1984

Vector C—A Vector Processing Language

Kuo-Cheng Li

Herb Schwetman

Report Number:
84-478

Vector C - A Vector Processing Language†

Kuo-Cheng Li and Herb Schwetman

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR-478

ABSTRACT

The language Vector C is a superset of the conventional (scalar) programming language C with extensions to facilitate vector processing. In this paper, a methodology for performing language extensions and the language features of Vector C are presented.

The implementation of Vector C on the Cyber 205 is nearing completion. Some empirical data are presented which demonstrate that Vector C can generate code which executes at speeds which meet or exceed those for equivalent statements from 205 Vector Fortran. Readers are assumed to have some knowledge of the C language.

KEYWORDS Cyber 205, Vector processing, Language design, Vector C

May 11, 1984

† This work was supported in part by the Purdue University Computing Center (PUCC).

Vector C - A Vector Processing Language†

Kuo-Cheng Li and Herb Schwetman

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR-478

1. Introduction

The CDC Cyber 205 is a vector supercomputer system. While the scalar speeds of the 205 represent a significant improvement over existing computer systems, the full potential of the 205 is realized only when the vector processing capabilities are utilized. Utilizing these capabilities may require new kinds of algorithms, and these algorithms will require a more powerful high level programming language that makes use of the capabilities of the underlying vector processing hardware.

There are several ways to design languages for vector computers, including 1) adding an automatic vectorizer to an existing language, 2) adding vector features to an existing language, and 3) developing a new vector-oriented language. For example, the FORTRAN compiler for the 205 (called FORTRAN 200) [CDC83] has been modified (extended) to allow access to the vector features of the system. Also, an automatic vectorizer in the compiler attempts to extract inherent parallelism in a sequential program and generate vector instructions whenever possible, even when they were not explicitly invoked by the programmer.

In [LiSc83], we listed some reasons for developing extensions to an existing programming language, C [KeRi78], and described the project to move conventional C to the 205. A major goal of this project is to preserve upward compatibility (i.e., the Vector C compiler accepts any legal scalar C programs). It is our conjecture that methods for accessing the vector

† This work was supported in part by the Purdue University Computing Center (PUCC).

processing capabilities of the 205 can be improved, to the benefit of the programmer, by adding vector features to the C language.

C is the major language of the UNIX‡ operating system. It is used both for writing "systems programs" (about 95% of the operating system is written in C) as well as many applications programs. C has a fairly terse syntax, with many features associated with modern, structured programming techniques. Part of the "tradition of C" is that it has statements which correspond to features of the underlying hardware (originally, the PDP 11). Thus, it is not a violation of the "spirit of C" to consider extensions which allow access to features of the host computer.

Vector C (VC), a superset of conventional C (CC), has been designed and is being implemented on the Cyber 205. In the following sections, the design goals, methodologies, and new vector features are presented along with extensive examples. Finally, the implementation of VC is briefly described, and some Vector C programs and related empirical timing data are shown. Readers are assumed to have some knowledge of the C language.

2. Design Goals

The design of VC has several goals; among these are:

a. The syntax of Vector C should permit easy, natural expression of vector algorithms, in a *direct* manner. We are not interested in an automatic vectorizer which can indirectly deduce situations which can exploit vector instructions.

b. Vector C should be consistent with the base language (Scalar C), i.e. it should be familiar to C programmers. While it should not be unpleasant to any programmers (including FORTRAN programmers), it should do things in the "C way" first.

---

‡ UNIX is a Trademark of Bell Laboratories.

c. Vector C should be functionally complete in the sense that all algorithms for the 205 can be expressed in it, and it should allow experienced programmers to write efficient programs which use the vector hardware of the 205.

d. Vector C should be small and implementable in an efficient manner. Because the 205 instruction set is so rich (complicated, baroque, etc.), it is impossible (and probably unnecessary) to have operators in C which correspond to every 205 instruction. The use of functions (which are possibly compiled as in-line code) may be required, to accommodate all of the instructions available.

e. Performance of the generated code is important. The major reason for using the 205 (as opposed to a VAX, for example) is to speed up execution of a program. We must continue to search for ways to further optimize the generated code with respect to execution time.

e. Provision for instrumentation (tools) for determining the performance of a running program is important. This will be essential to anyone trying to conduct research into vector algorithms and their implementation on the 205.

3. Methodology

The methodology which we used to guide the design of Vector C was based primarily on identifying those language constructs which were thought to be necessary for describing vector algorithms. A major source of information in determining these constructs was a study of existing works in this area.

The other factors which were part of the methodology are the existing base language (C) and the architecture of the target machine (the 205). These considerations led to the selection of a set of language constructs. We then organized the constructs into a hierarchy and then implemented them as extensions to C.

3.1. Hierarchy of Language Constructs

Language constructs for a specific vector machine can be classified into four levels:

Level 0 - General constructs:

The constructs in this level are general (and portable) to all machines (i.e., sequential processors, vector/array processors (SIMD) and multi-processors (MIMD) [Flyn72]); e.g., the fundamental arithmetic operations such as addition, subtraction, multiplication, and division, etc., the basic data types such as integer, floating point number, character, the logical operations such as 'and', 'or', 'xor', etc.,

Level 1 - Macro constructs:

This level of constructs include transformational operations such as dot-product, summation, product, minimum, and maximum, and some elemental operations such as absolute, floor, ceiling, etc. These operations are portable but may be expensive to implement on some machines which do not have the corresponding hardware instructions.

Level 2 - Vector-oriented constructs:

The constructs at this level include special vector-oriented constructs such as gather/scatter and compress/expand instructions, and vector data types. (They may be only appropriate to vector processors).

Level 3 - Machine dependent constructs:

In order to fully utilize the underlying machine capabilities, some constructs which reflect idiosyncrasies of the hardware may be necessary; hence this level of constructs is usually machine dependent. This means that transporting these language constructs onto another machine may be inefficient; at this level we find constructs such as the bit data type and conditional vector instructions of the 205.

The portability of constructs at each level decreases as the level number increases. In Vector C, we have constructs at all levels. If portability becomes an important issue, a preprocessor can be built; this preprocessor can convert constructs in the extended language to an "equivalent" sequence of statements or function calls in the base language.

3.2. Extensions

As we mentioned before, the extensive processing power of the Cyber 205 can be realized only when we can find ways to utilize its vector processor; Vector C is a means of achieving this goal. The extensions to C include:

a. *Language constructs:* Conventional C is extended to include 1) a means for specifying vectors and 2) operators and data/control structures for manipulating vectors. In other words, the vector extensions include vector data types, vector expressions, and vector operators and/or keywords.

b. *Built-in functions:* Because of the design goals and the constraints imposed by the 205 including 1) the goal that VC should be kept small, 2) the 205 instruction set is rich and 3) portability is an issue (a tradition of C), built-in functions are imperative. Built-in functions are implemented using either the appropriate hardware instructions or the extended language.

4. Extended Language Features

Once the framework of the language constructs had been determined, the actual language features were designed; each feature had to serve a recognized need and remain within the stated constructs. The large number of features are presented in a "bottom up" approach, beginning with data types and storage classes, and continuing with several forms of vector expressions and vector operators. Most of the descriptions of these features are accompanied by illustrative examples.

4.1. Data types and storage classes

The data types and storage classes of C have been augmented in Vector C, so that the new vector processing capabilities can be exploited; these include:

(a) VECTOR data types:

On the Cyber 205, a *vector* is defined to be an ordered sequence of elements stored in consecutive memory locations (i.e., this is denoted as vector with stride 1); a vector is represented by a base address, an offset and a length; due to a hardware limitation (the length field is only 16 bits), the length of a vector is limited to 65535 elements. Noncontiguous array elements can be reformed into a consecutive 'vector' by using gather/scatter and compress/expand instructions [CDC81a] (see Section 4.2.5).

There are two alternatives to declaring vectors:

1. Implicit declaration:

vectors are declared implicitly by using the conventional C array declarations, e.g.

```
float va[100];
int  vb[10][20];
char vc[50];
```

where, 'va' is a floating point vector of length 100, i.e., va[0], va[1], ..., va[99] (in C, array subscripts begin at 0). 'vb' is an integer matrix of size 10 x 20, stored in row-wise manner (the last subscript varies fastest). 'vc' is a character vector of length 50.

