論文 / 著書情報
Article / Book Information

Title	Code Optimization Technique for Indirect Addressing DSPs with Consideration in Local Computational Order and Memory Allocation
Authors	NOBUHIKO SUGINO, Akinori Nishihara
出典 / Citation	IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E84-A, No. 8, pp. 1960-1968
発行日 / Pub. date	2001, 8
URL	http://search.ieice.org/
権利情報 / Copyright	本著作物の著作権は電子情報通信学会に帰属します。 Copyright (c) 2001 Institute of Electronics, Information and Communication Engineers.

IEICE TRANS. FUNDAMENTALS, VOL.E84–A, NO.8 AUGUST 2001

PAPER *Special Section on Digital Signal Processing*

Code Optimization Technique for Indirect Addressing DSPs with Consideration in Local Computational Order and Memory Allocation

Nobuhiko SUGINO[†a)] ***and*** **Akinori NISHIHARA**[††], ***Regular Members***

SUMMARY Digital signal processors (DSPs) usually employ indirect addressing using address registers (ARs) to indicate their memory addresses, which often introduces overhead codes in AR updates for next memory accesses. Reduction of such overhead code is one of the important issues in automatic generation of highly-efficient DSP codes. In this paper, a new automatic address allocation method incorpolated with computational order rearrangement at local commutative parts is proposed. The method formulates a given memory access sequence by a graph representation, where several strategies to handle freedom in memory access orders at the computational commutative parts are introduced and examined. A compiler scheme is also extended such that computational order at the commutative parts is rearranged according to the derived memory allocation. The proposed methods are applied to an existing DSP compiler for μPD77230(NEC), and codes generated for several examples are compared with memory allocations by the conventional methods.

key words: *DSP compiler, indirect memory addressing, memory allocation, code optimization*

1. Introduction

Digital signal processors (DSPs) are now widely used to implement various real-time applications, such as filters, FFTs, CODECs, MODEMs, various control systems, etc. The programming of such real-time algorithms on DSPs, however, is nontrivial, cost and time consuming work; programmers are often required to write an efficient (short in execution time) program/code for real-time use, and hence they need to know details in both processor architectures and algorithms.

In order to reduce such heavy programming load, software tools such as high-level languages and its compilers are very important. Many programming tools and compilers are presented by venders and researchers [1]–[5]. In Ref. [4], [5] DIMPL (Digital network IMPlementation Language) and its compiler have been proposed. In these compilers, an efficient program benefits from effective use of registers. Efficient access of memory in a program, however, is as important as effective use of registers in order to derive an efficient program.

In many DSPs, memory content is accessed indirectly through address registers (ARs) [13]–[15]. Although AR must be updated before memory access, simple AR update operations such as increment or decrement (AR ± 1) operations can be concurrently performed in one instruction cycle besides data operations such as addition, subtraction and date movement. When the code requires to read a datum from memory or to write a datum into memory, an AR must be updated before the corresponding instruction cycle. If AR update amount is beyond the totals of simple AR operations, additional one instruction cycle is required for substituting memory address for AR (immediate AR load operation), which becomes overhead in a program. In order to derive an efficient program, reduction in the number of such immediate AR loads is very important. An appropriate memory address allocation can utilize AR ± 1 operations as much as possible and can save immediate AR loads. The problem to find such an efficient memory address allocation is called as "memory allocation issue." Besides, DSPs usually have multiple ARs and utilization of these ARs can improve memory access. The problem to find an appropriate memory access assignment into multiple ARs is called as "AR assignment issue."

In Refs. [6]–[12], for a DSP with these indirect memory addressing operations, methods to derive a memory access pattern with less overhead codes have been studied. These methods model program variables and AR operations by an access graph (AG). Liao [6] proposed a heuristic method to search a maximum-weighted Hamiltonian path in the AG. Leupers [7] introduced AR ± 1 and $\pm k$ operations, and gave a method to utilize additional registers. Wess [8] formulated memory address allocation problem by a matrix form and employed various numerical optimization techniques such like simulated annealings (SA) or genetic algorithms (GA).

These methods provide excellent memory access with less AR loads within a separated memory access optimization issue of overall code generation scheme. For further reduction in overhead codes, however, ef-

Manuscript received December 1, 2000.
Manuscript revised March 7, 2001.

[†]The author is with the Department of Advanced Applied Electronics, Interdisciplinary Graduate School of Science and Engineering, Tokyo Institute of Technology, Yokohama-shi, 226-8502 Japan.
[††]The author is with Center for Research and Development of Educational Technology (CRADLE), Tokyo Institute of Technology, Tokyo, 152-8552 Japan.
a) E-mail: sugino@ae.titech.ac.jp

fective combination of memory allocation method and other code generation and/or optimization techniques is very important. In the ordinary code generation scheme, a memory access sequence is given in accordance with the computational order of the program, where we still have freedom of choice in appropriate local computational orders. This paper presents a new memory address allocation method based on the graph representation, where local computational orders in a given program is utilized. In order to apply the method to the existing compilers, a new compiler scheme is also shown. Note that the proposed memory allocation method is a heuristic method, so that it gives an efficient memory allocation with less computational cost.

