ア ル ゴ リ ズ ム 81−2
（２００１．１１．２７）

ブロックソート圧縮アルゴリズムを用いた プログラム･コード圧縮

外村 元伸[†]

[†]（株）日立製作所　中央研究所

〒185-8601　東京都国分寺市東恋ヶ窪 1-280

TEL (042)323-1111, E-mail:tonomura@crl.hitachi.co.jp

概要　テキストやファイルなどのシリアル・データを圧縮するためによく使われている高効率な圧縮法は，プログラム・コードの圧縮にそのまま利用するには問題がある．特に，LZ 圧縮法は，ランダムに分岐した先で圧縮された命令コードを復号しながら解釈・実行することが困難である．そこで, LZ 法並の圧縮効率を持つブロックソート圧縮アルゴリズムがランダムな位置から復号化できる性質をもっているので，それを応用することを検討する．ブロックソート圧縮法では，コード列全体を巡回シフトし，適当に順序付け整列して配列をつくり，同じコードが連続しやすい最後列のみを圧縮符号化する．本提案では，ランダムに分岐した先で即復号しやすいように，最後列の整列前後の位置対応関係そのものを保存して圧縮符号化する方式を検討する．また，コード効率の悪い VLIW(Very Long Instruction Word)コードの圧縮にも応用する．

キーワード　ブロックソート圧縮法，BW 変換，プログラム・コード圧縮，VLIW コード圧縮

Program Code Compression using Block-Sorting Compression Algorithm

Motonobu TONOMURA[†]

[†]Hitachi Central Research Laboratory, Hitachi, Ltd.
1-280 Higashi Koigakubo, Kokubunji-shi, Tokyo 185-8601, Japan
TEL (042)323-1111, E-mail:tonomura@crl.hitachi.co.jp

Abstract　High efficient compression methods that are well used to compress serial data of texts and files have problems to use in that compression for program codes. Particularly, the LZ method is difficult to decode and execute the instruction words decompressing the compressed codes at randomly branched target address. So, in order to compress program codes, I consider a block-sorting compression algorithm that has property to decode at random addresses and the compression efficiency is equal to the LZ method. In the block-sorting compression method, whole code series are sorting in suitable order with the array of the cyclic shifts. In the proposal, compressing and coding only the last column that is easy to decode at branch address randomly in succession of same code. I consider a compressing and coding method that preserves the corresponding relation of previous and current sorting positions for the last column. And, it is applied to VLIW(very long instruction word) code compression which has poor code efficiency.

key words　block-sorting compression, BW transformation, program code compression, VLIW code compression

1. はじめに

携帯機器やコントローラ等に使われる組み込み型プロセッサでは，ハードディスクなどの 2 次記憶装置がほとんど利用できないばかりか，組み込まれるメモリ容量にも制限があるため，利用可能なプログラム容量が少ない．それで，できるだけコード効率のよいプロセッサ・アーキテクチャが望まれている．また，メモリを混載したシステム LSI では，貴重な内蔵メモリ容量を節約するためにもコード効率は重要である．

組み込み型プロセッサでは，当初，頻繁に使われる命令コードを短くした可変長の CISC(complex instruction set computer)型が使われていた．しかし，固定長命令コードの RISC(reduced instruction set computer;)型でコード効率のよいアーキテクチャを各メーカが追求したために，最近では RISC 型が主流になってきている．そして，各社組み込み型プロセッサのコード効率の追求がますます激しくなってきている．例えば，32 ビット長の命令コードをもつ ARM7 プロセッサは，その半分の 16 ビット長に短縮する命令コードを設けている [22]．これは実行時に，デコード可能な元の命令コードに伸張変換される．キャッシュ・メモリには，短縮コードで格納できるので，見かけ上のキャッシュ容量が増えてキャッシュ・ミスが軽減されるという効果もある．しかし，短縮コード化可能な命令は，部分的な命令セットに限られるので，大きな圧縮効果は期待できていない．圧縮率 70%程度の報告がなされている．