2. Explicit declaration:

In this approach, a vector is explicitly declared using the vector (a new keyword) storage class; e.g.,

```
vector float v[100], m[10][10];
float      a[100], b[10][10];
```

where, v and m are vector arrays (i.e, arrays which are supposed to be processed by the vector processor), and a and b are sequential arrays (i.e., arrays which are supposed to be processed by the scalar processor). For another kind of SIMD machines, namely, array processors (e.g., ILLIAC IV [BDMR72], ICL DAP [Redd73], Burroughs BSP [Aust79], etc.), this explicit declaration is necessary, so that the vector elements can be positioned properly in memory.

The first approach may cause an ambiguity problem between a pointer and a vector if we adopt *implicit vector referencing* (see Section 4.2.2), i.e. if we allow the use of V to stand for the whole vector. The second approach can solve this problem, but it introduces a *semantic difficulty*, namely the statement

```
array1  = array2 + C
```

is legal only if array1 and array2 are both declared as vectors.

Of the two alternatives, we decided to adopt the *implicit* approach; further analysis of the pros and the cons of these two approaches is in [Li83]. Recall that the use of explicit declaration is introduced primarily to solve the ambiguity caused by an unsubscripted array name: is it an implicit vector reference or is it a pointer (the address of an array)? This ambiguity can be eliminated if we do not allow implicit vector referencing, i.e., vector references must be explicitly specified (e.g., V[*], V[1:10], see Section 4.2.1 and 4.2.2); an implicit reference (e.g., V) is treated as the address of the array (as in conventional C).

We introduce an (explicit) syntax for specifying that a data object is to be referenced as a vector. This new syntax (see Section 4.2.1) is used in the subscript part of a vector reference. As examples, given the declaration

```
float V[100], M[10][10];
```

the following constructs can be used to refer to V and M:

```
V       -- the address of V[0], i.e., &V[0],
V[0]    -- the first element of V,
```

V[*] -- the entire vector V,
M -- the address of M[0][0],
M[2] -- the address of M[2][0], (not the third row of M),
M[2][*] -- the 3rd row of the matrix M (a vector).

Thus *an object is recognized as a "vector" by its appearance (reference), not by its declaration*. Another advantage of the former approach (implicit declaration and explicit referencing) is that it is easier to read VC programs than to read the programs defined by the second approach (explicit declaration and implicit referencing), since, in this approach, the vector designation is obvious.

(b) DESCRIPTOR declaration:

A vector descriptor is a pointer to a vector; it contains the address of the vector and the length of that vector. Since a vector on the Cyber 205 is defined as stride-1, a descriptor composed of these two items is sufficient. Figure 1 is the 205 descriptor representation; bits 0-15 form the length field and bits 16-63 form the virtual address field.

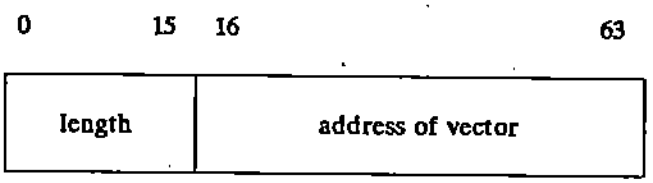


Figure 1: 205 descriptor representation

For the sake of generality and portability, a more general *array* descriptor [Li83] may be preferred; it could, for example, contain additional information, such as the number of dimensions (rank), the upper and lower bounds and the stride of each dimension (shape). In our current design, the simpler *vector* descriptor is used. A descriptor is specified by using a special character @, (similar to '*' for pointers in C), and each descriptor has a specified type associated with it; in the example

```
float @id;
int @vd[3][2];
```

'id' is a descriptor which represents a floating point array, and 'vd' is an array of descriptors which represent integer arrays.

(c) BIT data type:

Since the Cyber 205 is a bit-addressable machine and bit vectors are used in many instructions, the bit data type is introduced to allow access to this hardware capability; in the example

```
bit bv[100];
```

'bv' is a bit vector of length 100.

This data type is a level-3 construct; it will probably be inefficient to implement this on machines without bit-addressing capabilities.

(d) FORTRAN declaration:

In order to utilize the software parts which have already been provided for Fortran programs, the FORTRAN declaration is introduced to inform the compiler that Fortran linkages (rather than C linkages) should be generated; in the example

```
fortran FORTFUNC();
```

FORTFUNC is a Fortran function or subroutine.

(e) COLWISE storage class:

This storage class is provided so that a user can control the arrangement of a multi-dimensional array; in the example

```
colwise float m[10][10];
```

'm' is a matrix of floating point numbers, stored in the order m[0][0], m[1][0], ... It is the user's responsibility to specify proper and efficient array referencing. Improper referencing (e.g., accessing the columns in a matrix which are stored in rowwise manner) can incur a great deal of overhead. In addition to performance considerations, this construct is also useful for declaring matrices which will be passed to Fortran routines (since most matrices in Fortran are stored in a column-wise manner).

4.2. Vector expressions

A vector expression is an expression which contains at least one vector reference. The following sections describe the subscript expressions, vector referencing, vector initialization, bit expressions, vector arithmetic expressions, and vector relational expressions.

4.2.1. Subscript expressions

We have introduced seven different ways of specifying subscript ranges for referencing vectors:

(1) *initial* : *final* [: *increment*]

where, *initial*, *final* $\in$ {*zero* $\cup$ *positive integers*} and *increment* $\in$ *positive integers*. The pair of symbols [ ] stands for optional; the default *increment* is 1. Furthermore, *initial*, *final* and *increment* can be scalar constants, scalar variables, or scalar expressions. e.g., 0:4 specifies 0,1,2,3,4, and 1:10:2 specifies 1,3,5,7,9.

(2) *initial* # *length* [: *increment*]

where, *initial*, *length* $\in$ {*zero* $\cup$ *positive integers*}, *increment* $\in$ *positive integers*, and the default *increment* is 1, e.g., 0#4 specifies 0,1,2,3, and 2#5:2 specifies 2,4,6,8,10.

(3) *

whole-dimension selection; this is the same as *lower* : *upper* : 1, where *lower* and *upper* are the lower bound and upper bound of the corresponding dimension respectively. In fact, in C, the lower bound of any array dimension is always zero.

(4) *initial* : * [: *increment*]

here, asterisk is the upper bound of the corresponding dimension, e.g., 3:*:4 is the same as 3:*upper*:4.

(5) *index vector*

an index vector (containing natural numbers) can be used as an indirect indexing mechanism, e.g., V[IV[*]],

(6) *bit vector*

a control (bit) vector, composed of 0's and 1's, can also be used as a subscript mechanism, e.g., V[BV[*]],

(7) *descriptor*

an index vector or bit vector descriptor can also be a subscript, e.g., V[vd], where vd is a descriptor.

4.2.2. Vector referencing

Vector references could be represented in the following two ways:

1. *implicit referencing:* an array name without subscripts (or incomplete indexing for a multi-dimension array), e.g.,

   float V[100], M[20][20];

V	-- a reference to the entire vector,
M[2]	-- the third row of M,
M[0:10]	-- the upper half of the matrix M.

2. *explicit referencing:* an array name with vector subscript expressions: e.g.

   float V[100], M[20][20];

V[i]	-- scalar reference
V[i:f]	-- a (sub)vector consisting of elements V[i] through V[f]
V[i#len:2]	-- specifies V[i], V[i+2], ..., and V[i+2*(len-1)]
M[*][i]	-- the i-th column of the matrix M
M[*][*]	-- the entire matrix M
M[1:N-2][1:N-2]	-- the interior of the matrix M[N][N]
M[2][0#100]	-- a vector with 100 items start at the address of M[2][0]

As mentioned earlier, there is an ambiguity introduced by allowing unsubscripted array names or incomplete indices for multi-dimensional arrays (e.g., V or M[2]); to eliminate this ambiguity, we chose to adopt the second approach (explicit referencing).

