論文 / 著書情報
Article / Book Information

Title	DSP Compiler for Matrix and Vector Expressions with Automatic Computational Ordering
Authors	NOBUHIKO SUGINO, Seiji Ohbi, AKINORI NISHIHARA
Citation	IEICE Trans. Fundamentals, Vol. E78-A, No. 8, pp. 989-995
Pub. date	1995, 8
URL	http://search.ieice.org/
Copyright	(c) 1995 Institute of Electronics, Information and Communication Engineers

PAPER Special Section on Digital Signal Processing

DSP Compiler for Matrix and Vector Expressions with Automatic Computational Ordering

Nobuhiko SUGINO†, *Member*, Seiji OHBI††, *Nonmember and* Akinori NISHIHARA†, *Member*

SUMMARY A description language for matrix and vector expressions and its compiler for DSPs are shown. They provide both a user-friendly programming environment and efficient codes. In order to increase throughput and to reduce amount of memory required for matrix and vector computation, several methods based on mathematical laws are introduced. A method to decide the matrix and vector storage location suitable for processing on DSP is also proposed.

key words: *DSP compiler, code optimization, matrix and vector computation*

1. Introduction

Digital signal processors (DSPs) have recently been remarkably upgraded and are now used in wide fields, such as communication, control and image processing [1]. Some workstations utilized DSPs for numerical computations [2]. In such computation in signal processing and other fields, procedures or algorithms are often expressed in terms of matrices and vectors, and DSPs are so fabricated to have excellent performance in matrix and vector computation. In spite of this attractive computational power of DSPs, their software development imposes laborious work and long time on programmers. This is because DSP programmers are required to write a realtime program or an efficient program (short in execution time) using primitive tools such as assembler or low level (machine level) languages. Therefore, practical development of DSP programs has been relied on some experts who have enough knowledge of the DSP architecture and tricky techniques.

To reduce the time and cost of DSP programming, helpful tools such as a high level language and its compiler are very important. Several DSP manufactures now supply compilers for high-level languages such as C or Ada [3]. Efficiency of program, however, is heavily dependent on the way of programming. Thus primitive languages and tools still play major roles in DSP programming, and programmers' load has not yet been reduced.

In Refs. [6] and [7], the user-friendly language to describe linear digital networks and DSP compilers are proposed. Since the compiler itself automatically determines the computational order in the program, efficiency of program is independent of programming styles, and users of this compiler can obtain efficient codes without laborious work. However, this compiler can handle only scalar data.

In order to derive efficient code for matrix and vector expressions, even expert programmers spend lots of time and labor, because large amount of memory must be managed and multiple nestings of loops in the program must carefully be considered. Moreover, it is much more difficult to write efficient programs for pipelined DSPs or so-called the second generation DSPs [1]. For such DSPs, programmers must take an extreme care for data access to avoid pipeline hazard. Also, their branch(jump) instructions, so-called delayed branching, works later than the following instructions, and cause difficulty in DSP programming.

This paper shows a DSP code generation method for matrix and vector expressions, which gives the beneficial compiler to DSP programmers. This method is applicable to both non-pipelined and pipelined DSPs. Furthermore, this method is modified to lead powerful programming tools for general purpose high level languages.

2. Description of Matrix and Vector

Matrix and vector computation programs written in high level languages such as C, Fortran, Pascal, etc. are different in their representations from the original mathematical expressions. Thus, programmers get difficulty in formulating, writing, and debugging the programs. In the compilers for such languages, optimizations after the code generation may improve the quality of the resultant codes. But these optimizations are highly dependent on the way of programming and are not always effective enough for the matrix and vector computation. Also, some attempts have been made to improve the existing high level languages. In Fortran 90 [8], some descriptions about array operations are improved so that writing a program for matrices and vectors computations is convenient. However efficiency of compiler generated codes is still dependent on programming style, and validity of computation of matrices and vectors must be investigated by programmers.

Manuscript received January 5, 1995.
Manuscript revised March 8, 1995.