2. Memory Addressing

When reading a datum from memory or writing a datum into memory (in general, access of a datum in memory), memory address corresponding to the datum must be pointed by some ways, which are called as memory addressing modes. In general purpose processors, several memory addressing modes such as immediate addressing, indirect addressing, indexed addressing mode, and so on, are provided, so that programmers easily implement various data structures. These addressing modes are well utilized in compilers for high-level languages such like C.

As for DSPs, memory addressing modes are rather poor than general purpose processors in return for high performance in computation and low-chip cost. Almost all the DSPs have indirect addressing modes [13]–[15]. They have so called address registers (ARs) to point the memory addresses to be accessed. In this paper, the following simple indirect addressing mode is assumed [10].

Memory addressing mode of a DSP model

- Memory is accessed only by AR indirect addressing.
- AR can be post-modified (or auto-modified) by one (AR±1),
  $$\mathrm{AR} \leftarrow \mathrm{AR} \pm 1$$
  besides arithmetic operation or data movement at one instruction cycle (even though memory is not accessed at that cycle).
- Immediate AR load(hereafter AR load)
  $$\mathrm{AR} \rightarrow nnn (nnn\text{:Destination Address})$$
  costs one instruction cycle only by itself.

Some of DSPs [13], [14] have more sophisticated AR update operations such as AR±2, modulo AR±1 update operations, multiple AR operations and etc., and the corresponding memory allocation methods are considered in Refs. [11], [12]. In this paper, the simple addressing model is employed. The reasons are (1) to simplify the problem and hence algorithm, and (2) to measure contribution of local computational order over reduction in memory access overheads. The methods for DSPs with sophisticated AR update operations are investigated in future.

Let us think of memory address and AR operations with a simple example, signal flow graph of a second order IIR filter shown in Fig. 1. After an appropriate computational ordering, intermediate codes are derived as shown in left hand side of Table 1. A denotes the temporary variable which keeps the intermediate result in the register. The left hand side in Table 1 shows memory access address sequences and AR operations for a given computational order. In this case, by use of appropriate memory allocation method [9], [10], 3 immediate AR loads are required. They are marked by "LDs" in the table.

In the intermediate code, however, we may choose

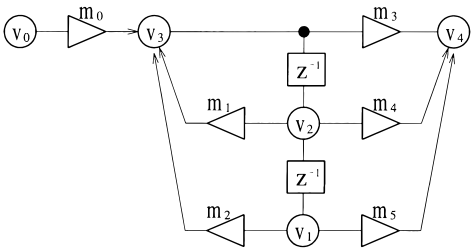


Fig. 1 2nd order IIR filter.

Table 1 Intermediate code for Fig. 1 and AR update.

Cycle	Intermediate Code	Memory Access	Memory Allocation and AR Operation	Intermediate Code	Memory Access	Memory Allocation and AR Operation
1	$A \leftarrow v_0 \times m_0$	v_0	0) $+1$	$A \leftarrow v_0 \times m_0$	v_0	0) LD
2	$A \leftarrow A + v_1 \times m_2$	v_1	1) $+1$	$A \leftarrow A + v_2 \times m_1$	v_2	3) -1
3	$A \leftarrow A + v_2 \times m_1$	v_2	2) $+1$	$A \leftarrow A + v_1 \times m_2$	v_1	2) -1
4	$v_3 \leftarrow A$	v_3	3)	$v_3 \leftarrow A$	v_3	1)
5	$A \leftarrow A \times m_3$	None	-2	$A \leftarrow A \times m_3$	None	$+1$
6	$A \leftarrow A + v_1 \times m_5$	v_1	1) $+1$	$A \leftarrow A + v_1 \times m_5$	v_1	2) $+1$
7	$A \leftarrow A + v_2 \times m_4$	v_2	2) LD	$A \leftarrow A + v_2 \times m_4$	v_2	3) $+1$
8	$v_4 \leftarrow A$	v_4	4) LD	$v_4 \leftarrow A$	v_4	4) -1
9	$A \leftarrow v_2$	v_2	2) -1	$A \leftarrow v_2$	v_2	3) -1
10	$v_1 \leftarrow A$	v_1	1) LD	$v_1 \leftarrow A$	v_1	2) -1
11	$A \leftarrow v_3$	v_3	3) -1	$A \leftarrow v_3$	v_3	1) LD
12	$v_2 \leftarrow A$	v_2	2	$v_2 \leftarrow A$	v_2	3
# of AR loads			3	# of AR loads		2

different computational order at codes 1–3 and 5–7. The right hand side in Table 1 shows memory access address sequences and AR operations for an appropriate computational order. In this case, only 2 AR loads are needed, and 1 AR load is saved. As the above example shows, if freedom in local computational orders is considered at memory allocation, memory access with further less overhead codes is expected.

