# RETARGETABLE COMPILER OF ANSI C SUPERSET FOR VECTOR AND SUPERSCALAR COMPUTERS

S. Katserov, A. Lastovetsky
(Computer Centre for Scientific Research, Moscow State University,
Vorob'ovy gory, 119899, Moscow, Russia)
S. Gaissaryan, D. Khaletsky, I. Ledovskih
(Institute for System Programming, Russian Academy of Sciences,
25, B.Kommunisticheskaya ul., 109004, Moscow, Russia
e-mail: ssg@ivann.delta.msk.su)

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

**Key words**: parallel programming, C language, vector computers, superscalar computers.

## 1 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 [1].

As computer architectures have changed it has become necessary to reflect the changes in the compiler's internal languages, by adding constructs to express new computing facilities, such as vector calculations. But if we want to use these new facilities explicitly in programs, they should also be 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 languages, named C[], which allows one to write portable efficient programs for SIMD (vector and superscalar) computer archutectures [2]. Our motivation of the C[] language is given in sec. 2.

In sec. 3 we give a brief description of the C[] language as it is published in [2]. In sec. 4 two new statements added to C[] after publishing [2] are discussed. In sec. 5 we discuss principles of C[] implementation for superscalar computers and especially for Intel i860. In sec. 6 the retargetable compiler prototype system is described.

## 2 Motivation

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], [4], [5], etc.), but the supersets we know have 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 the 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 imlementation for all computers of the class.
- The superset must be suitable for implementation of the portable and retargetable compilers.

## 3 Brief description of the C[] language

The C[] language is a strict superset of ANSI C [6]. The following a brief description of its main features as it was described in [2].

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. In contrast to arrays, a vector is not an object, it is a new sort of value.

An array is a container of vector values. The unary postfix [] operator is applied to a operand of array type and provides an access to the vector, being the value of the array object. Formally the [] operator cancels the conversion of the operand to a pointer.

The notion of array in the C[] language is extended by adding new attribute, namely, the step of allocation of array members in storage (in particular, it allows us to introduce the notion of subarray sensible enough). Correspondingly, the notion of pointer is extended as well as address arithmetic. Namely, a pointer has new attribute step, and address arithmetic takes into account this new attribute.

A formal parameter of a function may have an array type. 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.

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

The operand of unary `*`, `+`, `-`, `~`, `?`, `%`, `@`, `!` operators and scalar cast operators may have a vector type. One or both operands of binary `*`, `/`, `%`, `?<`, `?>`, `+`, `-`, `<<`, `>>`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&`, `^`, `|`, `&&`, `||` operators may have vector type. An assignment operator may have as its left operand an 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. The conditional operator may also have vector operands.

The linear (or reduce) operators are introduced. The unary linear `[*]`, `[/]`, `[%]`, `[?<]`, `[?>]`, `[+]`, `[-]`, `[<<]`, `[>>]`, `[&]`, `[^]`, `[|]` operators correspond to binary `*`, `/`, `%`, `?<`, `?>`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|` operators. These operators are applicable only to vector operands.

The set of C[] operators together with facilities of packing integer vectors into bit-fields provide a general set of vector manipulations in vector computers.

## 4 New constructs of C[]

In this section we discuss two new statements — **par** and **pipe**, — added to C[] after publishing of [2] to describe parallel and pipelined calculations.

The **par** statement is another form of compound statement: a list of statements is enclosed in braces `{` and `}` preceded by the keyword **par**. It means that all these statements may be executed in arbitrary order, in particular, in parallel. Each result of the execution is considered be correct even if it depends on the order the statements are executed.

The **par** statement allowes the programmer to express data dependencies of his program in terms of the source language. If the programmer is sure that there are no data dependencies between some statements, he may include them in **par** statement. In general it will lead to more profound program optimization.

The **pipe** statement is a special loop statement, which allowes to express "skewed" loops. The idea of **pipe** is borrowed from [7]. It has the form

**pipe** (<expression>$_{opt}$;<expression>$_{opt}$;<expression>$_{opt}$) <statement>

The <expression>s have the same semantics as those in `for` statement. If body of **pipe** is a compound statement, it may contain a special label `p+:`. It means that the part of the current iteration beginning with the statement marked by this label may be executed in parallel with the next iteration of the loop. If the label p+: is omitted it is implied that it marks the first statement and therefore all iterations of the loop can be executed in parallel. It is supposed that if the compound statement, being a pipe body, contains definitions of automatic variables, these variables are different for each iteration, that is if n iterations run in parallel, there are n different instances for each such variable.

## 5 C[] implementation for superscalars

It is obvious that C[] is suitable for vector pipelined computers But C[] is also suitable for superscalar computers because the vector expressions and constructs **par** and p+:, presented in sec. 4, allow to point parts of the program which can efficiently use their parallel facilities (pipelines).