†The authors are with the Faculty of Engineering, Tokyo Institute of Technology, Tokyo, 152 Japan.
††The author is with Sony Corporation, Tokyo, 108 Japan.

```
matprog seq_est(vector[10] znew; scalar y >
                vector[10] thetanew        );
{
        vector[10]        k;
        vector[10]        theta;
        matrix[10,10]     P, Pnew;
        thetanew = theta + k * ( y - trans(znew)*theta );     /* (1) */
        k        = P*znew / (1 + trans(znew)*P*znew);         /* (2) */
        Pnew     = (1 - k*trans(znew)) * P;                   /* (3) */
        P        = DLY(Pnew);
        theta    = DLY(thetanew);
}
```

Fig. 1 Example of input description.

In order to generate efficient code for the matrix and vector computation, a special language would be helpful. In this case, the descriptions for the matrix and vector expressions can be designed to be similar to mathematical formulas so that users can easily write, understand and debug the program. An optimization suitable for matrix and vector computation on DSPs can be expected in the procedure of compilation.

From such a point of view, we designed a new language to describe matrix and vector computation. For example, the following sequential least square method is written in Fig. 1, using the description proposed here.

$$\theta(t+1) = \theta(t) + \mathbf{k}(t+1)\{\mathbf{y}(t+1) - \mathbf{z}^{\mathbf{T}}(t+1)\theta(t)\} \tag{1}$$

$$\mathbf{k}(t+1) = \frac{P(t)\mathbf{z}(t+1)}{1 + \mathbf{z}^{\mathbf{T}}(t+1)P(t)\mathbf{z}(t+1)} \tag{2}$$

$$P(t+1) = \{1 - \mathbf{k}(t+1)\mathbf{z}^{\mathbf{T}}(t+1)\}P(t) \tag{3}$$

In the program list, $trans(arg)$ and $dly(arg)$ mean transpose and unit delay of arg, respectively.

The important feature of this description is that computation does not follow the order of sentences, so that the compiler decides a computational order. Moreover, type mismatches in operations of matrices and vectors can easily be detected by the compiler. On the way of the type detection process, constant number can be judged whether it is a matrix, a vector or a scalar data. For example, the constant number 1 in Eq. (3) is an identity matrix. Consequently, an identity matrix can be hidden in the constant number or the scalar variable and it simplifies the description.

3. Compiler

3.1 DSP Model

In this paper, a typical model DSP shown in Fig. 2 is assumed, which has a specific architecture for the efficient execution of multiply-add/subtract operations. Two operands of multiplication are simultaneously sent to the multiplier from each of separate memory blocks. The memory addresses are accessed by pointers, which can also be used for program controls such as branching conditions.

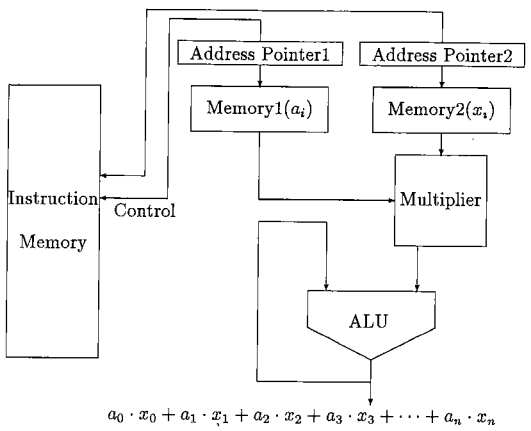


Fig. 2 The architecture model of the target DSP.

3.2 Outline of Compiler

The whole procedure of the matrix and vector compiler is shown in Fig. 3.

At first, the compiler reads the description of matrix and vector expressions. At this moment, the equations are represented by the computational trees for compiler internal use. An example of computational tree is shown in upper hand of Fig. 4. Second, the compiler examines validity of every operation, and useless or verbose operations are removed or unified. At the same time, constants and variables are identified as matrices, vectors or scalars from the context.

Then, after the optimization procedures based on some mathematical laws, the computational ordering of the expressions are determined with a heuristic search method.

Finally, code generation for a DSP follows the arrangement of matrix and vector data in the memory.

