An ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler

Sergey Gaissaryan, and Alexey Lastovetsky

Abstract

This article describes an ANSI C language superset for vector and superscalar computers and its retargetable compiler prototype. The superset, named C[], allows one to write portable efficient programs for SIMD (vector and superscalar) computer architectures. The article discusses the motivation of our approach, the vector superset of the C language, and the retargetable compiler system.

Introduction

The C language is commonly used by professional programmers because it allows one to develop highly efficient software portable within the class of UNIX systems. C reflects all main features of UNIX systems' architecture, which has an impact on the program efficiency.

As computer architectures have changed, it has become necessary to reflect the changes in compilers' internal languages, by adding the constructs to express new computing facilities, such as vector calculations. But if we want to use these new facilities explicitly in programs, they should be also added to the C language.

We created a C language superset with the same vector capabilities as vector computer assembly languages, by adding several new notions to ANSI C. The resulting extended C language, named C[], allows one to write portable efficient programs for SIMD (vector and superscalar) computer architectures.

This article is composed from three parts, describing the motivation of our approach, the vector superset of the C language, and the retargetable compiler system.

Motivation

The C language allows one to develop highly efficient software that is portable within the class of UNIX systems. This is because C reflects all the main features of the architecture of UNIX systems which have an effect on the program efficiency, namely: machine-oriented data types (short, char, unsigned, etc.); indirect addressing and address arithmetic (arrays, pointers and their relationships); and other machine level notions (++, --, += operators, bit-fields, etc.). UNIX architecture is reflected in C with such completeness that many individual features of each particular system can be expressed by compiler parameters [1]. On the other hand, the C language hides from programmers the peculiarities of each particular architecture that have no analogs in other computers of the class (for example, the peculiarities of register storage, details of stack implementation, and details of instruction sets, etc.). Finally, the C Standard [2] has been developed, along with high-quality portable C compilers retargetable to each particular system of the class [1].

Propagation of vector and superscalar architectures has caused the need in similar programming language for these architectures because the C language does not reflect parallel facilities of these architectures. As vector and superscalar architectures are an evolution of UNIX systems architecture, the language which plays the same part as C does for UNIX systems may be developed as a superset of the C language.

There have been many efforts to develop such C supersets [3- 5], but the supersets we know have the following disadvantages:

- the conceptual models of these supersets are not sufficiently developed (for example, the concept of vector value is absent);
- the conceptual models reflect some peculiar features of particular architectures having no analogs in other vector and superscalar architectures (for example, the notion of descriptor in Vector C language [3] is natural for Cyber 205 but not natural for supercomputers of Cray family because all their vector instructions are of register-to-register type; the notion of parallel objects in the C* language [4] is natural for the Connection Machine 2 but not natural for Cray, Cyber 205, and other shared memory supercomputers because it excludes explicit parallel processing of arrays);
- the supersets do not take into account requirements related to implementation of the compiler being portable and retargetable to particular architectures of considered class.

We considered the following requirements while developing the superset of C for vector and superscalar computers:

- the superset must adequately reflect all common features of the relevant architectures;
- the conceptual model of the superset must provide simple and efficient implementation for all computers of the class;
- the superset must be suitable for implementation of portable and retargetable compilers.

The C language might be extended for vector and superscalar computers by means of corresponding class libraries of C++ or even by means of a suitable library of C functions. To meet the first of requirements above these classes or functions must be implemented in assembly language. But if we do so the two last requirements will not be satisfied because of the necessity of reimplementation of assembly code. Hence, such a technique of extending C is unsuitable, if the class of target computers is wide enough.

Description of the C[] language

The C[] language is a strict superset of ANSI C. The following is a brief description of its main features.

Vectors

The basic new notion of the C[] language is a notion of *vector value* (or simply *vector*). A vector is defined as an ordered sequence of values of any type (the elements of the vector); the types of all the elements of a vector must be the same. It is important to emphasize that in contrast to array, a vector is not an object, it is a new sort of value. Unlike in C, in the C[] language the notion of the value of an array object is defined, and this value is a vector.