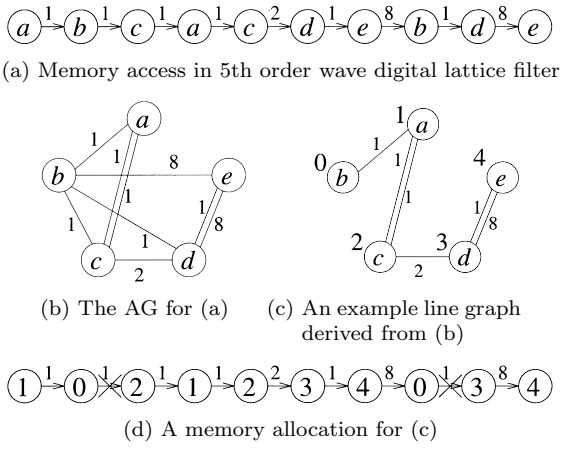
3. Memory Allocation Method

3.1 Access Graph and ALOMA [9], [10]

The memory allocation method in this article is based on ALOMA (Address LOad Minimization Algorithm) [9], [10]. This algorithm inputs a memory access sequence, which is defined directly and uniquely from a given program code. An example memory access sequence is shown in Fig. 2(a). The algorithm determines memory allocation of program variables, and AR update operations required between memory accesses.

In the algorithm, a given memory access sequence is translated into a graph notation, namely an access graph (abbreviated as AG in the followings), where each vertex and edge denote a variable to be accessed and a constraint on memory allocation, respectively. AG notation is quite useful to represent constraints over memory allocation issue under indirect memory addressing mode with automatic address update scheme. Although there exist other notations like matrix representation form [8], one may rewrite these notation into the AG notation by some manners. Note that even though the corresponding access sequence is directional, AG edges are not. The reason is that AR can be always updated to both $+1$ and -1 directions, and that the memory address allocation reverse to a derived memory allocation gives the same result in terms of number of AR loads. After an appropriate memory allocation, memory addresses are assigned for all vertices, and hence, we can find required AR update amount or a set of AR operations at each AG edge. If this required amount does not meet for realizable AR update amount (i.e. # of AR operations allowed at that edge), then we need immediate AR load operation, which becomes an overhead code.

For example, the AG of the memory access sequence in Fig. 2(a) is shown in Fig. 2(b), where the numbers labeled on each arrow in Fig. 2(a) and each edge in Fig. 2(b) denote the lengths, which correspond to the number of AR operations allowed between these two memory accesses. Therefore these numbers represent the maximum AR update ranges by use of AR$\pm$1, and they are called as "access lengths" or just lengths. Then, the algorithm removes appropriate edges from the given AG in order to remove forks and cycles in the AG, and AG is transformed into a line graph(s) or a chain sequence(s). For an example AG in Fig. 2(b), if we remove the edges b–e, b–c, and b–d, then we have a line graph as in Fig. 2(c). Finally, along with the line graph(s), an address location for every variable is decided. In return, immediate AR loads are sometimes required at the memory access corresponding to the removed edges according to their length, since removal of edges between two vertices (or variables), namely u and v, means that we lose the constraint to allocate u and v in the adjacent memory addresses. The exact number of AR loads for a given memory access sequence is not apparent until memory addresses are decided. For the above example, we have the memory access sequence with memory allocation as shown in Fig. 2(d). In this example, we need two AR loads at removed edges b–c and b–d. Meanwhile, AR update at edge (b, e) can be realized within its permissible AR update range.



(a) Memory access in 5th order wave digital lattice filter

(b) The AG for (a)

(c) An example line graph derived from (b)

(d) A memory allocation for (c)

Fig. 2 An example AG.

3.2 Local Computational Order

In a given program, there exists a local (consecutive) code part in which we may freely choose its computational order without violating the intended computation. Such a part consists of commutative operators, so that it is called as commutative code block in this paper. An example commutative code block in intermediate code level is shown in Fig. 3. In the figure, the operands 'a,' 'b' and 'c' are program variables. They are stored in memory space, while the other operands 'r_1' and 'r_2' are kept in arithmetic registers. We may freely compute this code block in various order such like $r_1 \rightarrow a \rightarrow r_2 \rightarrow b \rightarrow c$, $r_1 \rightarrow b \rightarrow a \rightarrow r_2 \rightarrow c$ and so

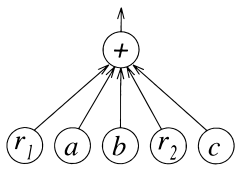


Fig. 3 Example of a commutative code block.

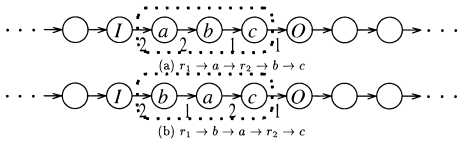


Fig. 4 Example commutative subsequences.