As the first part of the compiler is independent of target DSPs, slight modification only in the second part yields compilers for other target DSPs (right side in Fig. 3). Furthermore, the compiler can generate source programs in high-level languages such as C

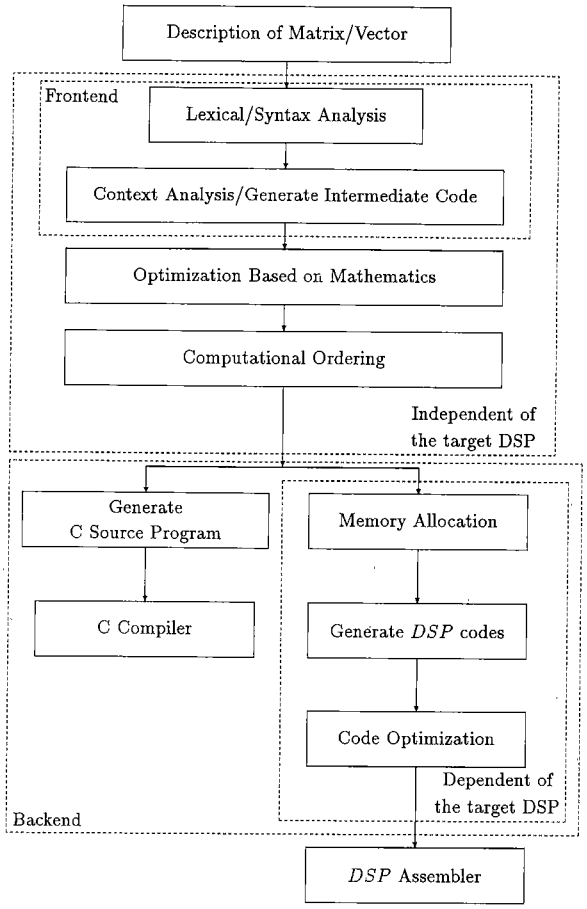


Fig. 3 Structure of the matrix and vector compiler.

in place of assembler language, as shown in the figure, which can be used as a helpful programming tool for general purpose compilers (left side in Fig. 3). This high-level language description allows one to confirm the given matrix and vector expressions on some personal computers or workstations.

3.3 Computational Ordering Method

There are precedence relations between scalar, vector or matrix variables in given expressions. Although these relations must be kept in DSP programs, they do not leads to a unique computational order and it is hard to find the optimal order. Therefore, a heuristic algorithm is applied to decide a computational order between variables.

In scalar computations, data are manipulated successively using a few registers. Thus, computational order with effective use of registers are desirable to derive an efficient program. On the other hand, the storage size of matrices and vectors data is much larger than that of scalars and they are mainly handled on memories; matrices and vectors data are loaded from memory and, after computation on a certain register, resultant

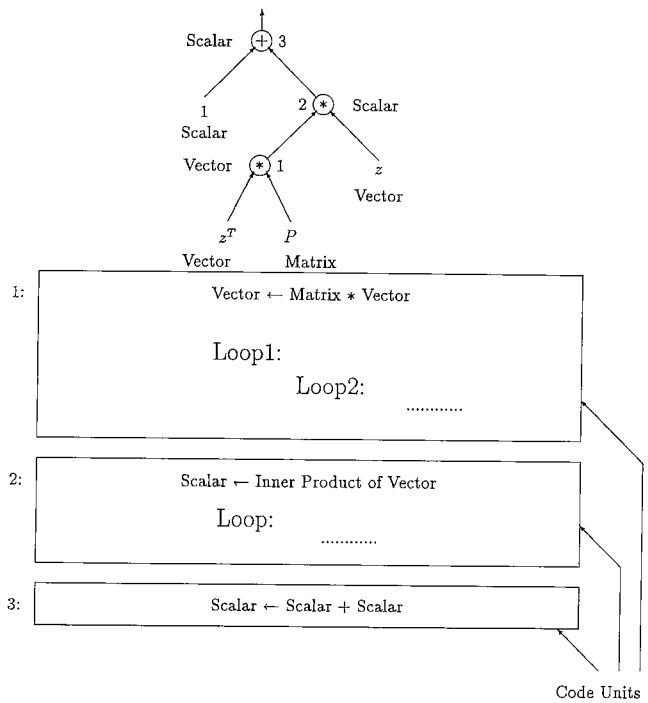


Fig. 4 An example of code generation.

data are stored back onto memory. Since in matrix and vector operation, registers are only used for temporarily preserving or computing data, computational ordering with effective use of registers like that for scalar data is no longer desired. Therefore, simple ordering method based on the 'depth-first-search' [6] is applied in this paper, and it gives satisfactory results. When the number of scalar operations is large and more effective ordering for scalar operations is required, the method in Ref. [7] is applicable and the method is also effective for the DSP which works under several stages of pipeline.

