp. 6-9, 1996.
6. Ren G., Wu P., Padua D. A Preliminary Study On the Vectorization of Multimedia Applications for Multimedia Systems. *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computers, October 2-4, 2003, College Station, Texas*. pp. 2-16, 1987.
7. Slingerland N.T., Smith A.J.,. Multimedia extensions for General Purpose Microprocessors: a Survey *Microprocessors and Microsystems*, Vol. 29, pp. 225-246, 2005.
8. Sreraman N., Govindarajan R. A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming*, Vol. 28, No. 4, pp. 363-400, 2000.
9. -. Intel C++ Compiler for Linux 9.0. *http://www.intel.com/software/products/compilers*.
10. -. MMX Technology Application Notes: Using MMX Instructions to Convert RGB To YUV Color Conversion. *http://cedar.intel.com*.
11. -. DSP Guru: Finite Impulse Response FAQ *http://www.dspguru.com/info/faqs/firfaq.htm*.
12. -. DSP Guru: Infinite Impulse Response FAQ *http://www.dspguru.com/info/faqs/iirfaq.htm*.