International Journal of Parallel Programming, Vol. 31, No. 2, April 2003 (© 2003)

An Extended ANSI C for Processors with a Multimedia Extension

Patricio Bulić[1] and Veselko Guštin[1]

This paper presents the Multimedia C language, which is designed for the multimedia extensions included in all modern microprocessors. The paper discusses the language syntax, the implementation of its compiler and its use in developing multimedia applications. The goal was to provide programmers with the most natural way of using multimedia processing facilities in the C language. 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.

KEY WORDS: Vector C; SIMD processing; ISA multimedia extensions.

1. INTRODUCTION

C language is frequently used by professional and system programmers because it allows them to develop highly efficient code. They choose C because it reflects all the main features of a system architecture, which has an impact on the program's efficiency. These features are:

1. machine-oriented data types,
2. indirect addressing and address arithmetic (pointers, arrays,...),
3. bit operations.

On the other hand, C hides from the programmer the peculiarity of each particular processor architecture (register storage and register allocation, stack implementation, details of the instruction set, activation modules, etc.).

[1] University of Ljubljana, Faculty of Computer and Information Science, Tržaška c. 25, SI-1000 Ljubljana, Slovenia. E-mail: {patricio.bulic, veselko.gustin}@fri.uni-lj.si

0885-7458/03/0400-0107/0 © 2003 Plenum Publishing Corporation

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—including super-scalar execution, out-of-order execution, speculative execution[1]—as well as those facilities which require support from external entities (i.e., assembler language and compilers) such as multimedia (also called short vector or SIMD within a register) processing ability[2–12] (i.e., Intel MMX, Intel SSE, Intel SSE2, Motorola Altivec, SUN VIS,...), support for multiprocessor architectures, etc. This was reflected in an extension of the assembly languages (extended instruction set).

But if we want to use them in high-level programming languages such as C, then we have to find some way to add these new facilities in some way to the high-level programming languages. In this paper we focus on porting multimedia extension processing facilities into high-level languages, particularly C.

The increasing need for multimedia applications has prompted the addition of multimedia extensions to most existing general-purpose microprocessors. These extensions introduce short vector or SIMD (*Single Instruction Multiple Data*) instructions to the microprocessor's scalar instruction set. This added instruction set is supported by special hardware that enables the execution of one instruction to packed data within a multimedia register. Such a vector instruction set is primarily used in multimedia applications and it seems that their use will expand rapidly over the next few years.

High-level programming languages do not support this kind of data parallelism. Thus, on the one hand we have modern execution hardware and on the other hand we have high-level programming languages that are not able to use these new facilities. So far we have noticed a number of different approaches that are designed to integrate these new facilities into high-level programming languages:

1. The use of assembly languages within high-level programming languages whenever we want to exploit vector processing. This approach gives us the ability to develop highly efficient multimedia code but it has the disadvantages of a long development time and a reluctance among most programmers to program in assembly language.
2. The addition of some special libraries[6, 13] that have a wide range of vector processing functions coded in assembly language. These libraries are often available from microprocessor manufacturers,

but they tend to only cover particular functions for multimedia processing so we cannot use them for some general vector processing. They are only intended for programmers who develop multimedia software for some particular class of microprocessors (i.e., the Intel family).

3. The use of vectorizing compilers.[14–31] This is a special class of compilers that can parallelize some simple loops within a high-level programming language code. These compilers are, in general, unable to use some special facilities of vector processing mainly because we cannot describe these facilities in high-level programming languages (there are no constructs to describe these special facilities). This is the case with, for example, saturation arithmetic. In ordinary C only expressions with modular arithmetic may be used.
4. The last approach is to extend the syntax and semantics of high-level programming languages and to redefine the semantics of existing operators and expressions. We find that this is the best way to migrate new vector processing facilities into high-level programming languages and we agree with those authors[32–41] who tried to develop such a class of high-level programming languages, although for a different execution model (mainly the large-scale SIMD execution model).

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. We redefined the semantics of the existing operators, we added a few new ones, and we also added some new syntax that enables access to array slices. 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 (Multi-Media C).

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. Finally, in Section 5 we make comparisons with related studies.

2. THE MMC LANGUAGE

MMC language is an extended ANSI C language with multimedia (short vector or SIMD within a register) processing facilities. It keeps all the ANSI C syntax plus the syntax rules for vector processing. It extends

the ANSI C syntax only in the access possibilities for the array elements and in the new vector operators. The syntax notation is mostly based on the notation that was first introduced in Ref. 36.

2.1. Arrays

Let us present some basic definitions for an array (vector) and a vector strip in MMC.

Definition 1. In the MMC language an array (or vector) is a data structure that consists of sequentially allocated elements of the same type with a strictly positive unit step.

Modern processors with multimedia execution hardware have only vector load/store instructions, which can only move sequentially allocated elements between the memory and the microprocessor. Gather/scatter operations are useful, for example, when multiplying matrices because of the different type of access to the elements in two matrices (in one we access to the column elements and in the other matrix we access to the row elements). But also, these operations are very expensive and we believe that, regarding to the existing multimedia execution hardware, it is better to force the programmer to correctly rearrange the array elements (actually, matrix multiplication can be implemented in a way that doesn't need gather-scatter operations). So, the extension of the array definition to the non-sequentially allocated elements (also called non stride-1 vectors) is redundant for this type of execution model.

2.2. Vector Strips

Because of hardware limitations, especially the multimedia execution hardware and the multimedia register set within a microprocessor, not all the lengths of the array components are permitted. So we will define some notations, which we will use throughout this paper and which represent different vector strips.

Definition 2. A vector strip is a subset of an array where all of the components have the same type. These components can be as long as 8 bits (or a byte), 16 bits (or a word), 32 bits (or a doubleword), 64 bits (or a quadword) and 128 bits (or a superword). The size of the vector strip is also constant, it is limited to the length of the multimedia register in a microprocessor, and for most modern microprocessors this length is 64 or 128 bits.