3.4 Code Generation

Along the determined computational order of expressions, all the operations are replaced by the corresponding sequence of instruction codes including loops inside, which is called a 'code unit' here. A simple example is shown in Fig. 4. A computational tree is shown in upper hand of the figure and, its computational order is given by the number labeled on each node. According to this order, code unit for multiplication between matrix and vector is generated as output codes, first. Then, code units of inner product of vectors and addition of scalars follow. Here, code units must be prepared in advance for all the basic operations such as addition, subtraction and multiplication of scalars, vectors and matrices.

Many code units are required according to the operation and its operands. For example, there are three different type of add operation; i.e. addition between

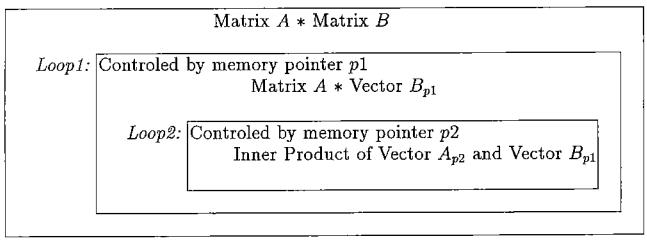


Fig. 5 An example of code unit for complicated operation.

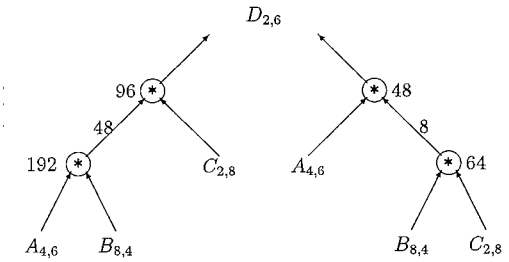


Fig. 6 Comparison of local ordering.

scalars, between vectors and between matrices, so that 3 code units are required. For the multiply operation, much more code units are required. If there are routines for these operations in library, they can be used as the code units. In case some operations are not included in a library, new code units must be prepared which requires large amount of time and labor.

Since every matrix is decomposed into column or row vectors and operations between matrices can be represented by the combination of vector operations, the number of the code units can be reduced. 'Basic code units' to be prepared for code generation are:

Basic code units

- Scalar Operations (`Add/Sub./Mult./Div.`)
- Add/Subtract Operation between Vectors (or Matrices)
- Multiplication of scalar with Vector (or Matrix)
- Inner Product between Vectors

Note that the code units of add/subtract operation for vectors can also handle that for matrices, because only one long loop rather than two short loops for row and column is enough for computation of all the elements in operation for matrices. Similarly, multiplication of a vector by a scalar and multiplication of a matrix by a scalar can utilize the common code unit by changing the loop length.

An example of other code units is shown in Fig. 5 for matrix A multiplied by matrix B, which can be represented by inner products of vectors and two loops.

Ordinary subroutines include codes for preserving and restoring register content, while code units in this method do not include them. Moreover, in this code generation method, transpositions can be combined with add, subtract or multiply operations, while, in ordinary subroutine calls, transposition and operation are handled separately. Therefore codes generated by this method are expected to have less overheads than the codes using subroutine calls especially when expressions include transposition of matrices and vectors.

3.5 Optimization

In this section, some optimization methods at procedure "Optimization method based on mathematics" in Fig. 3 are introduced.

3.5.1 Optimization Based on Associative Law

For scalar computation, there is an algorithm to generate local optimal codes [9]. This algorithm uses the associative and commutative properties of add and/or multiply operators, and the number of codes and/or the number of storage references are reduced. DIMPL compiler [7] also applies this algorithm in order to get efficient codes.