Example. The value of the array defined by the declaration

```
int a[3][2];
```

is the vector consisting of three vectors, each of which consists of two integers. So, the execution of the iteration statement

```
for(i=0; i<3; i++)
  for(j=0; j<2; j++)
    a[i][j]=i+j;
```

causes the value of the array a to be equal to the vector `{ {0,1}, {1,2}, {2,3} }`. The type of this vector is named by `int[3][2]`.

Arrays

In the C language an array comprises "a contiguously allocated set of elements of any one type of object".

In the C[] language an array comprises a sequentially allocated elements (with a constant positive, negative or zero 'step') of any one type of object. A negative step specifies the allocation of elements of the array in reverse order (from right to left). A zero step specifies the allocation of all elements of the array in the same element of storage.

Thus in the C[] language an array has at least three attributes, namely: the type of its elements, the number of elements and the allocation step.

In the C[] language, the array declarator syntax differs from the standard in following way. The rule

direct-declarator:
direct-declarator `[` *constant-expression*(opt)`]`

is replaced with the rules

direct-declarator:
direct-declarator `[` *constant-expression*(opt)
step(opt)`]`
step: `':'` *constant-expression*

If *step* is not specified then it is equal to 1.

Examples:

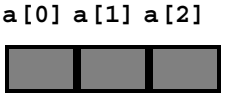
1. The declarations

```
int a[3:1];
```

and

```
int a[3];
```

both define an array of the form



The size of the slot between elements of the array is equal to zero.

2. The declaration

```
int a[3:3];
```

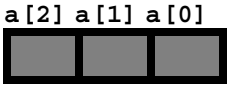
defines an array of the form

The size of the slot between array elements is equal to '2*sizeof(int) bytes.

3. The declaration

```
int a[3:-1];
```

defines an array of the form



Pointers

In the C language a pointer has only one attribute, namely the type of object it points to. This attribute is necessary for the correct interpretation of values of objects it points to as well as the address operators + and -. These operators are correct only if the pointer's operands and the pointer's results point to elements of the same array object.

The same rule is valid for the C[] language. Therefore, to support the correct interpretation of the address operators, we introduce one additional attribute of the pointer; this attribute is *step*.

In the C language Standard, "when an expression that has integral type is added to or subtracted from a pointer, the integral value is first multiplied by the size of the object pointed to". In the C[] language, the multiplier is equal to the product of the pointer step and the size of the object pointed to. In the C language, "when two pointers to elements of the same array object are subtracted, the difference is divided by the size of a element". In the C[] language, the divisor is equal to the product of the pointer step and the size of an element.

In the C[] language the pointer declarator is defined in following way:

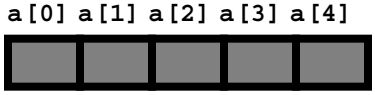
pointer:
* *step*(opt) *type-specifier-list*(opt)
* *step*(opt) *type-specifier-list*(opt) *pointer*

If *step* is not specified then it is equal to 1.

Example: The declaration

```
int a[]={0,1,2,3,4};
```

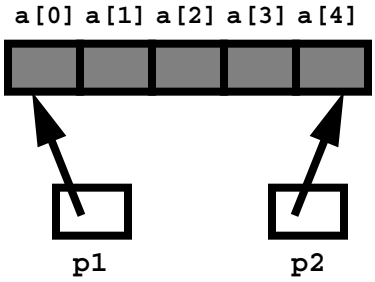
defines an array of the form



The pointer declarations

```
int *:2 p1=(void*)a, *:-1 p2=(void*)&a[4];
```

form the following structure of storage



The address expressions `(p1+1)` and `(p2+2)` point to the same element of the array, namely, `a[2]`. Indeed, the offset from `p1` defined by the expression `(p1+1)` is equal to

```
(2*sizeof(int))*1
```

bytes, and the offset from `p2` defined by the expression `(p2+2)` is equal to

```
((-1)*sizeof(int))*2
```

bytes.

Access to the Elements of an Array

In the C[] language, access to `e2`-th element of an array object `e1` is obtained with using one of the expressions `e1[e2]` or `(e2)[e1]`. Both are identical to `(*(e1+(e2)))`. Here, `e2` is an integral expression, `e1` is an lvalue that has the type "array of *type*". This lvalue is converted to an expression that has the type "pointer to *type*" and that points to the initial element of the array object (the attribute *`step`* of this pointer is identical to the attribute *`step`* of the array object).