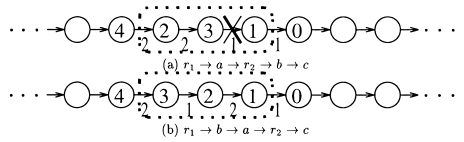


Fig. 5 Memory allocation examples for Fig. 4 (address allocation $a : 2$, $b : 3$, $c : 1$).

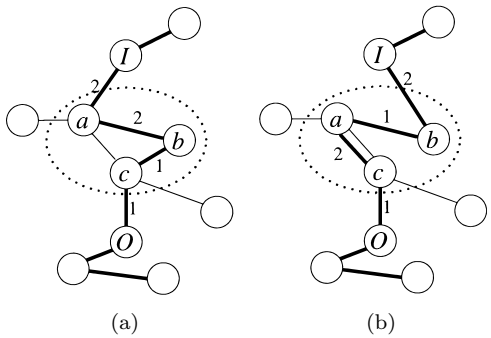


Fig. 6 AGs for Fig. 4.

on.

According to computational orders at commutative blocks, we then have various versions of access sequences. For example, memory access subsequence of Fig. 3 in the order of $r_1 \to a \to r_2 \to b \to c$ is given as Fig. 4(a). When we take another computational order $r_1 \to b \to a \to r_2 \to c$, then we have the corresponding memory access subsequence in Fig. 4(b). For these various access sequences, derived memory allocations differ each other. For example, if variables a, b, and c are allocated to addresses 2, 3, and 1, respectively in the above example, as results shown in Fig. 5, we need 1 AR load for (a), but no AR load for (b).

Since a memory access sequence is derived from a program code, computational order for a given program is already decided beforehand, and so as for commutative code blocks. In the compiler in [4], [5], [9], [10], computational orders for commutative code blocks are just tentatively decided. This is the usual case for other compilers.

If we examine all the possible access sequences by exhaustive manner, we have the most efficient memory allocation together with the computational orders at commutative blocks. Evidently, however, it is not practical. When the numbers of commutative code blocks and total program variables in commutative blocks increase, the number of access sequence to be examined increase enormously. For example, wave filter of 17th order in Table 2, we have 8 commutative blocks with 16 program variables. By the rough estimation, it takes about 1–2 days to complete $2^8 = 256$ exhaustive trials under the Pentium III 800 MHz personal computer or Ultra-sparc workstations, where it takes several minutes for memory allocation method. In fact, there exist instructions, which do not require any accesses for program variables in memory space, such like arithmetic instructions performed on registers r_1 and r_2 in the above examples. In such cases, the number of trials increases rapidly. Therefore, a heuristic method based on ALOMA is employed as described in the followings.

3.3 Commutative Subsequence and AG

Since we have to consider a large number of versions of memory access sequences in memory allocation procedure, as mentioned in the last subsection, we then have their corresponding AGs. For example, AGs for subsequences in Figs. 4(a) and (b) are given in Figs. 6(a) and (b), respectively. In this paper, a memory access subsequence at a commutative code block is called as a commutative subsequence. In the Fig. 4, commutative subsequences are covered by dotted curves. The corresponding AG edges are shown by the thick lines in Fig. 6. It should be noted that there usually exist AG edges for non-commutative subsequences, which are depicted by the thin lines in the figure for illustration.

For the simplicity of a memory allocation algorithm, it is quite benefiticial to introduce an AG, which unifies several versions of memory access sequences over the commutative code block. Moreover, a new weighting function is employed at commutative code blocks, so that the memory allocation method utilizes freedom in local computational order at commutative code blocks by memory accesses out of their subsequences. Therefore, memory allocation constraints for commutative subsequences are less significantly considered as AG edge weights, when a given access sequence is translated into AG. After the memory allocation, an appropriate computational reordering procedure for commutative code blocks must follow in order to utilize the derived memory allocation without introducing additional AR loads.

If we give too loose constraints for commutative subsequences, program variables in the subsequences are allocated to the scattered addresses in memory space. Thus, it causes significant increase in the number of AR loads. On the contrary, too tight constraints mean almost same constraints used in the previous method ALOMA, and give the same result in AR load reduction. In this paper, the 9 different strategies for the translation of commutative subsequences into an AG are experimentally examined.