For matrix and vector computation, the associative properties of multiply operator can be used to minimize both the computational cost and required memory capacity. For an example, consider a cluster of matrix multiplication

$$A_{6\times4}B_{4\times8}C_{8\times2} \rightarrow D_{6\times2},$$

where the subscripts show the sizes of matrices. The associative law leads to two cases for local computational order given in Fig. 6. In the figure, the number labeled on nodes and branches indicate required computational cost and memory capacity, respectively, where computational cost is defined as the number of multiply-add/subtraction operations. The total numbers of computational cost and memory capacity are also listed in this figure. Consequently, the righthand computation requires less cost and memory than the lefthand one.

A method to derive the local optimal order is given by scanning all the operators in a given program for a cluster of multiplications and by finding the combination with least computational cost among all the possible combinations of multiplication. This method is quite similar to the algorithm in Ref. [9].

3.5.2 Special Combination of Operations

More efficient codes would be obtained by handling the special combination of operations as one operation and preparing the corresponding code units. For example, the term like $\mathbf{z}(t+1)^T P(t)\mathbf{z}(t+1)$ in Eq. (2) of Fig. 1 often appears in signal processing algorithms. Scanning all the operations in a given program for such a term and handling it as one operator, as shown in Fig. 7, can

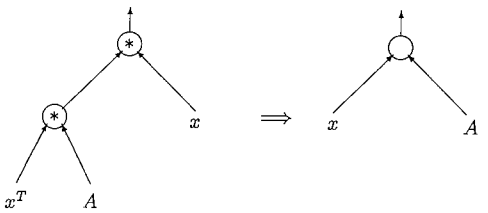


Fig. 7 Example of memory allocation.

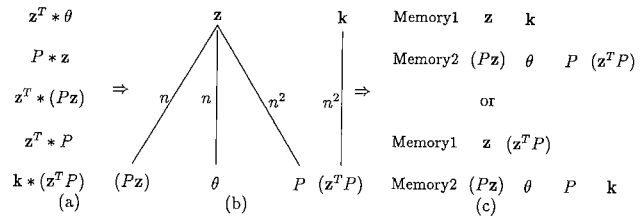


Fig. 8 Optimization by the special combination of operators.

improve the code unit itself and the whole program as well.

3.5.3 Term Expansion

Expanding terms in expressions sometimes saves computational cost and memory. For example, Eq. (3) can be writen as

$$P(t+1) = P(t) - \mathbf{k}(t+1)\mathbf{z^T}(t+1)P(t) \qquad (4)$$

Codes generated for this expression include no matrix–matrix multiplication, and require less cost and memory. However, it is difficult to get a general algorithm for such an expansion, the present version of compiler does not yet take this expansion into account.

3.5.4 Subexpression Coincidence

Generally, common subexpression coincidence saves lots of computational cost in spite of increase in storage to keep data for a short period. Especially, common subexpression of matrix and vector multiplication can greatly reduce computational cost with less increase in memory. In the example of the sequential least square algorithm, $\mathbf{z}^T(t+1)P(t)$ and $P(t)\mathbf{z}(t+1)$ are eliminated.

3.6 Memory Allocation

Since the DSP model assumed here consists of two pages of storage area and each of them is connected to multiplier, the location of matrix and vector data in storage pages gives a serious effect of generated codes. Especially, for multiplication, the execution steps of a bad allocated code are about 33% more than that of a good allocated code. A method to derive a memory allcation is presented below with an illustration of memory allocation for Fig. 1.

1. Extract a pair of matrix and vector multiplication operands (see Fig. 8 (a)).
2. Link every pair of the operands by a branch and label the branch by its computational cost (see Fig. 8 (b)). This graph is called an access graph.
3. Find the cut with the maximum total weight in the access graph and the bipartite graph is given.
4. Allocate disjointed sets of operands or nodes in the bipartite graph to distinct memory blocks (see Fig. 8 (c)).