Access to the Value of an Array

In the C[] language, the value of an array object is a vector. The `i`-th element of a vector is the value of the `i`-th element of the corresponding array object.

To support access to arrays as the whole, we introduced the postfix `[]` operator. The operand of this operator has the type "array of *type*". The `[]` operator blocks (forbids) the conversion of the operand to a pointer. Any lvalues that designate an array as the whole are called *blocked lvalues*. The rules for treatment of blocked lvalues are entirely similar to the rules for treatment of lvalues that have a scalar type.

A *modifiable blocked lvalue* is a blocked lvalue that does not have a type declared with the `const` type specifier.

Example: If the arrays `a`, `b` and `c` are declared as

```
int a[8], b[8:2], c[8:-1];
```

then the expression `a[]=b[]+c[]` assigns the sum of vectors that are the values of the arrays `b` and `c` to the array `a`. Here the expression `a[]+b[]` has type "vector of 8 `int`s".

Access to Subarrays

An object belongs to an array if it is an element of the array or it belongs to an element of the array. Any set of objects belonging to some array is called a subarray iff this set can be described as an array (using bounds and step attributes as defined above). In addition, any subarray can be referred to as an object belonging to its array.

In principle, the facilities introduced are sufficient to access subarrays. For example, if the array object 'a' is defined by the declaration

```
int a[5][5];
```

then the blocked lvalue

```
(*(int(*)[5:6])a)[]                    (1)
```

designates an array of five ints (with step 6) that contains the main diagonal of the matrix `a`, and the blocked lvalues

```
(*(int(*)[4:6])(a[0]+1))[]                    (2)
```
and
```
(*(int(*)[4:6])(&a[0][1]))[]                  (3)
```
designate an array of four ints (with step 6) that contains the diagonal of the matrix `a` which is placed above the main diagonal.

The more compact notation results if variables of type "pointer to array" are used. So, if the pointer objects `p1` and `p2` are defined by the declaration
```
int (*p1)[5:6]=(void*)a,
    (*p2)[4:6]=(void*)(a[0]+1);
```
then the expression `(*p1)[]` can be used instead of (1) and the expression `(*p2)[]` can be used instead of (2) and (3).

In the C language the array subscripting operator is redundant. This operator provides a comfortable abbreviation `e1[e2]` (or `(e2)[e1]`) for the expression `(*(e1+(e2)))` where the expression `e1` has type "pointer to *type*" and the expression `e2` has integral type. Similarly, we added the ternary `[:]` operator in the C[] language. This operator provides an abbreviation for the blocked lvalues shown in expressions (1)-(3) above. The expression `e1[e2:e3]`, where `e1` is a lvalue of type `T` and `e2` and `e3` are integral expressions, is equal to the expression
```
(*(T(*)[e2:e3])(&(e1)))[]
```
The step is equal to `e3*sizeof(T)`. The expression `e3` is optional (the expression `e1[e2:]` is equal to the expression `e1[e2:1]`).

In above example, the expression (1) can be written as `a[0][0][5:6]` and the expressions (2) and (3) as `a[0][1][4:6]`.

Example: If the array `b` is declared as
```
int b[4][4];
```
then the blocked lvalue designating the subarray represented in Fig.1 can be expressed as `b[2][0][2:][2:2]`.

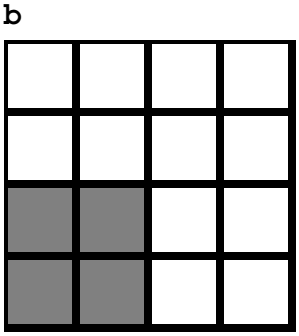


Figure 1: 2x2 subarray of array `b`

Segments of Arrays

Not every regular set of objects belonging to an array is a subarray. For example, the rectangular segment of the array `a` represented in Fig.2 is not a subarray. Therefore, an lvalue of type *array* does not exist for this segment. However, while using an array of pointers we can access its value.