そこで，テキストやファイルなどのデータを圧縮するための高効率な圧縮法(テキスト・データ:10～30%; プログラム・コード:45%)[9](1977 年に Ziv と Lempel によって提案された LZ 圧縮法[30]がよく使われている)を応用することが考えられる．しかしながら，プログラム・コードは分岐命令をもつのでランダムに分岐した先で圧縮コードを復号しながら解釈・実行できなければならない．LZ 圧縮法は，シリアルに動いていくデータの一部を窓辞書としてダイナミックにポインタで自己参照する方式なので，ランダムに分岐した先の圧縮コードを復号できない．プログラム全体にわたってスタティックな辞書をもつ可変長の Huffman 符号化方式では，出現頻度が少なくて長い圧縮符号は復号処理が複雑になる[3,10,11,12,27]．可変長符号が構成しやすい Arithmetic 符号によるコード圧縮が提案されているが，分岐のためにバイト単位で扱い，分岐専用の圧縮コードが設けられているなど本来の方式に制限がある[13,14,15]．その他様々なコード圧縮方式が提案されているが[1,6,7,8,16,17,19,28]，一般に命令語は，オペコードとオペランド部からなり，オペコードの繰り返しパターンの相関がオペランド(多様なため)のものよりも強いという捩れがあり，両パターンの組合せが多様化し，テキスト・データに比べて圧縮しにくいのが特徴である．

ところで，最近，LZ 圧縮法[30]の圧縮率に匹敵するブロックソート圧縮法と呼ばれる別の圧縮法がその圧縮効率の高さから理論的な面で関心が集まっている[2,4,18,21,26]．ブロックソート圧縮法は，テキスト・データ全体に対して巡回シフト(あるいは回転シフト)列を作り，すべての巡回シフト列に辞書式順序付けなどを行って配列し，その最後の列を取り出して符号化するものである．最後の列は，同じテキスト記号が連続しやすいという特徴があるので，その連続長を符号化することで，テキスト・データが圧縮できる．ブロックソート圧縮アルゴリズムを検索 (圧縮検索)に応用することで検索効率を上げようとする提案がなされている[24]．そこでは，ブロックソート圧縮データの復号化原理において，ランダムに(どこからでも)復号化できる特徴を指摘している．本報告では，プログラム・コードの圧縮法にこの原理を応用することを検討する．すなわち，ブロックソート圧縮法では，全プログラム・コードを整列させてまとめて効率的に符号化することで，全体的に圧縮効率を上げることができる．そして，ブロックソート圧縮アルゴリズムを用いて符号化された圧縮プログラム・コードをランダムに即復号化しやいように，最後列の整列前後の位置関係をはじめから保存するかたちで圧縮符号化する方式を提案する．さらに，復号速度を上げるために，VLIW(Very Long Instruction Word)コードの圧縮をベースに考え，並列復号化を検討する．

2. Burrows-Wheeler 変換

まず，プログラムのコード圧縮アルゴリズムに用いる Burrows-Wheeler 変換について説明する．長さ n のプログラム x を $x[1,n] = x_1x_2\cdots x_n$ のコード列で表す．ここで，コード $x_i(1 \le i \le n$, i:位置番号)は，バイト(byte)あるいはワード(word)単位で，順序“＜”が定義できる集合 A の要素($x_i \in A$)であるとする．適当な順序として，コード x_i の出現頻度や辞書式順序が考えられる．x の部分列を $x[i,j] = x_ix_{i+1}\cdots x_j$, $x[i] = x_i$ によって表す．すると，プログラム x の巡回シフト列は，$X_1 = x_1x_2\cdots x_n$, $X_2 = x_2x_3\cdots x_nx_1$, ・・・, $X_{n-1} = x_{n-1}x_n\cdots x_{n-2}$, $X_n = x_nx_1\cdots x_{n-1}$ のようになる．これらを定義順序“＜”で整列させて，$Y_1 < Y_2 <\cdots< Y_n$ を得て，整列の位置番号 1 から n を振り付ける．今，列のコード列に最後の列 $y[1,n] = Y_1[n]Y_2[n]\cdots Y_n[n]$ を選ぶ．これを Burrows-Wheeler 変換(BW 変換)と呼んでいる．y[i] を定義順序“＜”で並べ換えると，z[1, n]を得る．す