It may be pointed out dicussing the mapping if C[] to Intel i860 microprocessor.

All scalar operators of C[] are translated to corresponding scalar operations of the i860 processor.

The vector operators of C[] can be mapped on pipelined i860 operations. For example, vector expresion

c[] = a[] + b[] may be mapped in the following sequence of instructions:

```
//**************************************************************************
//      c[]=a[]+b[]
//                        input:     r16 - address of vector a[]
//                                   r17 - address of vector b[]
//                                   r20 - vector size
//                        output:    r15 - address of vector c[]
//**************************************************************************
        fld.q      r0(r16),f8        //Load first 4 elements of A[]
        fld.q      r0(r17),f12       //Load first 4 elements of B[]
        mov        -4,r14            //r14 is used to decrement counter
        adds       r14,r20,r20       //dectement counter
        .dual                        //Enter dual mode
        bla        r14,r20,L0        //Set LCC flag
        adds       -16,r15,r15       //prepare C address
L0::    pfadd      f8,f12,f0         //Put first elements in pipe
        fld.q      16(r16)++,f20     //Load next 4 elements of A[]
        pfadd      f9,f13,f0         //Put next elements in pipe
        fld.q      16(r17)++,f24     //Load next 4 elements of B[]
        pfadd      f10,f14,f0        //Put next elements in pipe
        bla        r14,r20,L2        //Check if there are more elements
                                     //and set LCC
        pfadd      f11,f15,f16       //Put next elements in pipe
        nop
        .enddual                     //Exit dual mode
        pfadd      f0,f0,f17         //Get result from pipe
        nop                               //Still in dual mode
        pfadd       f0,f0,f18             //Get result from pipe
        nop                               //Still in dual mode
        pfadd       f0,f0,f19             //Get result from pipe
        br          END                   //Goto END
        fst.q       r16,16(r15)           //Store results
L1::    pfadd       f8,f12,f29            //Put next elements in pipe
        fld.q       16(r16)++,f20         //Load next 4 elements of A[]
        pfadd       f9,f13,f30            //Put next elements in pipe
        fld.q       16(r17)++,f24         //Load next 4 elements of B[]
        pfadd       f10,f14,f31           //Put next elements in pipe
        bla         r14,r20,L2            //Check if there are more elements
                                          //and set LCC
        pfadd       f11,f15,f16           //Put next elements in pipe
        fst.q       r28,16(r15)++         //Store results
        .enddual                          //Exit dual mode
        pfadd       f0,f0,f18             //Get result from pipe
        nop                               //Still in dual mode
        pfadd       f0,f0,f19             //Get result from pipe
        br          END                   //Goto END
        fst.q       r16,16(r15)           //Store results
L2::    pfadd       f20,f24,f17           //Put next elements in pipe
        fld.q       16(r16)++,f8          //Load next 4 elements of A[]
        pfadd       f21,f25,f18           //Put next elements in pipe
        fld.q       16(r17)++,f12         //Load next 4 elements of B[]
        pfadd       f22,f26,f19           //Put next elements in pipe
        bla         r14,r20,L1            //Check if there are more elements
                                          //and set LCC
        pfadd       f23,f27,f28           //Put next elements in pipe
        fst.q       r16,16(r15)++         //Store results
        .enddual                          //Exit dual mode
        pfadd       f0,f0,f29             //Get result from pipe
        nop                               //Still in dual mode
        pfadd       f0,f0,f30             //Get result from pipe
        nop                               //Still in dual mode
        pfadd       f0,f0,f31             //Get result from pipe
        fst.q       r28,16(r15)           //Store results
END::   nop
//****************************************************************************
```

The expression:

```
    c[] = k * a[] + b[]
```

where k is a scalar variable, may be treated in similar way; this example allowes to achieve pipeline chaining.

The linear (or reduce) operators can be mapped to sequence of instructions containing pipelined operations. For example, the expression

```
s = [+] a[]
```