Another issue to be addressed is conformity (handling the lengths of vectors in statements). Non-conformable vectors have unequal lengths, and conformable vectors require (at least) the same lengths. In this version of VC, we adopt *weak conformity* (since it is more flexible); in more detail, we allow

a. the number of vector subscripts to be different in a vector statement, e.g., m[*][*] = v[*], where the left-hand side has two vector subscripts and the right-hand side has only one, and

b. the vector lengths in a vector statement to be different, e.g., a[0:10:2] = b[2:5], where the left-hand side specifies six elements and the right-hand side specifies four elements; the initial indices can be different as well (e.g., 0 vs. 2). Both Actus [PeCM83] and Fortran 200 [CDC83] enforce strong (restricted) conformity (e.g., the initial indices must be same).

In Vector C on the 205, nonconformity in a vector statement is resolved by the 205 hardware, and an optional warning message is generated by the compiler. When the lengths of two operands are different, the shorter one will be appended (by the hardware) with proper dummy data (depending on the instruction). For example, if the result operand (l-value) has length less than that of the source operands (r-values), the vector instruction terminates when the result vector field is filled, and in the reverse situation, the exhausted fields of the source operands will be extended with zero's or one's [CDC81a] depending on the instruction. As examples,

1) a[0:10] = b[0:5] + 2 implies that a[0:5] = b[0:5] + 2 and a[6:10] = 0 + 2,

2) a[0:10] = b[0:5] * 3 implies that a[0:5] = b[0:5] * 3 and a[6:10] = 1 * 3, and

3) a[0:5] = b[0:10] + 2 is evaluated as a[0:5] = b[0:5] + 2.

Many vector languages, such as CFD [Stev75], Actus and Fortran 200, etc., restrict parallelism to only one dimension. Vector C allows multi-dimensional parallelism (i.e., can have more than one vector subscripts in an array reference). On a vector machine, one dimensional

vectors are sufficient; e.g., M[100][100] can be treated as a vector of length 10000, with a starting address of M[0][0], because in the layout of physical memory, a 100 by 100 matrix is the same as a vector of 10000 elements. However, a general programming language should provide programmers with the capability of constructing matrix (or even higher dimensional structure) references and letting the compiler decompose them into vectors, e.g., M[1:98][1:98] specifies the interior of the matrix M[100][100].

A remaining language design issue is the one of indirect indexing (i.e., using a vector as the subscript of another array). Indirect indexing in a matrix can have two interpretations [HoJe81]:

1. The projection interpretation: This causes a reduction in rank; many languages have adhered to this interpretation; these include CFD, DAP Fortran[ICL79] and Actus[PeCM79], etc.
2. The general mapping interpretation: BSP Fortran[BSP77] and VECTRAN[PaWi75] have adopted this interpretation. With general mapping, the rank is not reduced but the shape may be changed.

We adopt the general mapping interpretation. The general mapping interpretation seems to be more consistent, because, using this, rank reduction occurs only when there is a scalar index. As an example, in X[0#n][2][*], the rank is reduced (from three) to two. If we had adopted the former interpretation, conflicts would have been created (see example below). Also, the projection interpretation requires restricted conformity. In the example below, if array IV is {2,2,0,1}, with length four, this does not match the length of the second dimension of array A (which is three); an error is detected because of nonconformity. However, this would not be an error using the general mapping interpretation.

Example:

```
int A[4][3];
int IV[3] = {2, 2, 0};
```

where A is

```
0 1 2
3 4 5
6 7 8
9 a b
```

then, by projection, A[IV[*]][*] is the vector

```
6 7 2
```

By general mapping, A[IV[*]][*] is the matrix, with changed shape,

```
6 7 8
6 7 8
0 1 2
```

4.2.3. Vector initialization

Initial values for vectors can be specified at compile time using a syntax similar to that used in subscript expressions, with the addition of a dyadic repeat operator, '#'. As in CC, only global (external/static) vectors of any type can be initialized in this manner. The formal syntax for vector initialization is given in [Li83]. As an example,

```
int a[15] = {1,2,3, 1:5, 2:10:3, 3#4}
```

is the same as

```
int a[15] = {1,2,3,1,1,1,1,1,2,5,8,3,4,5,6}
```

4.2.4. Bit expressions

A bit expression can be either a vector relational expression, a bit vector, a bit descriptor, a bit vector function reference, a bit (string) constant, or bit variable. No arithmetic operations are allowed on bit data types. The logical operators (e.g., &&, ||) in C are interpreted the same as the bitwise logical operators (e.g., &, |) when their operands are bit expressions.

Examples:

```
bit bv[] = b"11111000011";
```

```
bit bm[3][20] = { {1!10,015},b"111001", {0!10,115} }; /* dyadic ! is the repeat operator */
main()
{
        int     a[100], b[100];
        bit     z[100], x[100], y[200];
        bit     *p;

        z[*] = x[*] & y[0#100:2];    /* bit periodic gather */
        z[*] = a[*] && b[*];         /* same as z[*] = (a[*] != 0) & (b[*] != 0) */
        z[*] = ! a[*];               /* same as z[*] = a[*] != 0; */
        p = b"1101011100111";
        z[*] = p[0:12];
        z[*] = 3 @0 5 # 10;          /* bit vector 0001100011, see Section 4.3.4 */
}
```

4.2.5. Vector arithmetic expressions

A vector arithmetic expression is a vector expression which yields a numeric vector. Vector arithmetic expressions are illustrated by several examples. In order to clarity the meaning of these expression, each example is followed by a set of equivalent scalar C statements.

Mixed data types are allowed in expressions wherever appropriate. The data type conversion rules and operator precedence rules for scalar C statements apply to vector statements.

Examples:

```
        int     va[100], vb[100], vc[10], vd[100];
        int     ma[10][10], mb[10][10];
        int     vf[20];
        bit     bv[100];
```

1. consecutive range - stride-1

```
va[0#n] = vb[3#n] + c;  /* scalar variable (or constant) c is broadcast */
```

is equivalent to:

```
for (i = 0; i < n; i++)
   va[i] = vb[3+i] + c;
```

2. skipped range at the rhs - *periodic gather*,
   (if the skipped range is at the lhs, it invokes a *periodic scatter*)

```
vc[*] = vd[1:2:2] * vf[2#10:3];
```

is equivalent to:

```
for (i = 0, j = 1, k = 2; i < 10; i++, j += 2, k +=3)
    vc[i] = vd[j] * vf[k];
```

3. indirect indexing with an index vector - *gather/scatter*

```
vf[vb[0#10]] = vc[2#10];  /* scatter */
```

is equivalent to:

```
for (i = 0; i < 10; i++)
   vf[vb[i]] = vc[i+2];
```

4. bit vector indexing - *compress/expand*

```
va[*] = vb[bv[*]];      /* compress */
```

is equivalent to:

```
/* simulated scalar code - since CC does not have bit data type */
for (i = j = 0; i < 100; i++)
   if (bv[i] == 1) va[j++] = vb[i];
```

5. element-wise matrix multiplication

```
ma[*][*] *= mb[*][*]; /* or, ma[0][0#100] *= mb[0][0#100]; */
```

is equivalent to:

```
for (i = 0; i < 10; i++)
   for (j = 0; j < 10; j++)
      ma[i][j] *= mb[i][j];
```

4.2.6. Vector relational expressions

A vector relational expression is a vector expression which always evaluates to a bit (control) vector of zero's and one's. In a conditional vector assignment statement, a value is stored into the result vector depending on the corresponding bit in the control vector. Also, if there are nested vector control structures, the control vector is stacked when entering a new control structure, and unstacked when exited. The control structures involving vector relational expressions include:

a. CONTROL expression: The control-expression has the form as shown at the right-hand side of the statement below:

<vector_expr> = <control_vector_expr> ? <expr> ;

The <control_vector_expr> is evaluated to a bit control vector. The elements of <expr> are assigned to <vector_expr> when the corresponding control bits are 1; those elements in <vector_expr> which have the corresponding control bits 0 remain unchanged.