Definition 3. We can define the following possible vector strips:

1. A VB vector strip is an array slice composed of 8(16) byte components.
2. A VW vector strip is an array slice composed of 4(8) word components.
3. A VD vector strip is an array slice composed of 2(4) doubleword components.
4. A VQ vector strip is an array slice composed of 1(2) quadword component(s).
5. A VS vector strip is an array slice composed of 1 superword component.
6. A VSF vector strip is an array slice composed of 4 single-precision floating-point components.
7. A VDF vector strip is an array slice composed of 2 double-precision floating-point components.

2.3. 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."

   Example 1. We define two arrays of 100 integers:

   ```
   int A[100], B[100];
   ```

   We can now access the 46 th element of array A and put its value into the 34 th element of the array B with the expression statement:

   ```
   B[33]=A[45];
   ```

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.

Example 2. We define two arrays of 100 integers:

```
int A[100], B[100];
```

We can now put the upper 16 bits of the first 50 elements of the array A into the lower 16 bits of the last 50 elements of the array B with the expression statement:

```
B[99:50, 15:0]=A[49:0, 31:16];
```

If a programmer specifies something unusual like access to the array[7:3, 11:4], where array is of the byte type, the MMC compiler should divide this operation into several memory accesses (actually, the current laboratory version of the MMC compiler will only report an error). We have enabled such irregular access as we believe that the language should be designed for longevity and "look to the future." If these multimedia operations are to remain important in the future, some sort of bit scatter/gather hardware will become available on many platforms.

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.

Example 3. We define two arrays of 100 integers:

```
int A[100], B[100];
```

We can now move the low-order bits of all the elements in the array A into the high-order bits of all the elements of the array B with the expression statement:

```
B[, 31:16]=A[, 15:0];
```

The operation for one element is shown in Fig. 1.

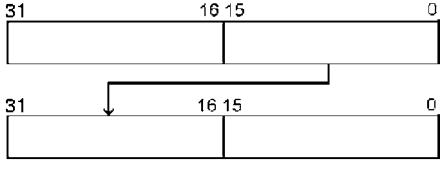

Fig. 1. Move the lower part of A[i] into the upper part of B[i].

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

   Example 4. We define two arrays of 100 integers:

   ```
   int A[100], B[100];
   ```

   We can now copy the array A into the array B with the expression statement:

   ```
   B[]=A[];
   ```

   The operator [] was first introduced in the C[] language as described in Ref. 36. It is called the *block operator* because blocks (forbids) the conversion of the operand to a pointer. We found it suitable to denote the whole array object and thus avoid any possible confusion of arrays with pointers.

To support these new access types we have to redefine the syntax of an array access expression. The postfix expression is defined as:

```
postfix_expression : primary_expression
| postfix_expression '[' expression ']'
| postfix_expression '(' ')'
| postfix_expression '(' argument_expression_list ')'
| postfix_expression '.' IDENTIFIER
| postfix_expression PTR_OP IDENTIFIER
| postfix_expression INC_OP
| postfix_expression DEC_OP
| postfix_expression '[' vector_access_expression ']'
```

and these new productions are added:

```
vector_size_expression : expression ':' conditional_expression
vector_access_expression : vector_size_expression
| ',' vector_size_expression
| vector_access_expression ',' vector_size_expression
```

And finally, we have to rewrite the production for the conditional expression in order to avoid ambiguity:

```
conditional_expression : logical_or_expression
| logical_or_expression '?' vector_size_expression
```

2.4. Operators

2.4.1. 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.

Example 5. First, we declare two arrays and then we assign negative values of the elements from the first array to the elements from the second array:

```
int A[]={1, 2, 3, 4};
int B[4];
  ...
B[]= – A[];
```

After that, array B will have the following values: $\{-1, -2, -3, -4\}$.

We have also, in a similar way to Refs. 36 and 41, 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 binary operation between the components of the given vector. The result is always a scalar value. Again, we believe that [op] notation, which was introduced in Ref. 36, in a more "natural" way indicates that the operation is to be performed over all vector components.

Example 6. First, we declare array A and then we make the sum of all its components:

```
int sum;
int A[]={1, 2, 3, – 4};
  ...
sum=[+] A[];
```

After that, the sum will be 2.

We have also added one new vector operator |/, which calculates the square root of each component in the vector (please note, that this works only with floating-point vectors, although the MMC compiler does not perform any type checking, and if we apply this operator to integer vectors the result my be undetermined).

To achieve these operations in the MMC languge we have to rewrite the grammar production for the unary operators in ANSI C. To the grammar rule for the unary operator we have added the following productions:

```
| VEC_COMPONENT_ADD
| VEC_COMPONENT_SUB
| VEC_COMPONENT_MUL
| VEC_COMPONENT_AND
| VEC_COMPONENT_OR
| VEC_COMPONENT_EXOR
| '|/'
```

where VEC_COMPONENT_ADD denotes the [+] operator, and it is similar for the others.

2.4.2. 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. If both operands are arrays of the same length then the result is an array of the same length (note that the length is measured in the number of components and not in the number of bits!). If one array operand has N elements and another array operand has M elements and $N < M$ then the operation is only performed over N elements. If arrays have different types then the MMC compiler reports an error. If one of the operands is of the scalar type then it is internally converted by the MMC compiler into a vector strip of the corresponding type and length. This type of element in the vector strip and its length strongly depends on the processor for which we compile our program. For example, if the array operand consists of word components then for the Intel Pentium processor the scalar operand is converted into the VW vector strip (vector of four 16-bit values).

Example 7. We can now make the sum of two arrays:

```
short A[4]={1, 2, 3, 4};
short B[4]={4, 3, 2, 1};
short C[4]={0, 0, 0, 0};
short d=7;
  ...
C[]=A[]+B[];
A[]=d+A[];
```

Note, that the + operator has both operands with an array type. Now, the components from array C will have the following values: (5, 5, 5, 5) and the components from array A will have the following values: (8, 9, 10, 11). Variable d is expanded internally by the MMC compiler into the vector [0007h, 0007h, 0007h, 0007h].