Computational cost of matrix and vector multiplication is defined by the number of multiply operations or the number of multiply-add/subtract operations, and is given by

$$Matrix/Vector[l,n] * Matrix/Vector[m,l] \Rightarrow Cost = lmn,$$

where multiply-add/subtract operation is assumed to cost only one instruction cycle for DSPs. This cost also gives the estimation of a increase in program steps, when the two operands of multiplication are stored in the same memory block.

In order to find the cut with the maximum total weight, all the possible combinations of branches are examined in this article. However this method costs a plenty of time, as the number of operands increases, so that a heuristic algorithm introduced in Ref. [10] can be applicable. For scalar computation, the method to derive an optimal register and memory allocation is introduced in Ref. [11].

4. Results

Compiler is now developed for μ**PD77230** (NEC) and can generate efficient codes. Figures 9 and 10 show a part of the generated DSP codes and the C source programs for the example program in Fig. 1, respectively.

Table 1 compares the number of execution program steps, which are derived for the example in Fig. 1. The table shows this compiler generates efficient codes.

5. Conclusion

A DSP compiler for matrix and vector expressions is proposed, which saves the programming load and reduces the period for developing DSP programs. This compiler is helpful not only for the beginners but also for expert programmers, who can modify the output of this compiler in order to get further efficient codes. The proposed method can also be applied to general-purpose processors and high level languages. Combination of this compiler and DIMPL compiler will give

```
;
;       trans(vector) * matrix * vector -> scalar
;
        clr  wr1                                              ;
        ldi  bp1, N                                           ;
Loop1:
        ldi  ix1, Pn1                                         ;
                        mov ar  ,bp1            stix1         ; P[] address Spc.
        ldi  bp1, minusN                                      ;
;
        ldi  ix0, ZNEW1                                       ; Z address
        ldi  bp0, N                                           ;
;
;               trans(vector) * vector -> scalar
;
        clr  wr2                        spcbi0  spcbi1        ;
Loop2:
                        mov klr1,ram0   decbp0  stix1         ;
        jnzbp0 Loop2                                          ;
        addf wr2,m                                            ;
        norm wr2                                              ;
;
;               scalar * scalar -> scalar
;
                        mov bp1 ,ar
                        mov bp0 ,bp1
                        mov lkr0,wr2            decbp1        ;
;
;               scalar + scalar -> scalar
;
        addf wr1,m                                            ;
        jnzbp1 Loop1                                          ;
        nop                                                   ;
Loop3:
                        mov klr1,ram0   decbp0  decbp1        ;
        jnzbp0 Loop3                                          ;
        addf wr1,m                                            ;
        norm wr1                                              ;
;
;       scalar + scalar -> scalar
;
        ldi  tr , 400000H                                     ;
        ldi  tre, 000001H                                     ; tr  <- 1.0
        addf wr1, ib    mov non ,tr                           ;
        norm wr1
;
;       scalar / scalar -> scalar
;
         ⋮
```

Fig. 9 Example of output DSP assembler code.

Table 1 Comparision of execution steps of the codes.

	Execution Steps (Order N)	Difference
without Optimization	$3N^3 + 25N^2 + 58N + 55$	———
with Optimization	$15N^2 + 44N + 34$	$-3N^3 - 10N^2 - 14N - 21$
Hand–Assembled	$12N^2 + 28N + 25$	$-3N^3 - 13N^2 - 30N - 30$

better programming environment for signal processing. Further optimization techniques and the more detailed computational ordering methods for scalar, vector and matrix must be investigated.

Acknowledgement

The authors are grateful to Prof. N. Fujii and Assoc. Prof. S. Takagi of Tokyo Institute of Technology, for their valuable discussions, and thanks to Mr. H. Ohotomo and members of NEC Corporation for their support.