なわち，置換π

$$\pi = \begin{pmatrix} y[1] & y[2] & \cdots & y[n] \\ z[1] & z[2] & \cdots & z[n] \end{pmatrix}$$

が行われる．図 1 に示すように，$Y_{j1} = X_1 = x_1x_2\cdots x_n$ であるとすると，$z[j_1] = x_1$ である．次に，X_1 の巡回シフト列は $X_2 = x_2x_3\cdots x_nx_1$ であるから，$y[j_2] = x_1$ なる j_2 に対して，$z[j_2] = x_2$ である．同様に順々に導いて，最後に X_{n-1} の巡回シフト列は $X_n = x_nx_1\cdots x_{n-1}$ であるから，$z[j_n] = x_n$ が出る．縦のコード列 y[1, n]に最後の列 $Y_1[n]Y_2[n]\cdots Y_n[n]$を選んだ理由は，巡回シフトによって，最後の列 $Y_1[n]Y_2[n]\cdots Y_n[n]$が先頭の列 $Y_1[1]Y_2[1]\cdots Y_n[1]$に置換されるからである．このように，最後の列のコード列を定義順序によって整列させて並べ替えると，先頭の列ができ，このとき置換によって(現整列位置番号，整列前位置番号)の対が求められるので，この関係を辿ることによって元のプログラムに復号できることがわかる．ただし，あらかじめ元のプログラムの整列位置番号がわかっている必要がある．

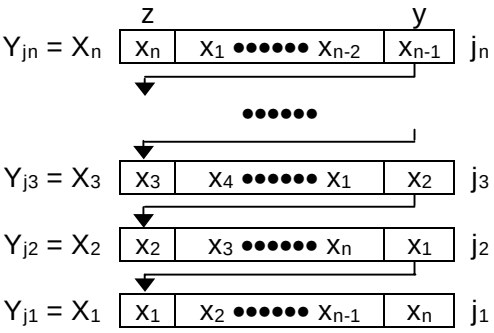


図 1　整列位置番号と置換の関係
Fig. 1　Relation between sorting position numbers and permutation.

3. コード圧縮

本論文で提案するブロックソート圧縮アルゴリズムに基づいたプログラム・コード圧縮法を説明するために，16 ビット長の命令セットをもつあるプロセッサの 31 個の命令語からなるプログラムの例 1(図 2)を用いる．命令語は表 1 に示すように，8 種類の機械語(Machine Word)で記述されているので，以下の説明を容易にするためにこれらを 0～7 の短縮コードに対応させて表現する．使用レジスタは，R0, R1, R2 と R3 である．@R2 は，