Example 8. The Intel SIMD instruction PMADDWD first carries out the component-wise product of the integers, and then, second, makes the sum of the products into an integer. This can also be written in the MMC language as:

```
short A[4], B[4];
int R[2];
  ...
R[1:1, 31:0]=[+] ( A[2:1, 15:0] * B[2:1, 15:0] );
R[2:2, 31:0]=[+] ( A[4:3, 15:0] * B[4:3, 15:0] );
```

We have overloaded the existing binary operators with 3 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. Let A[] and B[] be two vectors of the same dimension N and the same type. Then expression A[] @+ B[] has the same semantics as the expression (A[0]+B[0])/2},..., (A[N – 1]+B[N – 1])/2,
- ~ (tilde) this operator overloads the multiply operator in such a way that the result is the high part of the product,
- _ (underscore) this operator overloads the multiply operator in such a way that the result is the low part of the product.

Thus, we may have the following operations:

?+ for add with saturation (in the grammar denoted as VEC_ADD_SAT),

?– for subtract with saturation (in the grammar denoted as VEC_SUB_SAT),

@+ for average add (in the grammar denoted as VEC_ADD_AVG),

~* multiply, the result is the high part of the product (in the grammar denoted as VEC_MUL_HI),

_* multiply, the result is the low part of the product (in the grammar denoted as VEC_MUL_LO).

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:

|-| absolute difference (in the grammar denoted as VEC_SUB_ABS).

Example 9. The Intel SIMD instruction PADDSB makes the sum between two unsigned byte vectors (VB) and uses saturation arithmetic. This can also be written in the MMC language as:

```
unsigned char A[8], B[8];
unsigned char C[8];
  ...
C[]=A[] ?+B[] ;
```

Example 10. The Intel SIMD instruction PSADBW computes the absolute differences of the packed unsigned byte vector strips (VB). Differences are then summed to produce an unsigned word integer result. This can also be written in the MMC language as:

```
unsigned char A[8], B[8];   /* components are 8 bits long */
unsigned short c;
  ...
c=[+] (A[] |-| B[]) ;
```

Thus, to the grammar rule for the multiplicative expression in the ANSI C language we have added the following productions:

multiplicative_expression VEC_MUL_HI cast_expression
| multiplicative_expression VEC_MUL_LO cast_expression

and to the grammar rule for the additive expression in the ANSI C language we have added these productions:

additive_expression VEC_ADD_SAT multiplicative_expression
| additive_expression VEC_SUB_SAT multiplicative_expression
| additive_expression VEC_ADD_AVG multiplicative_expression
| additive_expression VEC_SUB_ABS multiplicative_expression

2.4.3. Conditional Expression

The conditional operator **?:** which is used in the conditional expression can now have array-type operands. If the first operand is a scalar or an array and the second and third are arrays then the result operand has the same array type as both operands. If the array operands have different array lengths or different types of components then the behavior of the conditional expression is undetermined. If the second or third operand is scalar then it is converted into a vector (the same conversion as for binary operators). If all the operands are arrays of the same length the operation is performed component-wise.

Example 11. The Intel SIMD instruction PMAXUB returns the greater vector components between two byte vectors (VB). This can also be written in the MMC language as:

```
int A[100, 8], B[100, 8];   /* components are 8 bits long */
int C[100, 8];
   ...
C[]=(A[] > B[]) ? A[] : B[];
```

Tables I and II summarize the multimedia instruction set supported by the Intel, Motorola and SUN processor families and the associated MMC expression statements.

3. IMPLEMENTATION OF THE MMC COMPILER AND PORTABILITY ISSUES

The laboratory version of the MMC compiler is implemented for Intel Pentium III and Intel Pentium IV processors. It is implemented as a

Table I. Relations Between Integer Multimedia Instructions and MMC Expressions

MMC expression	Intel MMX	SUN UltraSpark VIS	Motorola Altivec PowerPC
R[]=A[]+B[];	PADD[B | W | D]	vis_fadd[16 | 32]	vec_add[8 | 16 | 32]
R[]=A[]?+B[]; signed	PADDS[B | W]		vec_adds[8 | 16 | 32]
R[]=A[]?+B[]; unsigned	PADDUS[B | W]		
R[]=A[] – B[];	PSUB[B | W | D]	vis_fpsub[16 | 32]	vec_sub[8 | 16 | 32]
R[]=A[]? – B[]; signed	PSUBS[B | W | D]		
R[]=A[]? – B[]; unsigned	PSUBUS[B | W | D]		
R[]=A[] ~* B[]; signed	PMULHW		
R[]=A[] ~* B[]; unsigned	PMULHUW		
R[]=A[] _* B[]; signed	PMULLW		
R[]=A[] _* B[]; unsigned	PMULLUW		
A[]=A[] ≪ count;	PSLL[W | D]		vec_sl[8 | 16 | 32]
A[]=A[] ≫ count;	PSRA[W | D]		vec_sra[8 | 16 | 32]
A[]=A[] ≫ count;	PSRL[W | D]		vec_sr[8 | 16 | 32]
R[]=A[] op B[];[a]	POR, PAND, PXOR, PANDN		vec[_or | _and | _xor]64
R[]=(A[]==B[])? 0xFF : 0;	PCMPEQ[B | W | D]	vis_fcmpeq[16]	vec_cmpeq[8 | 16 | 32]
R[]=(A[] > B[])? 0xFF : 0;	PCMPGT[B | W | D]	vis_fcmpgt[16]	vec_cmpgt[8 | 16 | 32]
R[]=A[] > B[] ? A[] : B[];	PMAXU[B | W]		vec_max[8 | 16 | 32]
R[]=A[] < B[] ? A[] : B[];	PMINU[B | W]		vec_min[8 | 16 | 32]
R[]=A[] @+B[];	PAVG[B | W]		vec_avg[8 | 16 | 32]
R[]=A[] |-| 0;			vec_abs[8 | 16 | 32]
R=[+] (A[] |-| B[]); unsigned	PSAD[B | W]		

[a] op={ |, &, ^, ~&}.