Here, several new terms are introduced for expla-

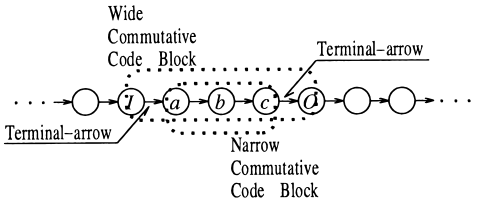


Fig. 7 Narrow and wide commutative subsequences.

nation of strategies. In Fig. 4, vertices marked *'I'* and *'O'* denote the variables accessed previous and next to the subsequence, respectively. They represent the entry and the exit of the subsequence. Since we cannot change the access order of I and O in each subsequence, the first and the last arrows are differently handled from the other arrows. These entry and exit arrows are refered as terminal-arrows of the commutative subsequence, while the other arrows are refered as the narrow commutative subsequence. An overall commutative subsequence, which includes the terminal arrow is called as the wide commutative subsequence. These terms are illustrated in Fig. 7. Note that a wide commutative subsequence includes I and O, while a commutative subsequence or a narrow commutative subsequence do not include the vertices I and O. Code blocks corresponding to wide and narrow commutative subsequences are refered as wide and narrow commutative code blocks, respectively.

By use of these terms, the 9 strategies are the followings. Corresponding AGs for the above memory access example are illustrated in Fig. 8.

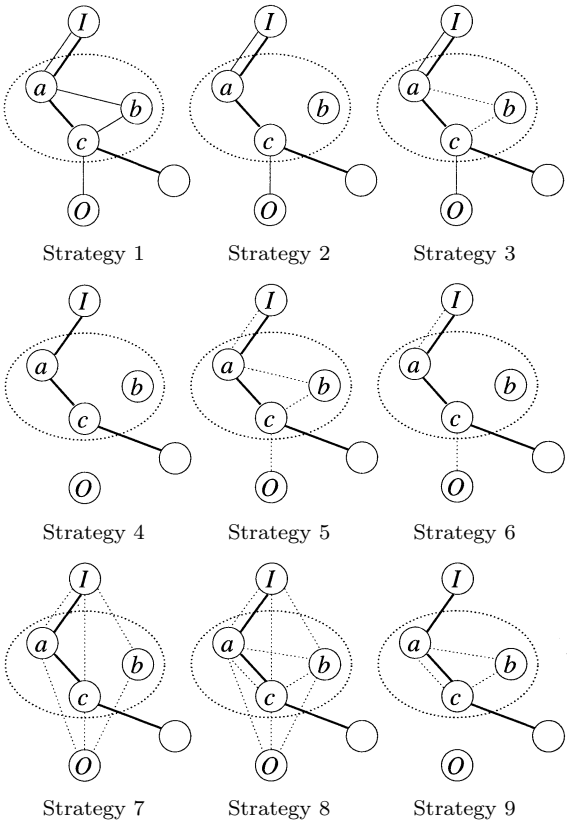


Fig. 8 Strategies for a commutative subsequence.

Strategies for a commutative subsequence

1. Use the original AG (Strategy 1 of Fig. 8).
2. Ignore AG edges at every narrow commutative subsequence (Strategy 2 of Fig. 8). AG edges which correspond to terminal-arrows of the subsequence are remain as the original AG. This strategy means that the memory allocation of the variables in the narrow commutative subsequence is determined by the constraints out of this subsequence. Then, reduction in AR loads is expected by computational rearrangement for narrow commutative code blocks.
3. Slightly consider AG edges at every narrow commutative subsequence by means of small weighting factors as the dotted lines in Strategy 3 of Fig. 8 shows. AG edges which correspond to terminal-arrows of the subsequence are remain as the original AG. This strategy means that the original memory access orders at narrow commutative subsequences are less significantly considered in the memory allocation procedure, so that the memory allocation of the variables in the narrow commutative subsequence is almost determined by the constraints out of this subsequence.
4. Ignore AG edges within every wide commutative subsequence (Strategy 4 of Fig. 8). This strategy means that the memory allocation of the variables in the wide commutative subsequence is determined by the constraints out of the subsequence. Then, reduction in AR loads is expected by computational rearrangement within wide commutative code blocks.
5. Slightly consider AG edges for every wide commutative subsequence by means of small weighting factors as the dotted lines in Strategy 5 of Fig. 8. This strategy means that the original memory access orders at wide commutative subsequences are less significantly considered in the memory allocation procedure, so that the memory allocation of the variables in the commutative subsequence is almost determined by the constraints out of the wide subsequence.
6. Ignore AG edges for every narrow commutative subsequence, but slightly consider AG edges for terminal-arrows of the commutative subsequence by means of small weighting factors as the dotted lines in Strategy 6 of Fig. 8. This takes a middle strategy between the strategies 2 and 4 in handling terminal-arrows of commutative subsequences. The AR edges for terminal-arrows are less significantly considered than those in strategy

2, but more significantly considered than those in strategy 4.