E.g.,

```
int a[5], b[5];
bit z[5];

a[*] = z[*] ? b[*];
```

if z[*] = 1,0,0,1,1, a[*] = 1,2,3,4,5, and b[*] = 6,7,8,9,0, then a[*] becomes 6,2,3,9,0.

b. MASK expression: The mask-expression has the form as shown at the right-hand side of the statement below:

<vector_expr> = <control_vector_expr> ? <expr1> : <expr2> ;

The element in <vector_expr> is assigned from <expr1> if the corresponding control bit is 1, otherwise the corresponding element in <expr2> is assigned to <vector_expr>.

E.g.,

```
a[*] = z[*] ? b[*] : c[*];
```

if z[*] = 1,0,0,1,1, b[*] = 6,7,8,9,0, and c[*] = 5,4,3,2,1, then a[*] becomes 6,4,3,9,0.

c. MERGE expression: The merge-expression has the form as shown at the right-hand side of the statement below:

<vector_expr> = <control_vector_expr> ? <expr1> <> <expr2> ;

This operation has an effect similar to the MASK operation, except that <expr1> and <expr2> are not skipped.

E.g.,

```
a[*] = z[*] ? b[*] <> c[*];
```

if z[*] = 1,0,0,1,1, b[*] = 6,7,8,9,0, and c[*] = 5,4,3,2,1, then a[*] becomes 6,5,4,7,8.

d. IF statement: The if-statement has the form:

```
if ( <if_expr> )  <if_block>;
```

or,

```
if ( <if_expr> )  <if_block>; else <else_block> ;
```

The <if_expr> is evaluated as an integer value (scalar) or a bit control vector. If it is the former (a scalar), the normal if-statement is executed. If it is the latter (a vector), this control vector is applied to all operations in the <if_block> but not to the arguments of function calls. This is because if the control vector is intended to apply to the arguments, the CONTROL expression is a way of accomplishing this (see example below); however, if the <if_expr> evaluated control vector is applied to the arguments of function calls by default, then there seems to be no way to disable this control vector effect on the operation of function arguments. If the "else" part is present, the evaluated control vector applies to the <else_block> with complementary effect, e.g., the result elements are updated when the corresponding control bits are 0. This vector if-statement differs from the scalar if-statement in one significant way; namely, in the vector if-statement, both the <if_block> and the <else_block> are always executed.

Examples:

1.
```
if (a[*] != 5)
        a[*]++;
  else
        a[*] = 0;
  or,
  a[*] = (a[*] != 5) ? a[*]++ : 0;   /* mask operation */
```

e.g., if a[*] = 2,5,2,4,5, then the result a[*] is 3,0,3,5,0

2.
```
if (A[*] > 0) {
    B[*] = C[*] / A[*] + V[*];
    V[*] = fun(A[*], B[*]+C[*]);
```

```
}
```

the control vector (A[*] > 0) has an effect on the operations:
(1) T[*] = C[*] / A[*], (2) B[*] = T[*] + V[*] and (3) V[*] = func();
but not on the function argument B[*]+C[*], (note T[*] is a temporary vector).

e. SWITCH statement: The switch-statement has the form:

```
switch ( <switch_expr> ) {
    case <const_expr> : <case_block> ;
    case <const_expr> : <case_block> ;
        .
        .
    }
```

When the <switch_expr> and <const_expr> are evaluated into control vectors, these control vectors are applied individually to the statements in each <case_block>.

Example:

```
switch (V[0#10]) {
    case 0: break;
    case 1:
    case 2: A[2#10]++;
            break;
    default: A[2#10] = 0;
             break;
    }
```

e.g., if V = 1 0 2 2 1 3 4 2 0 0 and A = x x 1 2 3 4 5 6 7 8 9 0, then
the result A = x x 2 2 4 5 6 0 0 9 9 0, where x means "don't care".

The effect of the above switch-statement is to increment by 1 those elements of A[2#10] which have the corresponding V[0#10] elements 1 and 2, leave unchanged to those elements of A[2#10] which have the corresponding V[0#10] elements 0, and set the rest of the elements of A[2#10] to zero.

f. WHILE statement: The while-statement has the form:

```
while ( <while_expr> ) <while_block> ;
```

When <while_expr> is evaluated as a control vector, this control vector applies to the operations in the <while_block> and the while loop is iterated until the evaluated control vector contains all zero bits.

Example:

```
while (V[0#5] > 0)
        V[0#5]--;
```

e.g. if V[*] = {0,3,2,3,1}, then it goes as follows:

```
0 3 2 3 1 -- V
0 1 1 1 1 -- mask
0 2 1 2 0 -- V
0 1 1 1 0 -- mask
0 1 0 1 0 -- V
0 1 0 1 0 -- mask
0 0 0 0 0 -- final V
0 0 0 0 0 -- mask
```

The mask (V[0#5] > 0) is applied to V[0#5]--;

g. FOR statement: The for-statement has the form:

for (<init_expr> ; <for_expr> ; <update_expr>) <for_block> ;

When <for_expr> is evaluated as a control vector, this control vector applies to the operations in the <for_block> and <update_expr>, and the for loop is iterated until the evaluated control vector contains all zero bits.

Example:

```
for (i= 0; V[0#5] < = 5 ; i++,V[0#5]++)
        B[0#5][i] = V[0#5];
```

if V[*] = {1,2,3,4,5}, then V evolves as follows:

```
1 2 3 4 5 -- V
1 1 1 1 1 -- mask
2 3 4 5 6 -- V
1 1 1 1 0 -- mask
3 4 5 6 6 -- V
1 1 1 0 0 -- mask
4 5 6 6 6 -- V
1 1 0 0 0 -- mask
5 6 6 6 6 -- V
1 0 0 0 0 -- mask
6 6 6 6 6 -- V
0 0 0 0 0 -- mask
```

the final B is

```
1 2 3 4 5
2 3 4 5 x
3 4 5 x x
4 5 x x x
5 x x x x
```

where x means "don't care", i.e, the original value of B is not updated.
Note that the mask (V[0#5] < = 5) is applied to B[0#5][i] = V[0#5] and V[0#5]++.

4.3. Vector operators

Operators or functions can also be classified into two categories:

1. *elemental operators/functions:* This class of operators/functions operates on the object(s) element-by-element, and each element of the object is independent, i.e., the result has the same rank and shape as the operand(s).

2. *transformational operators/functions:* This class of operations performs transformations on the entire array rather than element-by-element.

Most of the standard scalar operators are extended to perform *element-by-element* operations; these include:

1. arithmetic operators: +, -, *, /, %,

2. relational and logical operators: < , < =, ==, > =, > , !=, &&, ||, ,! ,&, |, ~, ^, > > , < < ,

3. assignment (and arithmetic) operators: =, +=, -=, ++, etc.

The vector arithmetic and logical operations have the same syntax as in conventional C, i.e.,

```
<unary_operator> <vector_expr>
<vector_expr> <binary_operator> <vector_expr>
```

There are several ways to extend the syntax of C to include the "new" vector operations. We have considered introducing new operators, using new keywords or using a function notation. The operator form permits succinct expression; the keyword form makes a program easy

to read, while the function form enhances the portability of a program. From an efficiency point of view, it would be nice to have a set of special-character operators (as in APL [Iver62]) to reflect the hardware capabilities; however because of the limited set of special-characters and the rich set of hardware instructions, this approach is impractical. Another plausible approach is to use multiple characters (e.g., /+, @!, etc.) as an operator. Although, this may make the language difficult to read initially (as compared with keywords), it has the advantage of conciseness and efficient lexical analysis (a tradition of C); therefore, we will define multiple-character operators whenever appropriate.

4.3.1. The vector descriptor operator

The operator @ is used to declare a vector descriptor when it is used in the declaration part of a program, and in most cases, to assign a vector to a descriptor. A descriptor pointing to a local/global vector is called a *normal descriptor*, while a descriptor pointing to a temporary vector located in the dynamic space [Li83] is called a *dynamic descriptor*. Normal descriptors point to the vector directly and, hence, can be an l-value and/or an r-value without any side effects. However, dynamic descriptors do not point to the "actual" vectors, but rather point to temporary vectors in the dynamic space. Thus, if a dynamic descriptor is used as an l-value, the result of an operation does not affect the original vector which this dynamic descriptor "tried" to represent. Dynamic descriptors created within a procedure remain active until that procedure is exited.