Table II. Relations Between Floating-Point Multimedia Instructions and MMC Expressions

MMC expression	Intel SSE	SUN VIS	Motorola Altivec
R[]=A[]+B[];	ADDPS		vec_add
R[]=A[] – B[];	SUBPS		vec_sub
R[]=A[] * B[];	MULPS		
R[]=A[] / B[];	DIVPS		
R[]=1 / A[];	RCPPS		vec_rc
R[]=|/^A[];	SQRTPS		
R[]=1 / (|/^A[]);	RSQRT		vect_rsqrte
R[]=A[] log_op B[];[a]	POR, PAND, PXOR, PANDN		vec_or | _and | _xor | _andn
R[]=(A[] rel_op B[])? 0xFF : 0;	CMPPS, rel_op1[b]		vec_cmp [rel_op2][c]
R[]=A[] > B[] ? A[] : B[];	MAXPS		vec_max[8 | 16 | 32]
R[]=A[] < B[] ? A[] : B[];	MINPS		vec_min[8 | 16 | 32]

[a] log_op={ |, &, ^, ~& }.
[b] rel_op1={==, < , <=, !=, !< , !<=, !? }.
[c] rel_op2={==, < , <=, !=, >, >= }.

translator to ordinary C code that is then compiled by an ordinary C compiler (in our example with Intel C++ Compiler for Linux[13]).

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 Fig. 2.

If we want to compile for another class of microprocessor, we have to use another macro library that is written for that particular class of microprocessor. In such a way we can easily port the programs to another machine. The macro libraries for different processors are easily written and all lower-level optimization of the code is done by an ordinary C compiler

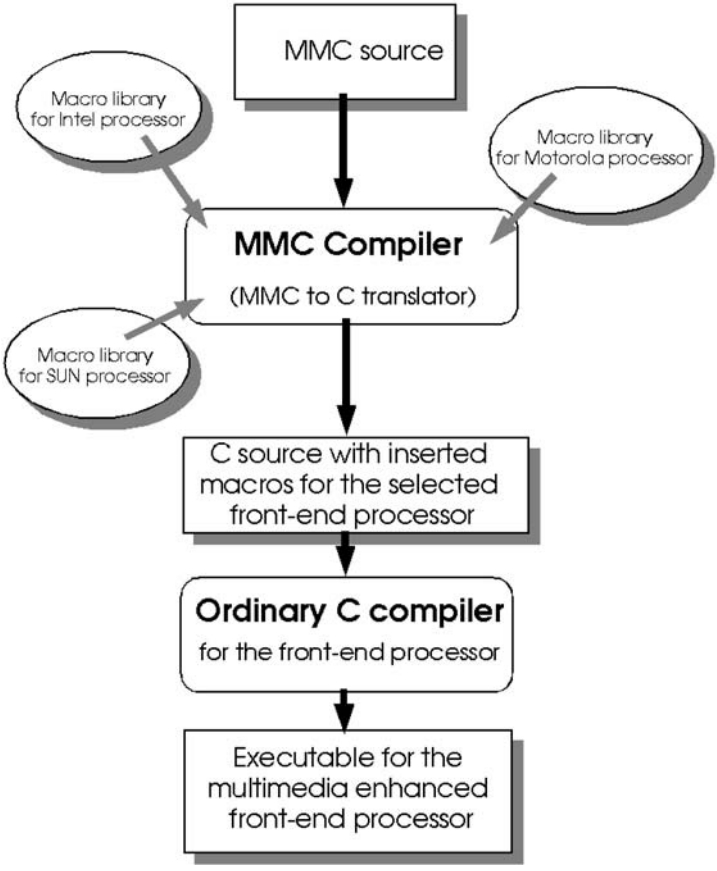

Fig. 2. Compilation of the MMC source.

for a particular microprocessor. In the event that a particular microprocessor does not support some parallel operation written in MMC with special multimedia machine instruction(s) we use a function written in C that executes sequentially instead of multimedia macro.

Here we will only show the macro library for the integer operations for the Intel Pentium class of microprocessor and that is used by the MMC compiler to translate MMC code into ordinary C code. The macro library itself is written in assembly language, and can be listed in some separate sets:

1. Arithmetic and logic instructions on vectors:

   (a) vectors A+B, A ?+ B to RESULT:

   ```
   ADD[B | W | D | B | UW | UD | USB | USW | USD]
   (RESULT, A, B);
   ```

   (b) vectors A – B, A ? – B to RESULT:

   ```
   SUB[B | W | D | UB | UW | UD | USB | USW | USD]
   (RESULT, A, B);
   ```

   (c) vectors A & B, !A & B, A | B, A ∇ B to RESULT:

   ```
   AND[Q | O] (RESULT, A, B);,
   ANDN[Q | O] (RESULT, A, B);,
   OR[Q | O] (RESULT, A, B);,
   EXOR[Q | O] (RESULT, A, B);
   ```

   (d) vectors A _* B, A ~* B to RESULT:

   ```
   MULL[W | UW] (RESULT, A, B);, MULH[W | UW]
   (RESULT, A, B);
   ```

   (e) `AVG[B | W] (RESULT, A, B);` // ([+](A-B))/2 to RESULT

   (f) `SRL, SLL[W | D | Q] (A, NOSHIFTS);` // shift R/L vector A for NOSHIFTS

   (g) `SRA[W | D] (A, NOSHIFTS);` // shift R vector A for NOSHIFTS

2. Miscellaneous instruction set:

   (a) `SUMMULTWD (RESULT, A, B);` // [+](A*B) to scalar RESULT

(b) SUMABSDIFFBW
(RESULT, A, B); // [+](A – B) to scalar RESULT

3. Compound (control) instructions:

(a) IFMGT[B | W | FP]
(MASK, A, B, R); // R=(MASK > A) ? A : B

(b) IFMEQ[B | W | FP]
(MASK, A, B, R); // R=(MASK==A) ? A : B

(c) IFGT[B | W | FP]
(A, B, D, E, R); // R=B > A) ? D : E

(d) IFEQ[B | W | FP]
(A, B, D, E, R); // R=(B==A) ? D : E

The complete macro library can be downloaded from http://lra-1.fri.uni-lj.si/vect/MacroVect.c.

The library MacroVect.c is under development, and during its use we will be able to evaluate its pros and cons. From this perspective we have developed a general set of mostly usable macros.

Example 12. The MMC statement:

```
R[]=(A[] > MASK) ? A[] : B[];
```

is evaluated during the compilation process into macro IFGTB(MASK, A, B, R); which is defined as:

```
#define IFMGTB(MASK, A, B, R); __asm{ mov eax, B \
  movq mm3, [eax] \
  mov ebx, A \
  movq mm2, [ebx] \
  mov ecx, MASK \
  movq mm1, [ecx] \
  pcmpgtb mm1, mm2 \
  movq mm4, mm1 \
  pand mm1, mm3 \
  pandn mm4, mm2 \
  por mm1, mm4 \
  mov edx, R \
  movq [edx], mm1 };
```

In Section 4 we can see the use of this macro in a real application.