7. Ignore AG edges for every narrow commutative subsequence. AG edges from *'I'* to the subsequence and from the subsequence to *'O'* are slightly considered.
   Slightly consider all the possible memory access orders within a subsequence at every commutative block. Consequently, this forms a clique graph with small weighting factors for variables included in the subsequence as shown by dotted lines in Strategy 7 of Fig. 8. This strategy is the extended version from the strategy 6, and it gives rather tight constraints between I (O) the variables in commutative subsequences, so that I (O) is expected to be allocated in the addresses adjacent to one of the variables in commutative subsequences. Note that pseudo-AG edge length between I (O) and each variable in the commutative subsequence is assumed to be 1.
8. Slightly consider all the possible memory access orders for every wide commutative subsequence by means of introducing pseudo-AG edges with small weighting factors between all the variables in the subsequence (including I and O). Consequently, this forms a clique graph for variables in the wide subsequence as shown by dotted lines in Strategy 8 of Fig. 8. This strategy gives rather tight constraints arround the variables in wide commutative subsequences, so that these variables are expected to be allocated in the addresses adjacent to one another. Note that pseudo-AG edge length between I (O) and each variable in the commutative subsequence is assumed to be 1.
9. Slightly consider all the possible memory access orders for every narrow commutative subsequence by means of introducing pseudo-AG edges with small weighting factors between all the variables in the subsequence (without I and O). Ignore the AG edges for terminal-arrows of the subsequence. Consequently, this forms a clique graph for variables in the narrow subsequence as shown by dotted lines in Strategy 9 of Fig. 8. This strategy gives rather tight constraints arround the variables in narrow commutative subsequences, so that these variables are expected to be allocated in the addresses adjacent to one another.

In some of the above strategies, original weight for edge e in a commutative sequence is replaced by quantity w_e. For such small weighting factor w_e in strategies 3, and 5–9, the following 2 manners are examined in this paper.

1. Absolute manner: $w_e = w_0$,
   where w_0 denotes a uniform small weight. Giving such a uniform weight for every edge means that we ignore its length in the memory allocation procedure. In this paper, the uniform weight $w_0 = 1/N$ is employed, where N denotes the number of variables in an overall sequence. The number $1/N$ implies a number less than the average edge length in a given sequence.
2. Relative manner: $w_e = w_k w(e)$,
   where w_k denotes a small weighting factor, and where $w(e)$ denote the original weight on edge e. Using the original edge weight multiplied by a small number w_k means that its length is slightly considered in the memory allocation procedure. In this paper, $1/N$ is used as w_k in order to give obvious distinction between the original and the compensated edge weights.

Strategies with larger or smaller w_0 or w_k are also examined for several example sequences for the comparison. In such over or under constrained cases, better memory allocations are hardly derived.

3.4 Outline of a New Memory Allocation Method

The extended ALOMA for commutative code blocks is called ALOMA-LCO (with consideration in Local Computational Order). The outline of ALOMA-LCO is given as the followings.

ALOMA-LCO

1. Translate a given memory access sequence into the AG by use of one strategy among the strateties mentioned above. Let the derived AG as the initial graph G for the following procedures.
2. Remove edges with length $l \geq |G| - 1$, where $|G|$ denotes the number of variables in G.
3. In the following procedures, edges between two vertices are handled as an edge group. Let two vertices u and v, and the edge group between u and v is represented as (u, v).
   For each edge group (u, v) in G, evaluate the cost function benefit(u, v) [9], [10],

$$\text{benefit}(u,v) = \frac{\text{fork}(u)+\text{fork}(v)+\text{cycle}(u,v)}{w(u,v)}. \tag{1}$$

   For a vertex v, fork(v) [9], [10] means its fork count, and is defined as

$$\text{fork}(v) = \max\{\deg(v) - 2, \quad 0\}, \tag{2}$$

   where $\deg(v)$ denotes the order of vertex v.
   For an edge group (u, v), cycle(u, v) [9], [10] means its cycle flag and is defined by

$$\text{cycle}(u,v) = \begin{cases} 1, & \text{edge group } (u,v) \text{ is} \\ & \text{included in a cycle.} \\ 0, & \text{edge group } (u,v) \text{ is} \\ & \text{not included in any cycles.} \end{cases} \tag{3}$$

In the denominator of the cost function Eq. (1), the weighting function $w(u, v)$ is given by

$$w(u,v) = \sum_{e=\text{edge in } (u,v)} w(e), \tag{4}$$

where weight $w(e)$ for edge e is given by

$$w(e) = \begin{cases} w_e \\ \quad \text{(for } e \text{ in a commutative sequence)} \\ \dfrac{(N-l_e)(N-l_e-1)}{(N-1)(N-2)} \quad \text{(otherwise)} \end{cases} \tag{5}$$

In the above equation, N denotes the number of variables in an overall sequence, and l_e denotes the edge length of edge e. When edge e belongs to a commutative subsequence, its weight is replaced by w_e according to the strategy as mentioned in the last subsection. When edge e does not belong to a commutative subsequence, $w(e)$ estimates the the number of AR load at the edge of l length, when this edge is removed in the graph linearization procedure [9], [10].

Totally, the cost function benefit(u, v) estimates how many forks and cycles in G are removed at the cost of AR loads for the removal edges (u, v). Although benefit gives less AR load estimations for the commutative subsequences by the compensation, they are expected to be recovered after the appropriate computational rearrangement, which is mentioned below.