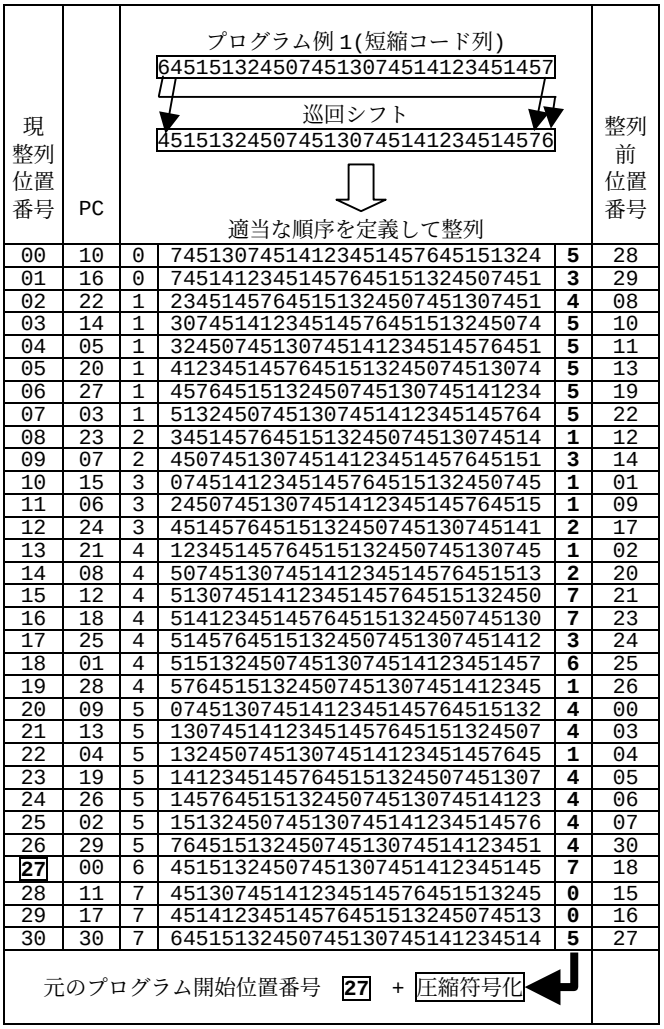
PC	label	MW	mnemonic	meaning
00		E000	MOV #0, R0	0 → R0
01		620C	MOV R0, R2	R0 → R2
02		6322	MOV.L @R2, R3	(R2) → R3
03	L1	330C	ADD R0,R3	R0 + R3 → R0
04		6322	MOV.L @R2, R3	(R2) → R3
05		330C	ADD R0, R3	R0 + R3 → R0
06		610C	MOV R0,R1	R0 → R1
07		4019	SHLR8 R0	R0 >> 8 → R0
08		620C	MOV R0, R2	R0 → R2
09		6322	MOV.L @R2, R3	(R2) → R3
10		3303	CMP/GE R0, R3	R3 ≥ R0 のとき true
11		8908	BT L2	true のとき L2
12		620C	MOV R0, R2	R0 → R2
13		6322	MOV.L @R2, R3	(R2) → R3
14		330C	ADD R0, R3	R0 + R3 → R0
15		610C	MOV R0, R1	R0 → R1
16		3303	CMP/GE R0, R3	R3 ≥ R0 のとき true
17		8906	BT L3	true のとき L3
18		620C	MOV R0, R2	R0 → R2
19		6322	MOV.L @R2, R3	(R2) → R3
20		330C	ADD R0, R3	R0 + R3 → R0
21	L2	620C	MOV R0, R2	R0 → R2
22		330C	ADD R0, R3	R0 + R3 → R0
23		4019	SHLR8 R0	R0 >> 8 → R0
24		610C	MOV R0, R1	R0 → R1
25	L3	620C	MOV R0, R2	R0 → R2
26		6322	MOV.L @R2, R3	(R2) → R3
27		330C	ADD R0, R3	R0 + R3 → R0
28		620C	MOV R0, R2	R0 → R2
29		6322	MOV.L @R2, R3	(R2) → R3
30		AFE3	BRA L1	Branch L1

図 2　プログラム: 例 1
Fig. 2　Program: example 1.

レジスタ R2 の内容が示すメモリ・アドレスを，#0 は，数値 0 を参照することを示す．比較命令はニーモニックが CMP/GE R0, R3 で，R3 ≥ R0 のとき分岐アドレスへ分岐する．加算命令は ADD R0, R3 で，レジスタ R0 と R3 の内容を加算し，R0 に格納する(R0+R3→R0)．シフ

表 1　機械語の短縮コードの定義
Table 1　Definition of short codes for machine words.

short codes	mnemonic		machine words
0	CMP/GE	R0, R3	3303
1	ADD	R0, R3	330C
2	SHLR8	R0	4019
3	MOV	R0, R1	610C
4	MOV	R0, R2	620C
5	MOV.L	@R2, R3	6322
6	MOV	#0, R0	E000
7	BT		89**, A***