4. THE USE OF THE MMC LANGUAGE TO DEVELOP MULTIMEDIA APPLICATIONS

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. Examples 13 through 15 show how the MMC code is translated by the MMC compiler into C code.

Example 13. We mix the two images, one a live picture from the video camera, and the second, the background map, which is in the memory. The principle of mixing is very simple, and is as follows:

```
if (Threshold > Live_picture) then Show = Background;
else Show=Live_picture;
```

As the color white is '0xFF' we put the Threshold a little lower, i.e., about '0xF0'. The above statement we rewrite in the MMC language:

```
char Show[NOITEMS];
char Live_picture[NOITEMS];
char Background_map[NOITEMS];
char Live_picture[NOITEMS];
  ...
Show[]=Thr > Live_picture[] ? Background_map[] : Live_picture[];
```

Where the constant Thr is 0xF0.

These statements in MMC are translated into the C code in such a way that the used scalar program can run in vectored mode:

```
// create new symbols for loop indexes:
int i00001, i00002;
// expand Thr into vector:
int ThrMASK[8]={Thr, Thr, Thr, Thr, Thr, Thr, Thr, Thr} ;

for( i00001=0; i00001 < NOITMES/8; i00001+=8 ) {
  A=Live_picture+i00001;      // prepare addresses for macro
  B=Background_map+i00001;
```

```
    R=Show+i00001;
    D=ThrMASK;
    IFMGTB(D, A, B, R);      // macro insertion
}

for( i00002=NOITMES/8; i00002 < NOITEMS; i00002++) {
    if (Thr > Live_picture[i00002]) {
        Show[i00002]=  Background[i00002];
    }
    else Show[i00002]=Live_picture[i00002];
}
```

We used our MMC for processing a b/w signal from a video frame-grabber. The processing kernel is the mixing function presented in the previous example. For a comparison we first used the program written in ANSI C and then the program written in the MMC language. In Table III we can see the results of both tests, the number of instructions, and the improvement of execution times for processing the array of 442368 bytes, shorts, and integers. We made the test on the Pentium III microprocessor.

We see that the execution time improves by about 30% on larger arrays (int), and by about 100% on smaller arrays (byte), even though the total number of instructions is greater.

Example 14. 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:

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

Table III. Execution Results. The Execution Time for the Code Written in ANSI C with Byte Arrays and Compiled with the Intel C++ Compiler Is Normalized to 1

Example in / Compiled with	Number of total instructions	Time byte	Time short	Time int
C / Intel C++	22	1	1.6	1.7
MMC / MMCC, Intel C++	22+9 SIMD	0.65	1.37	1.65

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. The MMC implementation of the FIR filter is as follows:

```
int j;
double h[FILTER_LENGTH];   // FIR filter coefficients
double delay_line[FILTER_LENGTH];   // delay line
double x[SIGNAL_LENGTH];   // input signal
double y[SIGNAL_LENGTH];   // output signal

for (j=0; j < SIGNAL_LENGTH; j++) {
   delay_line[0]=x[j];   // store input in the delay line

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

   // shift delay line:
   delay_line[FILTER_LENGTH:1]=delay_line[FILTER_LENGTH-1:0];
}
```

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:

```
int j;
float h[FILTER_LENGTH];   // FIR filter coefficients
float delay_line[FILTER_LENGTH];   // delay line
float x[SIGNAL_LENGTH];   // input signal
float y[SIGNAL_LENGTH];   // output signal

// create new symbols:
int i00001, i00002, i00003;

for (j=0; j < SIGNAL_LENGTH; j++) {
   z[0]=x[j];   // store input in the delay line
```

```
    // calculate FIR:
    // strip mining and macro insertion:
    for( i00001=0; i00001 < FILTER_LENGT/4; i00001+=4 ) {
      H=h+i00001;   // prepare addresses for macro
      Z=delay_line+i00001;
      OUT=y+j;

      SUMMULTWD(OUT, H, Z);   // macro insertion
    }

    for( i00002=FILTER_LENGTH/4;
         i00002 < FILTER_LENGTH; i00002++) {
      y[j]=y[j]+(h[i00002] * delay_line[i00002]) ;
    }

    // shift delay line:
    for( i00003=FILTER_LENGTH-2; i00003 >=0; i00003 - - ) {
      delay_line[i00003+1]=delay_line[i00003] ;
    }
  }
```

Example 15. 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:

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

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.

The MMC implementation of 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];   // forward IIR filter coefficients
float hb[FILTER_LENGTH_B];   // backward IIR filter coefficients
float in_delay[FILTER_LENGTH];   // input delay line
float out_delay[FILTER_LENGTH];   // output delay line
float x[SIGNAL_LENGTH];   // input signal
float y[SIGNAL_LENGTH];   // output signal

for (j=0; j < SIGNAL_LENGTH; j++) {
   in_delay[0]=x[j];   // store input in the delay line

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

   out_delay[0]=y[j];   // store output into the delay line

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

   // shift delay lines
   in_delay[FILTER_LENGTH_F:1]=in_delay[FILTER_LENGTH_F-1:0];
   out_delay[FILTER_LENGTH_B:1]=out_delay[FILTER_LENGTH_B-1:0];
   out_delay[0]=y[j];
}
```

This MMC code is translated into C as follows:

```
    int j;
    float hf[FILTER_LENGTH_F];   // forward IIR filter coefficients
    float hb[FILTER_LENGTH_B];   // backward IIR filter coefficients
    float in_delay[FILTER_LENGTH];   // input delay line
    float out_delay[FILTER_LENGTH];   // output delay line
    float x[SIGNAL_LENGTH];   // input signal
    float y[SIGNAL_LENGTH];   // output signal
```

```
// create new symbols:
int i00001, i00002, i00003, i00004, i00005;
float temp00001;

for (j=0; j < SIGNAL_LENGTH; j++) {
  in_delay[0]=x[j];   // store input in the delay line