4. Find an edge group with the maximum benefit and remove it from G .
5. Repeat 3–4, until G becomes a line graph or line graphs.
6. For the vertices in the derived line graph(s), memory addresses are decided.

4. Compiler

The above proposed memory allocation method is incorporated with the DIMPL compiler [5]. The diagram of this compiler is shown in Fig. 9. The compiler first reads the filter network description, and decides a computational order. According to the decided order, the compiler generates a DSP assembler code together with a memory access sequence. Then, the new memory allocation method is applied for this sequence. Note that the sequence is extended to mark subsequences of commutative code blocks.

In the original DIMPL compiler, the computational order has already been decided before memory allocation, so that the computational orders at the commutative code blocks must be rearranged in the following procedures. Therefore, the new compiler feeds

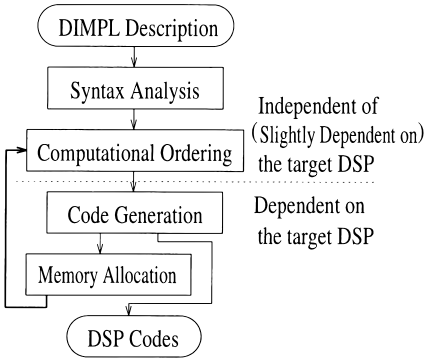


Fig. 9 Diagram of the proposed compiler.

the derived memory allocation back into the computationl ordering part and settles orders at commutative code blocks. Note that the memory access length at the commutative subsequence is the very important issue in this rearrangement procedure. When the memory access length at a commutative sequence does not meet, additional AR loads are required.

The proposed memory allocation method is easily applicable to other compilers, because its input consists only of an access sequence of program variables and additional marks on commutative code blocks. Additionally, however, the pre-procedure to detect commutative code blocks and the post-procedure to rearrange computational order at commutative code blocks are required.

5. Results

The proposed method with strategies is applied to a compiler for μPD77230 (NEC), and codes are generated for several filter examples. Since these filter examples include rather complicated memory access sequences and they include several number of commutative blocks as well, they are appropriate for the comparison.

Table 2 shows the comparison of the generated codes by the proposed method and the existing method [10] in terms of the number of AR loads. In the same table, the numbers of their commutative code blocks and total variables included in these blocks are shown. The number of AR loads for the proposal method is the minimum number of AR loads among the above mentioned 9 AG strategies for commutative subsequences. For reference, the numbers of AR loads by those 9 AG strategies are shown in Table 3, where strategies X-1 and X-2 denote the results by the strategy X together with weighting factors given in the relative and absolute manners, respectively. From Table 2, the numbers of AR loads are reduced by appropriate utilization of the commutative code blocks for memory addressing. From Table 3, strategies 2 and 3 give slightly better memory allocations for several examples. These strategies consider freedom of computational order in narrow

Table 2 Code generation result (μPD77230:NEC).

Examples	Total Steps	Variables	Total AR updates	# of Commutative Code Blocks (# of Vars.)	# of AR loads	
					Method in [10]	Proposed Method
Lattice(5)	39	5	13	0(0)	2	2
Lattice(10)	85	5	32	0(0)	7	7
Cascade(6)	34	7	22	3(6)	4	4
WDLF(5)	46	5	12	2(4)	1	1
WDF(5)	71	14	33	3(6)	8	7
WDF(7)	113	23	53	4(8)	14	11
WDF(9)	145	32	75	8(16)	20	17
WDF(11)	180	40	90	5(10)	27	23
WDF(13)	214	49	110	6(12)	39	34
WDF(15)	253	59	132	7(14)	47	45
WDF(17)	294	69	154	8(16)	59	54

WDLF() : Wave Digital Lattice Filter (Order)
WDF() : Wave Digital Filter (Order)

Table 3 Comparison of strategies (# of AR loads).

# strategy	1	2	3-1	3-2	4	5-1	5-2	6-1	6-2	7-1	7-2	8-1	8-2	9-1	9-2
Lattice(5)	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Lattice(10)	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Cascade(5)	4	5	4	5	6	5	4	5	4	5	4	5	4	5	4
WDLF(5)	1	2	2	2	1	3	1	3	2	3	3	2	2	1	1
WDF(5)	8	7	8	8	7	7	7	7	7	7	7	7	7	7	7
WDF(7)	12	12	12	12	11	11	11	11	11	11	15	11	15	11	11
WDF(9)	20	19	19	19	17	17	17	17	17	17	20	17	20	17	17
WDF(11)	27	24	24	24	23	23	23	23	23	23	26	23	26	23	23
WDF(13)	39	34	34	34	35	35	35	35	35	35	35	35	35	35	35
WDF(15)	47	45	45	45	50	50	50	50	50	50	48	50	48	50	50
WDF(17)	59	54	54	54	61	61	61	61	61	61	58	61	58	61	61

WDLF() : Wave Digital Lattice Filter (Order)
WDF() : Wave Digital Filter (Order)

commutative code blocks. Other strategies, however, does not give better memory allocations, so that the constraints of these strategies are thought to be over or under constrainted in the memory allocation procedure. Therefore, strategies 2 and 3 are said to give rather more appropriate constraints to the proposed memory allocation procedure for these examples. However, more detailed analysis must be continued in order to show their effectiveness in general.