現整列位置番号	PC				整列前位置番号
00	10	0	74513074514123451457645151324	**5**	28
01	16	0	7451412345145764515132450745 1	**3**	29
02	22	1	2345145764515132450745 1307451	**4**	08
03	14	1	3074514123451457645151324 5074	**5**	10
04	05	1	3245074513074514123451457645 1	**5**	11
05	20	1	4123451457645151324507451307 4	**5**	13
06	27	1	4576451513245074513074514123 4	**5**	19
07	03	1	5132450745130745141234514576 4	**5**	22
08	23	2	3451457645151324507451307451 4	**1**	12
09	07	2	4507451307451412345145764515 1	**3**	14
10	15	3	0745141234514576451513245074 5	**1**	01
11	06	3	2450745130745141234514576451 5	**1**	09
12	24	3	4514576451513245074513074514 1	**2**	17
13	21	4	1234514576451513245074513074 5	**1**	02
14	08	4	5074513074514123451457645151 3	**2**	20
15	12	4	5130745141234514576451513245 0	**7**	21
16	18	4	5141234514576451513245074513 0	**7**	23
17	25	4	5145764515132450745130745141 2	**3**	24
18	01	4	5151324507451307451412345145 7	**6**	25
19	28	4	5764515132450745130745141234 5	**1**	26
20	09	5	0745130745141234514576451513 2	**4**	00
21	13	5	1307451412345145764515132450 7	**4**	03
22	04	5	1324507451307451412345145764 5	**1**	04
23	19	5	1412345145764515132450745130 7	**4**	05
24	26	5	1457645151324507451307451412 3	**4**	06
25	02	5	1513245074513074514123451457 6	**4**	07
26	29	5	7645151324507451307451412345 1	**4**	30
27	00	6	4515132450745130745141234514 5	**7**	18
28	11	7	4513074514123451457645151324 5	**0**	15
29	17	7	4514123451457645151324507451 3	**0**	16
30	30	7	6451513245074513074514123451 4	**5**	27

図 3　コード列の巡回シフト生成とそれらの整列による配列化とその最後列の圧縮符号化.

Fig. 3　Generation of cyclic shifts for codes and their sorted array and compression for the last column.

ト命令は SHLR8 R0 で，レジスタ R0 の内容を右 8 ビットシフトし，R0 に格納する(R0>>8→R0)．移動命令は MOV R0, R1 で，レジスタ R0 の内容をレジスタ R1 に格納する(R0→R1)，MOV.L @R2, R3 で，レジスタ R2 の内容が示すメモリ・アドレスの **L**ong word 内容をレジスタ R3 に格納する((R2)→R3)など．分岐命令は BT で，分岐フラグが真(true)のとき分岐先 label へ分岐する．その機械語 89**の**には，分岐先アドレスを記述したいが具体的にどのように埋め込むかは後に説明する．また，分岐命令には，無条件分岐 BRA(機械語 A***)があるので，これも BT の中に含めることにし，同じ短縮コード 7 を用いる．89**と A***を区別する方法も，後で説明する．図 2 には，実際のプログラムの実行順序を示すためにプログラム･カウンタ PC が記述してある．

図 2 のプログラム例 1 を 16 ビット長の命令コード単位にプログラム･カウンタ(PC)順で並べる．すなわち，図 3 に示すようにコード列を表 1 の短縮コードを用いて記述する．さらに左へ 1 コード単位分ずつ巡回シフトし，それらを定義順序で整列し，配列を作る．整列された配列には，現整列位置番号をつける．配列の先頭列は，当然，同じ短縮コードが整列順に連続して固まる状況になる．そして，配列の最後列は，巡回シフトして定義順序で整列しているので，先頭列ほどではないが，同じ短縮コードが連続して固まりやすい状況になっている．本来のブロックソート圧縮法では，最後列のこのような性質を利用して，圧縮符号化している．その復号は，まず，この圧縮符号(圧縮符号化にはいくつかの方法が考えられているのでここでは適当な方法を仮定しておく)を伸張して最後列を得て整列させると，先頭列が自動的に求まることから，そのとき最後列の整列前位置番号と整列移動後の現整列位置番号を関係づけることによって行われる．このように，本来のブロックソート圧縮法では，復号操作が 2 段階で間接的になっているために，本提案では，最後列を圧縮符号化するかわりに，整列前位置番号と現整列位置番号の関係そのものを保存する圧縮符号化を行うことにより，復号操作が直接的になるようにする．