  // calculate FIR:
  // strip mining and macro insertion:
  for( i00001=0; i00001 < FILTER_LENGTH_F/4; i00001+=4 ) {
    HF=hf+i00001;   // prepare addresses for macro
    IND=in_delay+i00001;
    OUT=y+j;

    SUMMULTWD(OUT, HF, IND);   // macro insertion
  }

  for( i00002=FILTER_LENGTH_F/4;
       i00002 > FILTER_LENGTH_B; i00002++) {
    output[j]=output[j]+(hf[i00002] * in_delay[i00002]) ;
  }

  // calculate IIR:
  // strip mining and macro insertion:
  for( i00003=0; i00003 < FILTER_LENGTH_B/4; i00003+=4 ) {
    HB=hb+i00001;   // prepare addresses for macro
    OUTD=out_delay+i00001;

    SUMMULTWD(temp00001, HB, OUTD);   // macro insertion
  }

  y[j]=y[j]+temp00001;
```

```
    for( i00004=FILTER_LENGTH_B/4;
         i00004 < FILTER_LENGTH; i00004++) {
      y[j]=y[j]+(hb[i00004] * out_delay[i00004]) ;
    }

    // shift delay lines:
    for( i00005=FILTER_LENGTH_F-2; i00005 >=0; i00005 - - ) {
      in_delay[i00005+1]=in_delay[i00005] ;
    }
    for( i00005=FILTER_LENGTH_B-2; i00005 >=0; i00005 - - ) {
      out_delay[i00005+1]=out_delay[i00005] ;
    }
  }
```

Example 16. 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 $k = 0 \cdots K-1$ and $i = 0 \cdots K-1$:

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

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

$$\vec{x} = \mathbf{D}\vec{s}$$

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

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

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];   // D matrix
float v[N];   // DCT samples vector
float s[N];   // output samples vector
float D_row[N];   // D matrix row

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

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

Example 17. This example demonstrates how to convert RGB color space pixels to YUV color space pixels. Components of the YUV color space are linear combinations of the components of the RGB color space. Therefore, RGB-to-YUV color conversion is computed by multiplying a 3×3 coefficient matrix by a vector of RGB values. The RGB-to-YUV color conversion equation is taken from Ref. 42 and is as follows:

$$\begin{bmatrix} \mathrm{Y} \\ \mathrm{U} \\ \mathrm{V} \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.146 & -0.288 & 0.434 \\ 0.617 & -0.517 & -0.100 \end{bmatrix} \cdot \begin{bmatrix} \mathrm{R} \\ \mathrm{G} \\ \mathrm{B} \end{bmatrix}$$

The MMC implementation of the above equation is as follows:

```
float R[VECTOR_SIZE];
float G[VECTOR_SIZE];
float B[VECTOR_SIZE];
float Y[VECTOR_SIZE];
float U[VECTOR_SIZE];
float V[VECTOR_SIZE];

float matrix[]={0,299, 0,587, 0.114,
                -0.146, -0.288, 0.434,
                0.617, -0.517, -0.100};
```

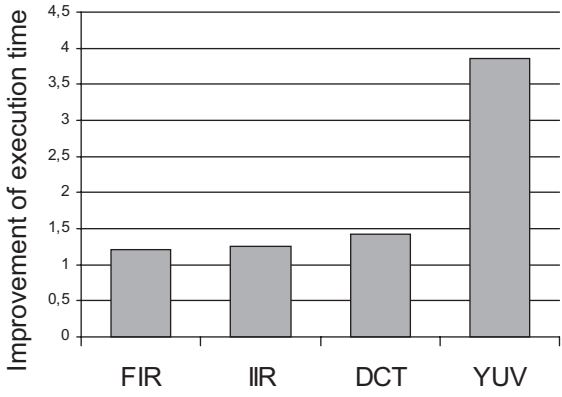

Fig. 3. Speedup on an Intel Pentium III using MMC.