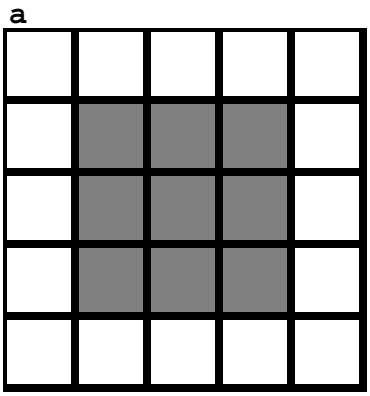


Figure 2: Rectangular 3x3 segment of array `a`

The value of this segment is the vector consisting of three vectors, each from which consists of three ints. If the array `p` is declared as

```
int (*p[3])[3]={a[1]+1, a[2]+1, a[3]+1};
```

then the expression `*p[]` provides the value of the required segment (here the `*` operator is applied to all elements of the vector `p[]`).

To ease access to similar sets of objects belonging to an array, we introduced new derived types *segments of arrays* and we extended the `[:]` operator.

A segment of an array may comprise sequential elements of the array allocated with a constant step (a unit of step size is an element of the array). Besides, if elements of an array are also arrays then a segment of the array may comprise a sequential segments (of the same type) of the array elements allocated with a constant step. In this last case, a unit of step size may be not only a segment of an element of the array but an element of a segment of an element of the array if an element of a segment of an element of the array is also an array and so on.

In one common case, in the expression `e1[e2:e3]` the third operand, which specifies the step, is described as follows:

segment-step:
integral-expression(opt)
':' *segment-step*(opt)

The expression `e1` is an lvalue whose type is not an array, or a blocked lvalue of type array, or an lvalue of type "segment of array". If `e3` is an integral expression, then the unit of step is equal to the size of the object designated by `e1`. If `e3` is `:e4` where `e4` is an integral expression then the unit of step is an element of the object designated by `e1`. If `e3` is `::e4` then a unit of step is an element of an element of the object designated by `e1`. (Naturally, in this case `e1` should be an lvalue of the corresponding type).

If `e3` is `:e4` then the expression `e1[e2:e3]` is an lvalue of type "segment of array" and designates the corresponding segment. If `e1` is an lvalue of type "segment of array" then the expression

`e1[e2:e3]` is also an lvalue of type "segment of array" and designates the corresponding segment. In all remaining cases the expression `e1[e2:e3]` is a blocked lvalue of type array.

An lvalue of type "segment of array" is never converted to a pointer that points to the initial element of the segment - it always designates the segment as the whole. For example, the lvalue of type "segment of array" that designates the segment represented in Fig.2 is `a[1][1][3:][3::2]`.

There are some constraints in the usage of lvalues that have type "segment of array". In particular, the `&` unary operator is not applicable to such lvalues. If `e1` is an lvalue of type "segment of array" and `e2` is an integral expression then the expression `e1[e2]` designates the `e2`-th element of the segment `e1` (counting from zero). In contrast to the array subscripting operator, the segment subscripting operator is not expressed by the `*` and `+` address operators; that is, in this case the expression `e1[e2]` is not equivalent to the expression `(*(e1+(e2)))`.

Lvector

In addition to notions of lvalue, blocked lvalue, and segment of array which are used for designation of objects and are allowed as left operand of assignment operators, we also introduced the notion of *lvector.*

Just as lvalue is an expression designating some object, an lvector is a vector expression designating a set of objects.

Example: If the objects `x`, `y` and `pa` are described as

```
int x, y, *pa[10];
```

then the vector expressions `{y, x, *pa[3]}` and `*pa[]` are lvectors, but the vector expression `{x+y, y, x}` is not.

Blocked lvalues of type array and segments of array can be considered as special cases of lvectors which designate the sets of objects allocated in the regular way.

An lvector is modifiable if every element of the set of objects it designates is modifiable.

Vector Conversions

The conversion of a vector of one type to a vector of another type is the composition of two conversions, the first of which converts the vector length, the types of elements remaining unchanged, and the second of which converts the type of vector elements, the vector length being unchanged.

We define a vector's *length* as the number of elements it contains. For example, the vectors `{1,2}` and `{{3,4,5},{6,7,8}}` both have the length 2.

There is no defined conversion of vector to non-vector (in particular, to scalar).

The conversion of a non-vector (in particular, of a scalar) to a vector of specified length and type of the elements is the composition of two conversions, the first of which converts the non-vector to a vector of specified length all elements of which have the same type and value as the initial non-vector, and the second of which converts the type of vector elements to the specified type.