Example: given

```
int va[200];
float vb[10][10];
int @descp1, @descp0;
float @descp2;

@descp0 = va[2#100];
@descp1 = va[2#100:2];
@descp2 = vb[0][0#100];
```

then,

```
descp0 = va[100#100]    -- assign va[100#100] to va[2#100],
descp2 = descp1         -- assign va[2#100:2] to vb[0][0#100].
```

where descp0 and descp2 are normal descriptors pointing to local vectors, and descp1 is a dynamic descriptor representing a temporary vector which is located in the dynamic space (because vectors which are not stride-1 are automatically gathered into the dynamic space).

A descriptor is similar to a pointer in C, so the C pointer arithmetic scheme [KeRi78] can be applied to descriptors too; in other words, VC provides a *descriptor arithmetic* capability, e.g.,

```
@descp0++; -- descp0 is modified to represent va[3#100]
descp0++;  -- increments all elements in va[3#100] by one
```

4.3.2. Operators for accessing information in a descriptor

Two operators for accessing the components of a descriptor are provided: @# accesses the length and @& accesses the address. For example, leng = @# descp1 causes leng = 100, and addr = @& descp1 causes addr = &va[2]. These may be useful, for example, when passing a descriptor to a function; the callee can use these operators to obtain information about the vector which was passed as an argument.

4.3.3. Reduction operators (transformational)

Several reduction operators are defined with multiple characters; these include: /. (dot product), /+ (vector sum), /$ (vector product), /< (minimum), and /> (maximum). They are level-1 constructs. As a matter of fact, the Cyber 205 has hardware instructions (macro instructions implemented in microcode) corresponding to these reduction operators; hence their implementation is economical.

Examples:

1. dot product

```
scal = VA[*] /. VB[*]; /* VA and VB have length n */
```

is equivalent to:

```
scal = 0;
for (i=0; i < n; i++)
   scal += VA[i] * VB[i];
```

2. vector sum

```
scal = /+ VA[0#100:2];
```

is equivalent to:

```
scal = 0;
for (i=0; i < 200; i += 2)
   scal += VA[i];
```

3. vector product

```
scal = /$ VA[1:100:2];
```

is equivalent to:

```
scal = 1;
for (i=1; i <= 100; i += 2)
   scal *= VA[i];
```

4. minimum

```
scal = /< VA[1:100];
```

is equivalent to:

```
scal = VA[1];
for (i=2; i<= 100; i++)
      if (VA[i] < scal)
         scal = VA[i];
```

5. maximum

```
scal = /> (VA[1:20] + VB[2:21]);
```

is equivalent to:

```
scal = VA[1] + VB[2];
for (i=2; i<= 20; i++)
   if (( tscal = VA[i] + VB[i+1]) > scal)
      scal = tscal;
```

These reduction operators are extended so as to be able to select the dimension on which the reduction is made. The extented syntax is as follows:

< unary_reduct_op> '[' < *dimension* > ']' < vector_expr>
< vector_expr> < binary_reduct_op> '[' < *dimension* > ']' < vector_expr>

e.g. v[*] = /+ [1] ma[*][*] reduces a matrix to a vector; the '[1]' means toward the first dimension. The dimension number is defined in the order 1, ..., n from left to right for n-dimensional arrays. The default *dimension* is 0, i.e., the reduction operation is performed on all elements of the specified array and the result is a scalar as shown above. For example, if ma[*][*] is the 2 by 4 matrix,

1 3 5 4
2 4 3 3

then, v[*] = /+ [1] ma[*][*] is evaluated to v[*] = {3, 7, 8, 7}, and v[*] = /+ [2] ma[*][*] is evaluated to v[*] = {13, 12}.

4.3.4. Other vector operators

Some other operations which have syntax similar to that defined above, are also provided; these include:

a. unary operators: @¯ (ceiling), @_ (floor), @| (absolute), @' (square root), @% (delta), @\ (reversal), /| (count one), /& (count equal),

b. binary operators: @/ (average), and

c. ternary operators: @! # (interval), @0 # (mask zero), @1 # (mask one).

For examples:

```
1. VA[*] = @| VB[*];          -- VA[i] = | VB[i] |, for all i,
2. VA[*] = @% VB[*];          -- VA[i] = VB[i+1] - VB[i], for all i,
3. VA[*] = @\VB[*];           -- reverse vector VB,
4. V[*] = VA[*] @/ VB[*];     -- V[i] = (VA[i] + VB[i])/2, for all i,
5. S = /| BV[*];              -- count the number of '1' bits in a bit vector,
6. VA[*] = i @! j # n;        -- VA[*] = i, i+j, ... , i+(n-1)j,
7. BV[*] = a @1 b # n;        -- a '1' followed by (b-a) '0' until length n,
8. BV[*] = a @0 b # n;        -- a '0' followed by (b-a) '1' until length n.
```

where, BV[*] is a bit vector, S is a scalar value, hence /& and /| are reduction operators.

4.3.5. Precedence and order of evaluation

The precedence table in [KeRi78] is augmented with extended VC operators [Li83]. The reduction operators and unary vector operators are right-associative and have higher precedence than all scalar operators except for '(', ')', '[', ']', '.' and '->'. The binary and tenary vector operators are left-associative and have lower precedence than most of the scalar operators.

For examples:

```
  a[*] = /+ b[*] + c;
is the same as
  a[*] = (/+ b[*] ) + c;

  a[*] = 3 + i @! j + 4 # n * m;
is the same as
  a[*] = (3 + i) @! (j + 4) # (n * m);
```

5. Built-in Functions

The main motivations for built-in functions were mentioned in Section 3.2. Some other reasons may be 1) we were unable to define reasonable operators, e.g., SEARCH, SELECT and SPARSE have more than three arguments [CDC81a], and 2) frequently used functions in some areas of application, e.g. MATMUL, ROTATE, TRANSPOSE, etc. FORTRAN 8X [FORT82] has specified several intrinsic functions; some of them may be candidates for the VC built-in library.

We can observed that there is a cost of using functions (as compared to operators or inline code), namely the overhead incurred by the prologue and the epilogue of function modules.

6. Vector Function Calls

When a vector is passed as an argument to a function (or procedure), only the descriptor, either user specified (explicit descriptor) or compiler generated (implicit descriptor), is passed. By using a dynamic space (another stack which is not in the C run-time stack [LiSc83]), vector-valued functions can be implemented. Appendix 12.1 shows a simple example of function calls, a vector-valued function is declared by '@'.

7. General Examples

The following examples demonstrate some of the extended operators and syntax of Vector C; more examples can be found in [Li83].

1. Sum all the diagonal elements of a matrix a, i.e., $S = \sum_{\substack{i,j=0 \\ i \neq j}}^{N-1} |a_{ij}|$

```
float  a[N][N],P;
bit    bv[N][N];

bv[*][*] = 1 @1 (N+1) # N*N;  /* 1 followed by N 0's until length N*N */
P = /+ @| a[0][bv[*][*]];     /* compress */
```

or

```
P = /+ @| a[0][0 # N : N+1 ];  /* periodic gather */
```

Among these two alternatives, the second one (periodic gather) is much faster than the first one, because the density of the bit mask (bv) is very low when N gets large. Analytic timing analyses [CDC82] of these two solutions show that $(244 + 1.5N + 0.56N^2)$ cycles and $(186 + 2.75N)$ cycles respectively are required for these statements, (each cycle is 20 *nanoseconds*).

2. Matrix multiplication, $C_{n \times m} = A_{n \times l} * B_{l \times m}$. There are several ways to implement matrix multiplication [HoJe81], e.g.,

   (1) inner product

```
for (i = 0; i < n; i++)
```

```
for (j = 0; j < m; j++)
   C[i][j] = A[i][*] /. B[*][j];
```

(2) middle product

```
for (i = 0; i < n; i++)
   for (k = 0; k < l; k++)
      C[i][*] += A[i][k] * B[k][*];   /* Linked-triadic operation*/
```

(3) outer product

```
for (i = 0; i < l; i++)
      C[*][*] += spread(A[*][i],2,m) * spread(B[i][*],1,n);
```

or,

```
for (i = 0; i < l; i++) {
      IV1[*] = i @! 0 # m;
      IV2[*] = i @! 0 # n;
      C[*][*] += A[*][IV1[*]] * B[IV2[*]][*];
      }
```

where, the first spread function expands vector A[*][i] to the right (see the figure below) for m times, and the second one expands vector B[i][*] upwards n times.