```
Y[]=R[] * matrix[0]+G[] * matrix[1]+B[] * matrix[2] ;
U[]=R[] * matrix[3]+G[] * matrix[4]+B[] * matrix[5] ;
V[]=R[] * matrix[6]+G[] * matrix[7]+B[] * matrix[8] ;
```

Note that the matrix coefficients are expanded by the MMC compiler into VSF vector strips.

In Fig. 3 we can see the performance improvement when using the MMC instead of the ANSI C for the given examples. Both codes, MMC and ANSI C, were finally compiled with an Intel C++ Compiler, and executed on an Intel Pentium III personal computer.

5. COMPARISONS WITH OTHER STUDIES

The **Vector C language**[38, 39] was designed and implemented on the CDC Cyber 205 at Purdue University. Vector C extends C by allowing arrays, in effect, to be treated as first-class objects (vectors) by using a special subscripting syntax to select array slices. Vector C targets general vector machines with many vector processing facilities that multimedia-enhanced processors do not have. On the other hand, the operators in Vector C do not cover all processing facilities that are present in multimedia-enhanced processors. The syntax of Vector C allows periodic scatter/gather operations and compress/expand operations. Two new data types, the vector descriptor (which acts as a pointer to the array but is extended in such a way that it can handle non-stride-1 vectors) and the bit vector, as well as vector function call and multidimensional parallelism are also

introduced. The standard C operators act element-wise on vectors and some twenty new, expression operators have been added. It introduces a large number of new, vector operators that have no analogue in ordinary C and are not supported by the existing multimedia hardware extensions. Moreover, Vector C relies on a view of arrays as first-class objects, whereas the confusion of arrays with pointers is essential to the character of the C language. Vector conditional expressions in the Vector C language are handled with the bit vector. For example, the elements from the vector **a** are changed if corresponding bits in the bit vector are one, otherwise they remains unchanged. Multimedia hardware does not support this kind of operation which depends on the bit vector, thus in the MMC language we had to redefine the act of conditional assignment. Our method generates two vector strips that act as masks. The method was described in Section 3 and in Ref. 5. MMC also differs from Vector C in providing fewer special facilities for vector manipulation (for example, vector inner product is not a primitive operator in MMC although it could be easily expressed as we saw in previous examples) and in preserving the interchangeability of arrays and pointers.

The **C[]** language[36] is very similar to Vector C language. It targets general vector machines and, like Vector C, it introduces a large number of new vector operators that have no analogue in ordinary C and are not supported by the existing multimedia hardware extensions. But we found the syntax notation introduced in the C[] language the most suitable for MMC expressions of multimedia operations over packed data within a register (for example, we used the [] operator to describe most multimedia operations, rather then the @ operator used in Vector C).

The **C***[40] language is a commercial data-parallel language from Thinking Machine Corporation, which was compiled onto their SIMD CM-2 machine. The main difference between our work and C* is that C* targets large-scale SIMD machines while MMC targets the multimedia extension. C* targets large-scale data-parallel model, which assumes a system with a front-end processor (FE) that controls the overall system and many "processing elements" (PE's). C* extends C by having many processors instead of just one, all executing the same instruction stream. The C* execution model may be summarized as providing the programmer with lots of processors of an conventional nature, operating with a uniform address space in a synchronous execution mode. C* also adds to C additional overloaded meanings of existing operators and new library functions. These overloaded operators provide patterns of communication (i.e., fetching one value from a particular PEs memory, storing one value to the particular PEs memory, broadcasting a value to all PEs, communication among PEs,...). It also extends the declarations in such a way that we can

declare to which memory some variable should be stored. The authors of C* have added two new parallel operators (min, max). Both could easily be expressed in MMC through semantically extended C operators. C* also differs from MMC in adding a new type of statement to C, the selection statement, which is used to activate multiple processors. And finally, MMC tries to incorporate as much as possible of multimedia processing facilities and in addition to provide as few as possible new operators and type extensions to ANSI C.

6. CONCLUSION AND FUTURE WORK

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 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 scientific and multimedia applications have significant good performance improvements. In the future we should try to describe the difficulty or ease of using the presented language. In addition, more detailed performance-evaluation results should be provided and compared to the hand-optimized SIMD code.

Our MMC, its compiler and macro library are still in their infancy. Although successful, we believe their effectiveness can be further improved.

ACKNOWLEDGMENT

We would like to thank Prof. Ljubo Pipan from the Laboratory for Computer Architecture, Faculty of Computer and Information Science, University of Ljubljana for his helpful comments and discussions during the preparation of this paper. We would also like to thank the anonymous reviewers for suggesting improvements and their helpful comments.

REFERENCES

1. A. Moshovos and G. S. Sohi, Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling, *Proc. IEEE* **89**(11):1560–1575 (November 2001).
2. I. Kuroda and T. Nishitani, Multimedia Processors, *Proc. IEEE* **86**(6):1203–1221 (June 1998).
3. R. Lee, Accelerating Multimedia with Enhanced Processors, *IEEE Micro* **15**(2):22–32 (1995).
4. R. Lee and M. D. Smith, Media Processing: A New Design Target, *IEEE Micro* **16**(4):6–9 (1996).
5. M. Mitall, A. Peleg, and U. Weiser, MMX Technology Architecture Overview, *Intel Technology Journal* (1997).
6. Pentium (R) II Processor Application Notes, MMX (TM) Technology C Intrinsics. http://developer.intel.com/technology/collateral/pentiumii/907/907.htm.
7. Intel Architecture Software Developer's Manual Volume 1: Basic Architecture. http://download.intel.nl/design/pentiumii/manuals/24319002.pdf.
8. Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference. http://download.intel.nl/design/pentiumii/manuals/24319102.pdf.
9. Intel Architecture Software Developer's Manual Volume 3: System Programming. http://download.intel.nl/design/pentiumii/manuals/24319202.pdf.
10. V. Lappalainen, T. D. Hamalainen, and P. Liuha, Overview of Research Efforts on Media ISA Extensions and Their Usage in Video Coding, *IEEE Trans. Circuits Systems Video Tech.* **12**(8):660–670 (2002).