may be mapped to the following sequence of instructions:

```
//***************************************************************************
// s = [+] a[]
//        input:                r16 - &a[]
//                              r17 - size of the vector a[] (must be >5)
//        output:               f16 - s
//***************************************************************************
//
       fld.d        r0(r16),    f20                  //Load first 2 elements
       mov          -2,         r21                  //Loop decrement for bla
       .dual                                         //Enter dual mode
       pfadd.ss     f0,         f0,         f0       //Clear adder pipe (1)
       adds         -6,         r17,        r17      //Decrement size by 6
       pfadd.ss     f0,         f0,         f0       //Clear adder pipe (2)
       bla          r21,        r17,        L1       //Initialize LCC
       pfadd.ss     f0,         f0,         f0       //Clear adder pipe (3)
       fld.d        8(r16)++,   f22                  //Ld 3th & 4th elements
//***************************************************************************
L1::   pfadd.ss     f20,        f30,        f30      //Add f20 to pipe
       bla          r21,        r17,        L2       //If more go to L2 after
       pfadd.ss     f21,        f31,        f31      //adding f21 to pipe and
       fld.d        8(r16)++,   f20                  //loading next f20:f21
       // If we reach this point, at least one element remains
       // to be loaded. r17 is either -4 or -3.
       // f20, f21, f22, and f23 still contain vector elements.
       // Add f20 and f22 to the pipeline, too
       pfadd.ss     f20,        f30,        f30
       br           sumup                            //Exit loop after adding
       pfadd.ss     f21,        f31,        f31      //f21 to pipe
       nop
L2::   pfadd.ss     f22,        f30,        f30      //Add f22 to pipe
       bla          r21,        r17,        L1       //If more go to L1 after
       pfadd.ss     f23,        f31,        f31      //adding f23 to pipe and
       fld.d        8(r16)++,   f22                  //loading next f22:f23
       // If we reach this point, at least one element remains
       // to be loaded. r17 is either -4 or -3.
       // f20, f21, f22, and f23 still contain vector elements.
       // Add f20 and f22 to the pipeline, too
       pfadd.ss     f20,        f30,        f30
```

```
        nop
        pfadd.ss     f21,         f31,         f31
        nop
sumup::.enddual                                         //Exit dual mode
        pfadd.ss     f22,         f30,         f30      //Still in dual mode
        mov           -4,         r21
        pfadd.ss     f23,         f31,         f31      //Last dual mode pair
        bte          r21,         r17,         done     //If there is one more
        fld.l        8(r16)++,    f20                   //element, load it and
        pfadd.ss     f20,         f30,         f30      //add to pipeline
        // Intermediate results are still in the adder pipeline.
        // Let A1:A2:A3 represent the current pipeline contents
//***************************************************************************
done:: pfadd.ss      f0,          f0,          f30      // 0:A1:A2     f30=A3
        pfadd.ss     f30,         f31,         f31      // A2+A3:0:A1  f31=A2
        pfadd.ss     f0,          f0,          f30      // 0:A2+A3:0   f30=A1
        pfadd.ss     f0,          f0,          f0       // 0:0:A2+A3
        pfadd.ss     f0,          f0,          f31      // 0:0:0       f31=A2+A3
        pfadd.ss     f30,         f31,         f16      // f16=A1+A2+A3
//***************************************************************************
```

But C[] program may contain portions of sequential code, and this code should also be optimized, especially the loops. **par** and **pipe** statements allow compiler to reduce delays of the instruction pipeline of superscalar processor and therefore to optimize scalar code portions.

In the case of loops **par** statement provides the information necessary for loop body optimization. **pipe** statement provides the information for mutual optimization of loop iterations.

## 6 The compiler prototype system

The portable and retargetable compiler prototype system was implemented on Sun SPARC Workstation in UNIX. To implement the compiler prototype system the Karlsruhe toolbox for compiler construction [8] was used.

The compiler consists of four stages. The first stage analyses the source program file and builds its internal representation (the abstract syntax tree).

The second stage translates the internal representation into an intermediate language being an extension of the RTL language used in GCC [1].

In the third stage, the intermediate program is tuned to the target computer.

The fourth stage is a retargetable code generator. Currently, it generates code for the Russian Cray-like supercomputer [9] and for the Intel i860 [10].

## References:

1. R.M.Stallman. Using and Porting GNU CC // Free Software Foundation, 675 Mass Ave, Cambridge, MA, May. 1992

2. S.Gaissaryan, and A.Lastovetsky. An ANSI C Superset for Vector and Superscalar Computers and its Retargetable Compiler //J. C Lang. Transl., v.5, No.3, March, 1994, p.p. 183 — 198

3. Kuo-Cheng Li, and H. Schwetman. Vector C: A Vector Processing Language // J. Parall. Vect. Comput. v.2, No.2, 1985, p.p. 132-169

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

5. R.Gisselquist. An experimental C Compiler for the Cray-2 Computer // ACM SIGPLAN Notices, 21(9), 1986, p.p. 32-36

6. ANSI. Programming Language C. X3.159-1989. Americn National Standard Institute, 1989

7. T.G.Lewis. Foundations of Parallel Programming: A Machine-Independent Approach. / IEEE Comput. Soc. Press, 1994

8. J.Grosch. Toolbox Introduction. / Compiler Generation Report No 25, GMD Forchungsstelle an der Universitaet Karlsruhe, 1992

9. V.A.Melnikov, Ju.I.Mitropolsky, V.Z.Shnitman, V.P.Ivannikov, A.N.Tomilin, and S.S.Gaissaryan. High Performance Computer System "Electronica SSBIS" / Proc. Internat. Conf. Parallel Computing Technologies, Novosibirsk, Sept. 7 — 11, 1991, p.p. 47-55

10. **i860** Microprocessor Architecture. / Intel, Osborne, McGraw-Hill, 1990