(整列前位置番号，現整列位置番号)の対関係そのものを保存して圧縮符号化するために,表 2 を作成する．ここでは符号化の圧縮効率を上げるために，絶対的な現整列位置番号を用いるかわりに，各短縮コードごとに相対的な位置番号を使う．例えば，短縮コード 1 の ADD(330C)命令については，現整列位置番号 02～07 にあるが，これを短縮コード 1 の表の相対位置番号 0～5 で表現する．そうすると，整列前位置番号も短縮コード別の相対位置番号で表現できる．さらに，整列前位置番号が，いくつかの連続する番号のグループに細分されるため，それらを下位表番号で相対表現する．そして，効率的な圧縮符号化の観点から，相対番号 p が p，p+1 と連続する場合，‘ + ’ 記号によって p+のように示す．さらに連続する場合，すなわち，p，p+1，...，p+1+j のように j+2 個(j は 0 より大きい数)連続している場合，p+j のように示す．このように相対番号が連続する場合，エントリ番号 q へのポインタだけを示すために，途中の相対位置番号 q+1+i へエントリする場合は，q#i のように記述する．ただし，i が 0 のときは，q#とする．実際には，q は下位表番号で記述する．

このようにして作成された表 2 を用いて，表 7 に示すような圧縮符号化を行う．各短縮コード別に，整列前の短縮コード s をポイントし，相対番号 p からどれくらい連続するかを記述する．ここで,区切り記号の( )を除いて符号化して圧縮する．あと，プログラムの開

始位置番号に対応する短縮コードと相対番号を同時に符号化しておく．このように圧縮符号化することで，直接的な復号操作が行える．

表 2　整列前位置番号と現整列位置番号の対応関係の保存と圧縮符号化
Table 2　Preserving and compression coding corresponding relation of previous and current sorting positions.

現整列位置番号	整列前絶対位置番号	下位表番号	整列前位置番号	
			短縮コード	相対番号の符号化
0: CMP/GE (3303)				
00	28	0	7	0+
01	29			
1: ADD (330C)				
02	08	0	2	0
03	10	1	3	0+
04	11			
05	13		4	0
06	19			1#4
07	22		5	1#
2: SHLR8 (4019)				
08	12	0	3	1
09	14	1	4	1
3: MOV R0, R1 (610C)				
10	01	0	0	0#
11	09		2	1
12	17	1	4	1#2
4: MOV R0, R2 (620C)				
13	02	0	1	0
14	20	1	5	0+
15	21			
16	23			1#1+2
17	24			
18	25			
19	26			
5: MOV @R2, R3 (6322)				
20	00	0	0	0
21	03	1	1	1+3
22	04			
23	05			
24	06			
25	07			
26	30		7	1
6: MOV #0, R0 (E000)				
27	18	0	4	1#3
7: BT (89), BRA (A***)**				
28	15	0	4	1#+
29	16			
30	27	1	6	0

表 3　圧縮符号化
Table 3　Coding for Compression.