Shortening a vector is performed by dropping its trailing elements. Lengthening a vector is performed by adding of elements having indefinite values to its tail.

Integer Vector Packing

The C[] language provides bit-fields packing facilities for integer vectors. Namely, an array declarator is allowed in position of optional *declarator* in the bit-field declarator of the

declarator(opt) : *constant-expression*

kind. The bit-field declarator `A[N:L]:M`, where `A` is an identifier, `N`, `L`, and `M` are constant expressions, specifies `N` sequentially allocated bit-fields, with the step `L`, each of width `M`. The width of the bit-fields and the step size are measured in bits. The assignment of an integer vector of length `N` to the member `A` of a structure causes the vector elements to be packed into corresponding bit-fields. Unpacking an integer vector packed in bit-fields is performed during the access to the corresponding member of the structure by means of the `.` operator.

Scalar Operators

We added two groups of scalar operators to the C[] language.

The first group consists of two binary `?>` and `?<` operators that calculate the maximum and minimum, together with their corresponding compound assignments.

The second group contains three unary bitwise operators, namely: `?` (counting one bits), `%` (counting leading zeroes), `@` (word reversing).

Vector Operators

The operand of unary `&, *, +, -, ~, ?, %, @,` and `!` operators and scalar cast operators may have a vector type. In this case the result of such an operator is a vector of elements which results from applying the corresponding operator to the elements of its operand.

If a vector type name is used in a cast operator the vector length specified by the operator may be specified by an arbitrary (not only constant) integral expression.

One or both operands of binary `*, /, %, ?<, ?>, +, -, <<, >>, <, >, <=, >=, ==, !=, &, ^, |, &&,` and `||` operators may have vector type.

If both operands are vectors of the same length then the result is a vector the elements of which are the results of application of corresponding operator to the elements of the operands. If one of the operands is scalar then it is converted to a vector of the same length as the vector operand.

If vector operands of a binary operator have different lengths then the behavior is undefined.

If the first operand of conditional operator is a scalar and the second or third operand or both are of vector type then the result of the operator has the same vector type as for binary operators discussed above. The first operand of a conditional operator may have vector type. In that case the second or the third operand but not both of them may be omitted. If none of the operands is omitted then unlike the C language all three operands are evaluated. If all three operands are vectors of the same length then the result is produced by elementwise application of the operator. If vector operands of a conditional operator have different lengths then behavior is undefined. If the second or the third operand is non-vector then the length of that operand is converted to the length of the vector operands. If the first and the second (or the third) operands are vectors, the third (the second) operand is omitted, and the elements of the first operand have scalar type, then the result will be the vector of the same type as the second (the third) operand; the `i`-th element of the result is equal to the `k(i)`-th element of the second (the third) operand where `k(i)` is the index of the `i`-th non-zero (zero) element of the first operand. The other elements have indefinite values. For example, execution of

```
int a[5], b[5], c[5];
a[]={1,2,3,4,5};
b[]={3,3,3,2,6};
c[]=a[]<b[]?a[]:; /*the 3rd operand is omitted */
```

results in the vector `c[]` equal to {1,2,5,*w*,*w*}, where *w* denotes an undefined value.

If the first and the second (or the third) operands are vectors, the third (the second) operand is omitted, and the elements of the first operand have vector type, then the result is achieved by elementwise application of the operator.

An assignment operator may have as its left operand a modifiable blocked lvalue, a segment of modifiable array, or any other modifiable lvector. In that case its right operand may have vector type. In any case the type of its right operand converts to the type of the left operand's value. For example, the execution of following fragment

```
int a[]={0,1,2,3,4};
int *pa[]={a+1,a+2,a+3,a+4,a};
*(pa[])=a[];
```

results the vector {1,2,3,4,0} as value of array `a`.

The unary linear `[*]`, `[/]`, `[%]`, `[?<]`, `[?>]`, `[+]`, `[-]`, `[&]`, `[^]`, and `[|]` operators correspond to binary `*`, `/`, `%`, `?<`, `?>`, `+`, `-`, `&`, `^`, and `|` operators. These operators are applicable only to vector operands. Let `v[0]`, `v[1]`,...,`v[N]` denote the elements of vector operand `v`. Then the expression `[op] v[]` has the same semantics as the expression of

```
(...((v[0] op v[1]) op v[2]) op ... op v[N])
```

kind. For example, the execution of following fragment

```
int a[]={0,1,2,3,4}, sum;
sum=[+]a[];
```

produces the value of `sum` equal to the value of the expression

```
a[0]+a[1]+a[2]+a[3]+a[4]
```

which is equal to `10`.