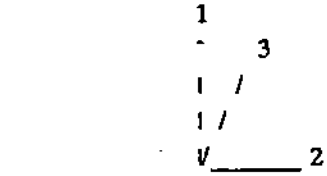
For example:

if A[*][i] = {1,2,3} then spread(A[*][i],2,4) is

```
1 1 1 1
2 2 2 2
3 3 3 3
```

and if B[i][*] = {4,5,6,7} then spread(B[i][*],1,3) is

```
4 5 6 7
4 5 6 7
4 5 6 7
```



(4) n^3 parallelism

```
C[*][*] = /+[2] (spread(A[*][*],3,l) * spread(B[*][*],1,n));
```

3. Solve a first order recurrence equation: $x_i = x_{i-1}+d_i$ for i = 1,2,...,N and x_0=0, partial results are needed; use cascade partial sum technique [HoJe81].

```
#define N 1024
#define LOGN 10
float   x[N+1], d[N+1];
int     i;

x[*] = d[*];
k = 1;
for (i = 0; i < LOGN; i++) {
        x[k:N] += x[1:N-k];          /* shift right */
        k *= 2;
        }
```

4. Use MVA (Mean Value Analysis) algorithm [Schw80] to solve a single-class closed QNM (Queueing Network Model) with K devices and multi-programming level N, i.e., to find the performance attributes at each station (device). Assume the queueing discipline is FCFS.

```
#define K 20
#define N 10

float V[K], S[K], R[N][K], Nbar[N][K], X0[N], X[N][K], U[N][K];
int n;


Nbar[0][*] = 0;
for (n = 1; n <= N; n++) {
        R[n][*] = S[*] * (1 + Nbar[n-1][*]);    /* device response time */
        X0[n] = n / (/+ (V[*]*R[n][*]);         /* system throughput rate */
        X[n][*] = X0[n] * V[*];                 /* device throughput rate */
        Nbar[n][*] = X[n][*] * R[n][*];         /* mean queue length at each device */
        U[n][*] = S[*] * X[n][*];               /* device utilization */
        }
```

8. Implementation and Timing Information

The implementation of VC has not been completed. We still need to insure that all constructs proposed here are really useful and necessary. For example, CONTROL, MASK, MERGE, and IF, etc. seem to be essential (these have already been implemented), but the usefulness of other constructs such as WHILE, SWITCH, FOR etc. is still uncertain.

The implementation of VC was accomplished by enhancing the entire Amsterdam EM compiler Kit [TSKS83][LiSc83], namely the Front-end, the Optimizer and the Back-end. Also, the EM intermediate code was extended to include vector intermediate code. The storage class **register** was implemented, so that the Cyber 205 high speed register file (256 registers) is further utilized; this leads to smaller run-time stack and faster execution of programs. The reserved keywords **fortran** and **asm** were also implemented. Thus, users can take advantage of existing application libraries (e.g., MAGEV [MAGE83] - MAthematical/GEophysical Vector library, contains highly optimized FORTRAN/META subroutines/functions); and, for experienced programmers, the **asm** statement provides a way of accessing low level instructions when necessary.

Example:

```
#define FREE    asm("     rtor    pdy,dyn_stack")
fortran float FORTFUNC(); /* Fortran module names must be upper case */
float a[10], b, c;

c = FORTFUNC(a, &b);    /* call-by-reference */
asm("  wjtime  c_1n");      /* reset timer */
FREE;  /* Free all dynamic space allocated so far in the current procedure */
```

A special feature of the Cyber 205 hardware is the *linked triadic* operation [CDC81a], akin to *chaining* in the Cray-1 computer [Russ78]. By using this feature, two vector operations can be combined into nearly one vector operation for certain sequences of vector instructions (e.g., +, -, *, &, |, ^, ~, and some logical compare operations). The requirements are 1) both operations must be different, 2) there must be at least one scalar and at least one vector operand, and 3) logical compare operators can apply only to the second evaluated operation.

(The hardware linked triadic instruction is restricted, but these restrictions can be relaxed by use of software [Li83]).

Examples:

```
float  a[100], b[100], c[100];
bit    z[100];
float  d, e;

a[*] = b[*] * c[*] + d;
z[*] = (a[*] + d) >= b[*];
a[*] = b[*] - c[*] + e - c[*] * d;  /* two linked triadic */
```

In the following sections, some empirical results based on the currently available VC constructs are presented.

8.1. Timing information of some 205 operations

A simple Vector C program was written to measure the execution speed of some 205 vector floating point arithmetic operations (namely, add, multiply, divide and linked triadic operations) under varying vector lengths, (see Table 2). In this table, we can see that the speed of divide operation is much slower than that of add and multiply operations; the reason is that the division functional unit is not pipelined on the 205. This table confirms that the maximum or asymptotic performance (r_∞) of addition and multiplication operations is 100 MFLOPS (Million FLoating-point Operations Per Second) for two pipes (Purdue configuration) and full word arithmetic operations, and the half performance length ($n_{1/2}$) is 100 [HoJe81].

8.2. Comparison of timings between VC and Fortran for the conjugate gradient algorithm.

One example of problems which can exploit the vector capabilities of the 205 is solving a large system of equations, $A \cdot \vec{x} = \vec{f}$, using the *conjugate gradient* algorithm. A 205 Fortran implementation of this algorithm was given in [GaRR83]. A direct conversion to C of this Fortran conjugate gradient program is in Appendix 12.2. The timing comparison between these two programs is shown in Table 3, which indicates that VC compiler generates faster

Length	Addition	Multiplication	Division	Linked-triadic
10	9.61168781	9.61538462	5.68214103	11.89980365
20	16.66666667	16.66666667	7.57547063	21.73913043
40	29.40960224	29.41176471	11.11141976	40.00000000
60	37.50000000	37.50000000	12.09701809	53.57142857
80	45.45454545	45.45454545	12.98722382	66.66666667
100	50.00000000	50.00000000	13.29787234	75.75757576
200	67.56528496	67.56756757	14.70599049	111.10802478
400	80.51367726	80.64678723	15.29040298	142.85459188
600	86.20689655	86.20689655	15.52799050	157.89473684
800	89.28571429	89.28571429	15.64939104	166.66666667
1000	91.24087591	91.24087591	15.72324572	172.41379310
2000	95.30935032	95.42030258	15.85287266	185.18347052
4000	97.65386587	97.65625000	15.92862067	192.30353189
6000	98.41696315	98.42116055	15.94547498	194.80013506
8000	98.75421557	98.72045945	15.95089040	196.07338537
10000	99.01637137	99.04353657	15.95780246	196.83373258

Table 2: Execution Speeds (in MFLOPS)

executing code than 205 Fortran compiler.

Size	VC	Fortran	Speed-up(%)
8	3161	3871	18.34
16	5968	6889	13.37
32	17145	18759	8.6
64	62928	66729	5.7

Table 3: Execution Times for Conjugate Gradient Algorithm (in microseconds)

8.3. Sorting

Several vector sorting algorithms were presented in [Ston78] and [Moss82]. Stone's paper gives the analysis of two sorting algorithms (Quicksort and Batcher Sort) on a STAR 100 machine [HiTa72], while Mossberg, working on the Cyber 205, analyzed several sorting algorithms (Bubble Sort, Insertion Sort, Merge Sort, Quicksort, Batcher Sort, Diamondsort, and

Heap Sort). The conclusion in both of these papers was that Quicksort is the fastest sorting algorithm for vector machines.

Table 4 shows the running times of BSORT and QSORT, where BSORT is a Batcher Sort (perfect shuffle exchange scheme) implemented in VC (see Appendix 12.3) and QSORT is a Quicksort subroutine, called from a VC program; QSORT is provided in the MAGEV library and implemented in META (the 205 assemble language). In Table 4, the column $BSORT_1$ is the timing results excluding procedure call overhead.