6. Conclusion

For a DSP with simple indirect memory addressing mode, a new memory allocation method which utilizes freedom in local computational orders for reduction in memory access overhead is proposed. This new method is applied to the existing DSP compiler, where the compiler scheme is extended for compensation in computational order for the derived memory allocation. Codes generated by this compiler include less memory access overhead for several examples.

More detailed analysis of the proposed method and their improvement must be continued. Memory allocation methods for sophisticated AR operations or multiple ARs will also be studied in near future. For further reduction in memory access overhead, an allocation method utilizing global computationl order for memory allocation or a method combined with other code optimization techniques will be investigated.

Acknowledgement

The authors are grateful to Prof. N. Fujii and Assoc. Prof. S. Takagi of Tokyo Institute of Technology for the valuable discussions. This work is supported by the Research Body of CAD21 (Computer Aided Design for 21th Century, Chair: Prof. H. Kunieda) in Tokyo Institute of Technology.

References

[1] R. Simar, Jr. and A. Davis, "The application of hign-level langueges to single chip digital signal processors," 1988 ICASSP, pp.1678–1681, 1988.
[2] J. Hartung, S.L. Gay, and S.G. Haigh, "A practical C language compiler/optimizer for real-time implementations on a family of floating point DSPs," 1988 ICASSP, pp.1674–1677, 1988.
[3] E.A. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A design environment for DSP," IEEE Trans. ASSP, vol.37, no.11, pp.1751–1761, Nov. 1989.
[4] N. Sugino, A. Toshikiyo, E. Watanabe, and A. Nishihara, "Computational ordering of digital signal processing networks and its application to compilers for signal proces-

sors," IEICE Trans., vol.J71-A, no.2, pp.327–335, Feb. 1988.
[5] N. Sugino, S. Ohbi, and A. Nishihara, "Computational ordering of digital networks under pipeline constraints and its application to DSP compilers," IEICE Trans., vol.E72, no.12, pp.1299–1306, Dec. 1989.
[6] S. Liao, S. Deradas, K. Keutzer, S. Tijang, and A. Wang, "Storage assignment to decrease code size," ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp.186–195, June 1995.
[7] R. Leupers and P. Marwedel, "Algorithms for address assignment in DSP code generation," ACM/IEEE ICCAD, pp.109–112, Nov. 1996.
[8] B. Wess, "Automatic code generation for integrated digital signal processors," Proc. ISCAS 1991, pp.33–36, June 1996.
[9] N. Sugino, S. Iimuro, A. Nishihara, and N. Fujii, "DSP code optimization utilizing memory addressing operation," IEICE Trans. Fundamentals, vol.E79–A, no.8, pp.1217–1224, Aug. 1996.
[10] N. Sugino, H. Miyazaki, and A. Nishihara, "DSP code optimization methods utilizing addressing operations at the codes without memory accesses," IEICE Trans. Fundamentals, vol.E80–A, no.12, pp.2562–2571, Dec. 1997.
[11] N. Kogure, N. Sugino, and A. Nishihara, "Memory allocation method for indirect addressing DSPs with ±2 update operations," IEICE Trans. Fundamentals., vol.E81-A, no.3, pp.420–428, March 1998.
[12] N. Kogure, N. Sugino, and A. Nishihara, "DSP memory allocation methods for indirect addressing with wide range update operation by multiple registers," Proc. 1998 IEEE APCCAS, pp.435–438, 1998.
[13] MOTOROLA, DSP56000/DSP56001 Digital Signal Processor User's Manual, 1990.
[14] Texas Instruments, TMS320C5x Digital Signal Processor User's Manual, 1997.
[15] NEC, μPD77230 Users Manual, 1987.

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 Associate Professor of Department of Advanced Applied Electronics, Faculty of Engineering. His main research interests are implementation techniques for digital signal processing, especially in compiler and software development tools for digital signal processors. Dr. Sugino is a member of IEEE.

Akinori Nishihara received the 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 Professor of the Center for Research and Development of Educational Technology. His main research interests are in one- and multi-dimensional signal processing, and its application to educational technology. He served as an Associate Editor of the IEEE Transactions on Circuits and Systems II from 1995 to 1997, and Editor-in-Chief of the Transactions of IEICE Part A from 1998 to 2000. He is now Educational Activities Committee Chair of IEEE Region 10 (Asia Pacific Region). Dr. Nishihara is a member of IEEE, EURASIP, ECS and JET.