References

[1] Shear, D., "EDN's DSP Benchmarks," *EDN*, pp.126–148, Sep. 1988.
[2] Thompson, T. and Baran, N., "The NeXT Computer," *BYTE*, pp.158–175, Nov. 1988.
[3] Leibson, S.H., "DSP development software," *EDN*, pp.156–165, Nov. 1990.
[4] NEC Corporation, "μPD77230 Users' manual," 1986.
[5] Sugino, N., Ohbi, S. and Nishihara, A., "DSP Compiler for Matrix/Vector Expressions with Automatic Computational Ordering," in *Proc. ECCTD '91*, pp.669–678, Sep. 1991.
[6] Sakamoto, H., Watanabe, E. and Nishihara, A., "Code Optimization of digital network for signal processors," in *Proc. of ISCAS*, pp.1403–1406, Jun. 1985.
[7] Sugino, N., Ohbi, S. and Nishihara, A., "Computational ordering of digital network under the pipeline constraints and its application to compiler for DSPs," in *Proc. ECCTD '89*, pp.395–399, Sep. 1989.
[8] Metcalf, M. and Reid, J., *Fortran 90 Explained*, Oxford University Press, 1990.
[9] Sethi, R. and Ullman, J.D., "The generation of optimal code for arithmetic expressions," *Journal of ACM*, vol.17, no.4, pp.715–728, Oct. 1970.
[10] Goemans, M.X. and Williamson, D.P., ".878-approxima-

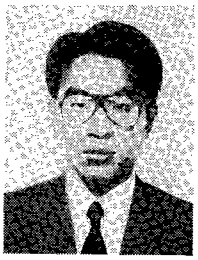
```
main()
{
    float   znew[10] ;
    float   y ;
    float   thetanew[10] ;
    float   knew[10] ;
    float   theta[10] ;
    float   P[10][10] ;
    float   Pnew[10][10] ;
/*
    float   _V1[10] ;
*/
#define _V1 knew
    float   _S2 ;
    float   _V3[10] ;
    float   _S4 ;
    float   _S5 ;
/*                                          */
/*  trans(vector) * matrix * vector-> scalar */
/*                                          */
    int     _L7 ;
    int     _L8 ;
    float   _T9 ;
    _S5 = 0.0 ;
    for (_L7=0;_L7<10;_L7++) {
        _T9 = 0.0 ;
        /*                                  */
        /*  trans(vector) * vector -> scalar */
        /*                                  */
        for (_L8=0;_L8<10;_L8++) {
            _T9 += znew[_L8] * P[_L7][_L8] ;
        }
        _S5 += _T9 * znew[_L7] ;
    }
/*                                      */
/*  scalar + - * / scalar -> scalar     */
/*                                      */
    _S4 = 1 + _S5 ;
        :
}
```

Fig. 10 Result of C-source generation in Fig. 1.

tion algorithms for MAX CUT and MAX 2SAT," in *Proc. 26 Ann. ACM Symposium on Theory and Comp.*, pp.422–431, 1994.

[11] Ohbi, S., Sugino, N. and Nishihara, A., "Automatic generation of DSP codes with effective use of memory/registers," *Proc. of 4th Symposium on Digital Signal Processing*, pp.37–41, Dec. 1989.

Nobuhiko Sugino was born in Yokkaichi, Mie, Japan on November 19, 1964. He received B.E., M.E. and Dr. Eng. degrees in physical electronics from Tokyo Institute of Technology in 1986, 1989 and 1992, respectively. Since 1992, he has been with Tokyo Institute of Technology, where he is now a Research Associate of Department of Physical Electronics, Faculty of Engineering. His main research interests are in hardware and software for digital signal processing, especially in software development tools for Digital Signal Processors. Dr. Sugino is a member of IEEE.



Seiji Ohbi was born in Zentsuji, Kagawa on September 28, 1966. He received B.E. and M.E. degrees in electronics from Tokyo Institute of Technology in 1989 and 1991. Since 1991 he has been with Sony Corporation. His main research interests are in digital signal processing systems.

Akinori Nishihara was born in Fukuoka, Japan on February 26, 1951. He received B.E., M.E. and Dr. Eng. degrees in electronics from Tokyo Institute of Technology in 1973, 1975 and 1978, respectively. Since 1978 he has been with Tokyo Institute of Technology, where he is now Associate Professor of Department of Physical Electronics, Faculty of Engineering. His main research interests are in filter design, 1D and multi-D signal processing, and modern applications of classical circuit theory. From 1990 to 1994 he served as an Associate Editor of the IEICE Trans. Fundamentals. Dr. Nishihara is a member of IEEE, EURASIP and ECS.