Batcher Sort has a sequential computational complexity of $O(N*\log^2 N)$, while Quicksort has a complexity of $O(N*logN)$, where N is the length of the sorted vector. The QSORT is reported [MAGE83] as having a time complexity of $3.75*N*(28 + \ln N)$. This means that when $N < e^{28} (\approx 1.4*10^{12})$, QSORT has a linear behavior, (see Figure 2).

Length	BSORT	$BSORT_1$	QSORT
16	242	167	45
32	392	274	83
64	604	429	167
128	951	706	346
256	1516	1209	708
512	2656	2281	1419
1024	5162	4696	2844
2048	10786	10220	5663
4096	23664	22993	11742
8192	53250	52455	23402
16384	130726	129683	47353
32768	316994	315539	96801

$BSORT_1$ - excluding procedure call overhead

Table 4: Execution times for Batcher Sort and Quicksort (in microseconds)

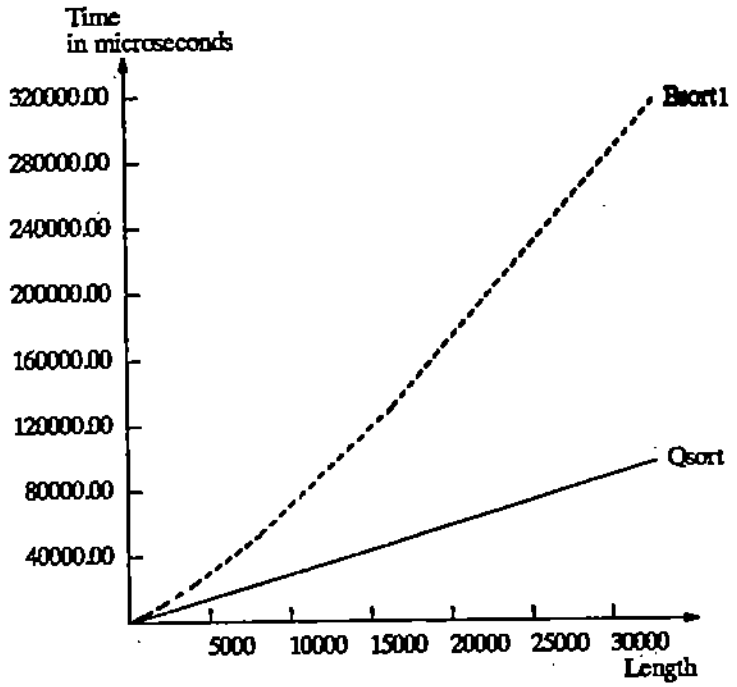


Figure 2: Plot of Table 4

8.4. Matrix Multiplication

As discussed in Section 7, there are several ways to perform matrix multiplication on a vector processor. One of these, the middle product method, was implemented in both Vector C and 205 Fortran. Table 5 shows the execution speeds of this matrix multiplication method for different sizes and versions. In Table 5, *Size* is the matrix size (all are square matrices of *Size x Size*), MXMPYR is a MAGEV library routine, which is called from a VC program, and the execution speed (in MFLOPS) is calculated as *Time* / (*Size***Size**(2**Size* −1)).

9. Summary

The development of Vector C on the Cyber 205 has extended the features of CC205, the earlier scalar complier [LiSc83]. This version of Vector C provides users in the C community with a means of utilizing vector processing capabilities and of accessing highly optimized software libraries (e.g., MAGEV).

Size	VC	Fortran	MXMPYR
10	6.86	6.98	14.80
50	31.77	31.76	43.91
100	55.12	54.30	73.16
150	72.25	72.19	91.87
200	82.53	83.07	101.54
250	93.31	93.34	112.01
300	101.77	101.43	119.85

Table 5: Execution speeds for Matrix Multiplication (in MFLOPS)

Vector C exhibits a more concise and modern syntax than 205 Vector Fortran, e.g., gather/scatter, compress/expand, vector function calls, descriptor arithmetic, and flexible control structures. Also, the implementation of Vector C generates better code than 205 Fortran in some cases. As an example, in Vector C, the vector expand leaves the unmasked area (say, the area with control bit 0) unchanged, but in 205 Fortran, the Q8VXPND call [CDC83] clears this unmasked area.

In view of the increasing number of vector computers, portability could become an important issue. In order to achieve both portability and efficiency, one approach could be to build a compiler and a preprocessor, where the compiler accepts the (proposed) vector constructs and produces efficient code, while the preprocessor translates the new constructs into statements in the base language and/or library function calls. Portability then can be achieved by providing a function library (just as the standard I/O library in the scalar C). Recall that the portability of the conventional C language is enhanced by having all machine dependent issues delegated to run-time library functions.

There are several issues of interest which have not yet been resolved; these include:

a. *Portability:* Vector C is designed to facilitate the design and implementation of vector-oriented algorithms; is it portable to other vector machines (e.g. the CRAY series)?

b. *Extensibility:* How easy/hard would it be to extend VC to machines with multiple vector processors (e.g. CRAY X-MP or ETA GF-10)? In other words, is it possible to design more constructs (to support parallel operations) on the top of VC - a superset of VC?

c. *Usability:* Some work has been done on the compiling techniques (lexical analysis, parsing and code generation) for a vector machine [DoKa75][Kroh75][Fisc80], but it is likely that none of these have been applied to a real compiler. Since conventional C has been used as an implementation language for many compilers (e.g., PASCAL, FORTRAN 77, ADA, etc.) and Vector C provides vector capabilities, will the availability of VC help fulfill the earlier work and lead to the development of new compiling techniques?

We believe that Vector C will be of great use to programmers devising new algorithms for vector computers. Using the vector constructs of the language, users can employ a direct, almost intuitive programming style. The use of C as the base language means that many programmers can use Vector C with little learning effort required. The ability to call routines in the Fortran compatible libraries mean that existing routines (often highly optimized) can be easily invoked. The power of vector supercomputer systems to solve large problems is great; making effective use of them will continue to be a challenge. Vector C is one step in a progression of steps addressing this challenge.

10. Acknowledgments

This paper has received valuable comments from the following members of the Department of Computer Sciences at Purdue: Dennis Gannon, Bradley Lucier, Jairo Panetta, John Rice, Saul Rosen, and Marinus Veldhorst.

11. References

[Aust79] J.H. Austin, "The Burroughs Scienctific Processor," *Infotech State of the Art Report: Supercomputers*, Vol 2 ed. C.R. Jesshope and R.W. Hockney, pp1-31, 1979.

[BDMR72] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick, "The Illiac IV System," *Proceeding of the IEEE*, Vol. 60, No. 4, April 1972.

[BSP77] BSP, "Burroughs Scientific Processor - implementation of FORTRAN," *Burroughs Document 61391E, 1977*.

[CDC81a] "CDC CYBER 205 Hardware Reference Manual," *Control Data Corporation*, 1981.

[CDC82] "Engineering Specification," *Control Data Corporation*, No. 10358026, September 1982.

[CDC83] "FORTRAN 200 Version 1 Reference Manual," *Control Data Corporation*, 1983.

[DoKa81] M.K. Donegan and S.W. Katzke, "Lexical Analysis and Parsing Tcechniques for a Vector Machine," *SIGPLAN Notices*, p138-145, March 1981.

[Fisc80] C.N. Fischer, "On Parsing and Compiling Arithmetic Expressions on Vector Computers," *ACM Trans. Programming Languages and Systems*, April, 1980.

[Flyn72] M.J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Computers*, September 1972.

[FORT82] "Proposals Approved for FORTRAN 8X," *X3J3/S6.81* May 1982.

[GaRR83] D. Gannon, J. Rice and S. Rosen, "CS590V - Vector and Parallel Computing," *Class Notes*, Dept. of Computer Sciences, Purdue University, 1983.

[HiTa72] R.G. Hintz and D.P. Tate, "Control Data STAR-100 Processor Design," *COMPCON'72 Digest*, pp 1-4, 1972.

[HoJe81] R.W. Hockney and C.R. Jesshope, *Parallel Computers*, Adam Higer Ltd. Bristol. 1981.