Example: Here is a program performing some matrix calculations.

```
double a[10][10], b[10][10],
       axb[10][10], bxa[10][10],
       diag_axb[10];
void main()
{
   int i, j;
   for(i=0; i<10; i++)
     for(j=0; j<10; j++)
       axb[i][j]=[+](a[i][]*(*(float(*)[10:10])&b[0][j])[]);
   for(i=0; i<10; i++)
     for(j=0; j<10; j++)
       bxa[i][j]=[+](b[i][]*(*(float(*)[10:10])&a[0][j])[]);
   diag_axb[]=([&][&](axb[]==bxa[]))?
       (*((float(*)[10:11])axb))[]:        /*Storing the diagonal*/
       (*((float(*)[10:-11])&axb[9][9]))[];/*Storing the diagonal
                                             in reverse order*/
}
```

In the C[] language, unlike in the C language, a formal parameter of a function may have an array type. The declaration of such a formal parameter must explicitly specify all array attributes including the step specification. For example, `int a[10:1]` in the list of formal parameters specifies the array `a` of `10 int`s with step `1`. (Note, that `int a[10]` specifies the usual pointer to `int`). The corresponding argument is an expression of the same vector type that is defined by the formal parameter. The function value also may have vector type.

Arrays and Structures of `register` Storage Class

An array whose elements are of scalar type and whose step is equal to 1 may be declared with the storage-class specifier `register`. This causes the corresponding array priority allocation in a vector register if one is free. If no vector register is free or if an array of the declared type should not be allocated in a register, the storage-class specifier is ignored.

There are some constraints in usage of register variables of array type. In particular, the address `&` operator and the `[:]` operator should not be applied to it. The array subscripting operator may not expressed by the `*` and `+` address operators for a register array.

A structure with members of scalar types may be declared with storage-class specifier `register`. This causes the corresponding structure allocation in available scalar registers which allows collective exchange with main storage.

The Compiler

In April 1993 we started the development of portable and retargetable compiler system for the C[] language. To implement our compiler we use the Karlsruhe toolbox for compiler construction presented by GMD (Germany). Currently, we have implemented the version 1.0 of the compiler.

The compiler consists of four stages (components). The first stage analyses the source program file and builds its internal representation including the table of types, table of names, and attributed tree of the source program file.

The second stage translates the internal representation into an intermediate language. This language is an extension of the RTL language used in GCC. Extended RTL is a low-level intermediate language for vector and superscalar computers.

In the third stage, the intermediate program is tuned to the target computer. For example, when tuning intermediate code to Cray Systems, instructions that includes vector expressions with a length more than 64 are transformed to equivalent sequence of instructions where vector lengths do not exceed 64.

The fourth stage is a retargetable code generator. Currently, it generates code for Russian Cray-like supercomputer Elektronika SSBIS as well as ANSI C code.

Our current activities concern setting up the compiler on other vector and superscalar computers, as well as the development of some optimizing stages of the compiler.

Acknowledgments

We are grateful to Victor P. Ivannikov for his support of our work and especially to Bob Jervice for his helpful comments.

The work is supported by Russian Basic Research Foundation.

References

1. Stallman R. Using and Porting GNU CC. Free Software Foundation, 1988.

2. First working draft on programming language C (draft proposed by ANSI). Ottawa, ISO/TC97/SC22/WG14, 1986.

3. Kuo-Cheng Li, Schwetman H. Vector C: A Vector Processing Language. Journal of Parallel and Vector Computing, 1985, 2, No 2, pp. 132-169.

4. Connection Machine Model CM-2 Technical Summary (Version 6.0). Thinking Machines Corporation, Cambridge, Massachusetts, Nov. 1990.

5. Gisselquist R. An Experimental C Compiler for the Cray-2 Computer. ACM SIGPLAN Notices, v 21, No 9, 1986, pp 32-49.