11. S. Oberman, G. Favor, and F. Weber, AMD 3DNow! Technology: Architecture and Implementation, *IEEE Micro* **19**(2):37–48 (1999).
12. A. Peleg and U. Weiser, MMX Technology Extension to the Intel Architecture, *IEEE Micro* **16**(4):42–50 (1996).
13. Intel C++ Compiler for Linux 6.0. http://www.intel.com/software/products/compilers/c60l/.
14. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *ACM Trans. Progr. Lang. Sys.* **9**(4):491–542 (1987).
15. D. F. Bacon, S. L. Graham, and O. J. Sharp, Compiler Transformations for High-Performance Computing, *ACM Comput. Surv.* **26**(4):345–420 (1994).
16. U. Banerjee, R. Eigenman, A. Nicolau, and D. A. Padua, Automatic Programm Parallelization, *Proc. IEEE* **81**(2):211–243 (1993).
17. A. J. C. Bik, M. Girkar, P. M. Grey, and X. M. Tian, Automatic Intra-Register Vectorization for the Intel® Architecture, *Int. J. Parallel Progr.* **30**(2):65–98 (2002).
18. P. Boulet, A. Darte, G. A. Silber, and V. Frederic, Loop Parallelization Algorithms: From Parallelism Extraction to Code Generation, *Parallel Comput.* **24**:421–444 (1998).
19. P. Bulić and V. Guštin, Macro Extension for SIMD Processing, *Proceedings of the 7th European Conference on Parallel Processing EURO-PAR 2001Manchester, UK, 28–31 August, 2001, Lecture Notes in Computer Science*, Vol. 2150, pp. 448–451 (2001).
20. P. Bulić, *The Compilation of High-Level Languages with Regard to the Instruction Set for Parallel Processing. Master Thesis*, University of Ljubljana, Faculty of Computer and Information Science (2001).
21. F. Corbera, R. Asenjo, and E. Zapata, New Shape Analysis and Interprocedural Techniques for Automatic Parallelization of C Codes, *Int. J. Parallel Progr.* **30**(1):37–63 (2002).
22. R. Gupta, S. Pande, K. Psarris, and V. Sarkar, Compilation Techniques for Parallel Systems, *Parallel Comput.* **25**:1741–1783 (1999).

23. M. Gupta, S. Mukhopadhyay, and N. Sinha, Automatic Parallelization of Recursive Procedures, *Int. J. Parallel Progr.* **28**(6):537–562 (2000).
24. V. Guštin and P. Bulić, Extracting SIMD Parallelism from "for" Loops, *Proceedings of the 2001 ICPP Workshop on HPSECA, ICPP Conference, Valencia, Spain, 3–7 September, 2001*, pp. 23–28 (2001).
25. A. Krall and S. Lelait, Compilation Techniques for Multimedia Processors, *Int. J. Parallel Progr.* **28**(4):347–361 (2000).
26. S. Larsen and S. Amarasinghe, Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *Processing of the SIGPLAN'00 Conference on programming Language Design Implementation, Vancouver, B.C., June 2000*. http://www.cog.lcs.mit.edu/slp/SLP-PLDI-2000.pdf.
27. K. Psarris, Program Analysis Techniques for Transforming Programs for Parallel Execution, *Parallel Comput.* **28**(3):455–469 (2002).
28. V. Sarkar, Optimized Unrolling of Nested Loops, *Int. J. Parallel Progr.* **29**(5):545–581 (2001).
29. N. Sreraman and R. Govindarajan, A Vectorizing Compiler for Multimedia Extensions, *Int. J. Parallel Progr.* **28**(4):363–400 (2000).
30. J. Y. Tsai, Z. Jiang, and P. C. Yew, Compiler Techniques for the Superthreaded Architectures, *Int. J. Parallel Progr.* **27**(1):1–19 (1999).
31. M. J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison–Wesley (1996).
32. P. Bulić and V. Guštin, Introducing the Vector C, to appear in *VECPAR2002: Seleced Papers and Invited Talks, Lecture Notes in Computer Science*.
33. P. Cockshot, *Vector Pascal*, Department of Computer Science, University of Glasgow (September 2001).
34. R. Fisher, Compiling for SIMD Within a Register, *Processings of Workshop on Languages and Compilers for Parallel Processing*, North Carolina (August 1998).
35. R. Fisher, Compiling for SIMD Within a Register, *Lecture Notes in Comput. Sci.* **1656**:290–304 (1999).
36. S. Gaissaryan and A. Lastovetsky, An ANSI C for Vector and Superscalar Computers and Its Retargetable Compiler, *J. C Lang. Transl.* **5**(3):183–198 (1994).
37. V. Guštin and P. Bulić, Introducing the Vector C, *Proceedings of the 5th International Meeting on High Performance Computing for Computational Science VECPAR 2002, Part I, Porto, Portugal, 26–28 June, 2002*. pp. 253–266 (2002).
38. K. C. Li and H. Schwetman, Vector C—A Vector Processing Language, *J. Parallel Distrib. Comput.* **2**:132–169 (1985).
39. K. C. Li, A Note on the Vector C Language, *ACM SIGPLAN Notices* **21**(1):49–57 (1986).
40. J. R. Rose and G. L. Steele, C*: An Extended C Language for Data Parallel Programming, *Proceedings of the Second International Conference on Supercomputing ICS87, May, 1987*, pp. 2–16 (1987).
41. J. R. Rose and G. L. Steele, The C[] Language Specification. http://www.ispras.ru/~cbr/cbrsp.html.
42. J. R. Rose and G. L. Steele, MMX Technology Aplication Notes: Using MMX Instructions to Convert RGB to YUV Color Conversion. http://cedar.intel.com.
43. I. Ahmad, Y. He, and M. L. Liou, Video Compression With Parallel Processing, *Parallel Comput.* **28**:1039–1078 (2002).
44. W. Amme and E. Zehender, Data Dependence Analysis in Programs with Pointers, *Parallel Comput.* **24**:505–525 (1998).
45. W. Amme, P. Braun, F. Thomasset, and E. Zehender, Data Dependence Analysis of Assembly Code, *Int. J. Parallel Progr.* **28**(5):431–467 (2000).

46. U. Banerjee, *Dependence Analysis*, Kluwer Academic Publishers, Dordrecht (1997).
47. F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang, Distributed pC++: Basic Ideas for an Object Parallel Language, *Sci. Progr.* **2**(3):7–22 (1993).
48. P. Y. Calland, A. Darte, Y. Robert, and F. Vivien, On the Removal of Anti- and Output-Dependences, *Int. J. Parallel Progr.* **26**(3):285–312 (1998).
49. P. Faraboschi, G. Desoli, and J. A. Fisher, The Latest Word in Digital and Media Processing, *IEEE Signal Proc. Mag.* **15**(2):59–85 (1998).
50. M. Ferretti and D. Rizzo, Multimedia Extensions and Sub-Word Parallelism in Image Processing: Preliminary Results, *Lecture Notes in Comput. Sci.* **1685**:977–986 (1999).
51. A. John and J. C. Brown, Compilation of Constraint Programs with Noncyclic and Cyclic Dependences to Procedural Parallel Programs, *Int. J. Parallel Progr.* **26**(1):65–119 (1998).
52. J. K. Lee and D. Gannon, Object Oriented Parallel Programming Experiments and Results, *Processing Supercomputing 91, IEEE Computer Society Press*, pp. 273–82 (1991).
53. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers (1997).
54. Z. Shen, Z. Li, and P. C. Yew, An Empirical Study of Fortran Programs for Parallelizing Compilers, *IEEE Trans. Parallel Distrib. Syst.* **3**(1):356–364 (1992).
55. F. L. Van Scoy, Developing Software for Parallel Computing Systems, *Comput. Phys. Commun.* **97**:36–44 (1996).
56. M. E. Wolf and M. Lam, A Loop Transformation Theory and an Algorithm to Maximize Parallelism, *IEEE Trans. Parallel Distrib. Syst.* **2**(4):452–470 (October 1991).
57. M. E. Wolf and M. Lam, DSP Guru: Finite Impulse Response FAQ. http://www.dspguru.com/info/faqs/firfaq.htm.
58. M. E. Wolf and M. Lam, ANSI C Yacc grammar. http://www.lysator.liu.se/c/ANSI-C-grammar-y.html (1995).
59. M. E. Wolf and M. Lam, Motorola AltiVec Technology Programming Manual, http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf, 1999.
60. M. E. Wolf and M. Lam, SUN VIS Instruction Set User's Manual, http://www.sun.com/processors/manuals/805-1394.pdf (2001).