[ICL79] ICL, "DAP: FORTRAN language reference manual," *ICL Tech. Pub.* TP6918, 1979.

[Iver62] K.E. Iverson, "A Programming Language," *John Wiley and Sons*, 1962.

[KeRi78] B.W. Kernighan and D.M. Ritchie, "The C Programming Language," *Prentice-Hall, Inc.* 1978.

[Kroh75] H.E. Krohn, "A Parallel Approach to Code Generation for FORTRAN-like Compilers," *SIGPLAN Notices*, p146-152, March 1981.

[Li83] K.C. Li, "Design of the Vector C language (for the Cyber 205)," *Working Notes*, December 1983.

[LiSc83] K.C. Li and H.D. Schwetman, "Implementing a Scalar C Compiler on the Cyber 205," *Software-Practice and Experience*, (to appear).

[MAGE83] "MAGEV - Mathematical/geophysical vector library," *Purdue University, Computer Center*, V3.1 (3/83).

[Moss82] B. Mossberg, "Sorting on the CYBER 205," *Proceedings of Symposium on CYBER 205 Applications*," Institute for Computational Studies at Colorado State University, August 1982.

[PaWi75] G. Paul and M.W. Wilson, "The VECTRAN language: an experimental language for vector/matrix array processing," *IBM Research Report 320-34, 1975*.

[PeCM83] R.H. Perrott, D. Crooles, and P. Millign, "The Programming Language ACTUS," *Software-Practice and Experience*, Vol. 13, p305-322, 1983.

[Redd73] S.F. Reddaway, "DAP - a distributed array processor," *1st Annual Symp. on Computer Architecture (IEEE/ACM)*, 1973.

[Russ78] R.M. Russell, "The CRAY-1 Computer System," *Comm. ACM*, pp 63-72, January 1978.

[Schw80] H. Schwetman, "Implementing the Mean Value Analysis Algorithm for the Solution of Queueing Network Models," *CSD-TR-355, C.S. Dept., Purdue Univ.*, October 1980.

[Stev75] K. Stevens, "CFD - a FORTRAN-like Language for the ILLIAC IV," *SIGPLAN Notices*, March 1975.

[Ston78] H.S. Stone, "Sorting on STAR," *IEEE Trans. Software Engineering*, March 1978.

[TSKS83] A.S. Tanenbaum, H. Van Staveren, E.G. Keizer, and J.W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," *Comm. ACM, Vol 26*, pp 654-660, September 1983.

12. Appendix

12.1. Vector function calls

The following simple example demonstrates passing vectors as function arguments and returning vectors from functions (vector-valued functions).

```
#define N 100
main()
{
    float  v[N], m[N][N], temp[N];
    float  b;

    float  @vd;                  /* descriptor */
    float  fun1(), fun2(), fun3(); /* scalar functions */
    float  @fun4();              /* vector function */

    b = fun1(v,N);               /* pass address and length */
    b = fun2(v[0#N]);            /* pass a descriptor which is formed implicitly */
    @vd = v[0#N];                /* setup descriptor explicitly */
    b = fun2(vd);
```

```
        b = fun3(v,N);                  /* call a scalar function */
        m[2][*] = fun4(v[*], vd) + 3;
}

/* performing vector operations */
float fun1(v,n)
float  *v;
int    n;
{
        float  scal;

        scal = /+ v[0#n];
        return(scal);
}

/* performing sequential or vector operations */
float fun2(vd)
float  @vd;
{
        float  scal, *p;
        int    n;

        n = @# vd;                      /* get vector length */
        if (n > 5)
            scal = /+ vd;
        else {                          /*  short vector -- use scalar instructions */
            p = @& vd;                  /* get address */
            scal = 0.0;
            for (i = 0; i < n; i++)
                scal += *p++;
            }
        return(scal);
}

/* fun3() is a pure scalar function */
float fun3(v,n)
float  v[];
int    n;
{
        float  scal = 0.0;
        int i;

        for (i = 0; i < n; i++)
            scal = scal + v[i];
        return(scal);
}

/* function returning a vector (descriptor) */
float @fun4(v1d,v2d)
float @v1d, @v2d;
{
        return(v1d + v2d * 20);
}
```

12.2. Solve a large system of equations, $A \cdot \vec{x} = \vec{f}$, using *conjugate gradient scheme.*

```
/*
      Conjugate Gradient Scheme
 */
#define N 16
#define NSQ N*N
#define M  N+2
#define NM  N*M
#define K 100

main()
{
      float x[NSQ], r[NSQ], p[NSQ], f[NSQ], ap[NSQ], t[M][M], apt[M][M];
      register float   rr, a, b, oldrr;
      register float   @tad, @tbd, @tcd, @tdd, @ted, @aptd;
      register float   @pd, @fd, @rd, @xd, @apd;
      bit     bv[NM];
      register bit @bvd;
      register int i;

      @tad = t[1][1#NM];
      @tbd = t[0][1#NM];
      @tcd = t[2][1#NM];
      @tdd = t[1][2#NM];
      @ted = t[1][0#NM];
      @aptd = apt[0][0#NM];
      @pd = p[*];
      @fd = f[*];
      @rd = r[*];
      @xd = x[*];
      @apd = ap[*];
      @bvd = bv[*];


      bvd = N @1 M # NM;            /* build bit vector N one's followed by 2 zeros */

      fd = 1.0 @1 1.0 #NSQ;         /* setup f vector -- 1.0,2.0,3.0... */
      rd = fd = pd = 2.47 * fd * fd; /* 2.47 is broadcast */
      xd = 0.0;
      oldrr = rd /. rd;             /* dot-product */
      t[0][0#(M*M)] = 0.0;

      for (i = 0; i < K; i++) {
            (&t[1][1])[bvd] = pd;   /* expand p to t */
            aptd = bvd               4.0 * tad - tbd - tcd - tdd -ted;
            apd = apt[0][bvd];      /* compress apt into ap */

            a = oldrr / (apd /. pd);
            xd += a * pd;
            rd -= a * apd;
            rr = rd /. rd;
            b = rr / oldrr;
            pd = rd + b * pd;
            oldrr = rr;
```

```
            }
}
```

12.3. Batch sort - perfect shuffle exchange scheme

```
/*
      Batcher Sort
 */
#define N 16
#define NH N/2
#define LOGN 4
bit   shubv[N];
main()
{
      int  x[N];
      register int i,j,k;

      /* initialition */
      srand(1);
      for (i = 0; i < N; i++)
            x[i] = rand() % 799;

      shubv[*] = 1 @1 2 # N;    /* create shuffle bit mask 101010... */

      cm_ex(x,1);
      shuffle(x);
      for( i=1; i < = LOGN -2; i++) {
            for ( j = 1; j < = LOGN - 1 - i; j++)
                  shuffle(x);
            for ( j = LOGN-i; j < = LOGN; j++) {
                  cm_ex(x,j+i+1-LOGN);
                  shuffle(x);
                  }
            }
      for( i=1; i < = LOGN; i++) {
            cm_ex(x,0);
            shuffle(x);
            }
}
/*
      Compare Exchange
 */
cm_ex(x, j)
register int j, *x;
{
      register int @x1d, @x2d, @t1d, @t2d;
      int     t[N];
      bit     bv1[N], bv2[N];
      register bit @bv1d, @bv2d;

      @x1d = x[0#NH];
      @x2d = x[NH#NH];
      @t1d = t[0#NH];
      @t2d = t[NH#NH];
```

```
        @bv1d =bv1[0#NH];
        @bv2d =bv2[0#NH];

        bv1d = x1d < x2d;

        if (j) {
                k = 1;
                for (i=1; i< j; i++) k *= 2;
                bv2d = k @0 (2*k) # NH;
                bv1d = bv1d ^ bv2d;
                }
        t1d = bv1d ? x1d : x2d;
        t2d = bv1d ? x2d : x1d;
        x1d = t1d;
        x2d = t2d;
}
/*
        Shuffle
 */
shuffle(x)
register int *x;
{
        register int @td;
        int t[N];

        @td = t[0#N];

        td = shubv[*] ? x[0#NH] <> x[NH#NH];
        x[0#N] = td;
}
```