短縮コード	符号化
0	7(0+)
1	2(0)3(0+)4(0,1#4)5(1#)
2	3(1)4(1)
3	0(0)2(1)4(1#2)
4	1(0)5(0+,1#1+2)
5	0(0#)1(1+3)7(1)
6	4(1#3)
7	4(1#+)6(0)

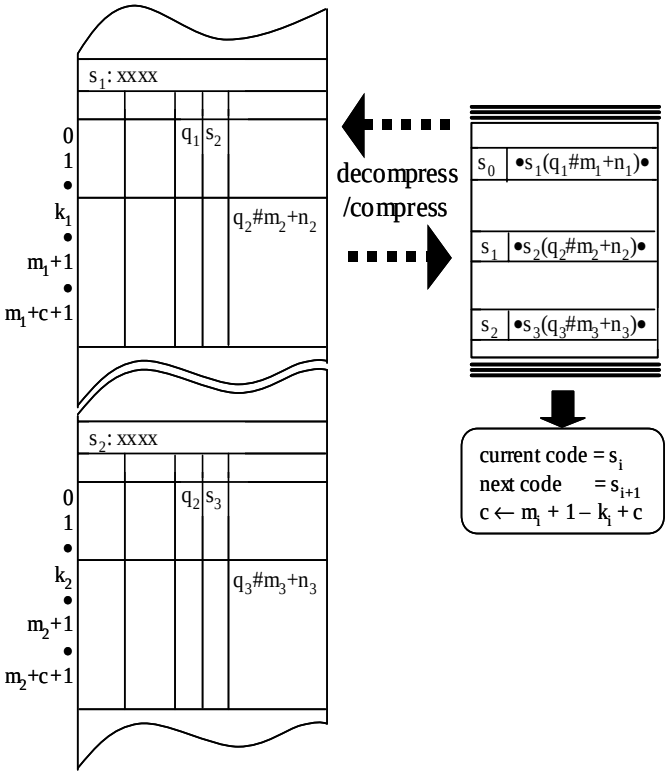


図 4　圧縮符号化とその復号化
Figure 4　Compression coding and its decompression decoding.

表 4　復号化の進行状況：圧縮符号の読み出しと実際の整列位置番号との対応付け計算.
Table 4　Decoding process: reading the compression codes and calculating the corresponding relation of the real sorting position.

PC	エントリの絶対位置番号（**太字**）	次エントリ符号読み出し	相対位置の内部カウント (c)
00	**27**	4(1#3)	16,17,18 (2)
01	**18**	5(1#1+2)	21,22,23,24,25 (4)
02	23,24,**25**	1(1+3)	07(0)
03	03,04,05,06,**07**	5(1#)	21,22 (1)
04	**22**	1(1+3)	03,04 (1)
05	03,**04**	3(0+)	11 (0)
06	10,**11**	2 (1)	09 (0)
07	**09**	4(1)	14 (0)
08	**14**	5(0+)	20(0)
09	**20**	0(0)	00 (0)
10	**00**	7(0+)	28 (0)
11	**28**	4(1#+)	14,15 (1)
12	**15**	5(0+)	21 (0)
13	20,**21**	1(1+3)	03 (0)
14	**03**	3(0+)	10 (0)
15	**10**	0(0#)	00,01 (1)
16	**01**	7(0+)	28,29 (1)
17	28,**29**	4(1#+)	16 (0)
18	15,**16**	5(1#1+2)	21,22,23 (2)
19	**23**	1(1+3)	05 (0)
20	03,04,**05**	4(0)	13 (0)
21	**13**	1(0)	02 (0)
22	**02**	2(0)	08 (0)
23	**08**	3(1)	12 (0)
24	**12**	4(1#2)	16,17 (1)
25	**17**	5(1#1+2)	21,22,23,24 (3)
26	23,**24**	1(1+3)	06 (0)
27	03,04,05,**06**	4(1#4)	16,17,18,19 (3)
28	**19**	5(1#1+2)	26 (0)
29	23,24,25,**26**	7(1)	30 (0)
30	**30**	6 (0)	27 (0)

表 4 に復号化の進行状況を示す．プログラム例 1 の圧縮符号(表 3)では，短縮コード 6(命令コード E000)の相対番号 0 からエントリして復号操作が開始される．元のプログラムの PC = 00 に相当するアドレスの機械語 E000 が命令デコードされ，実行される．次の PC = 01 に相当する命令コードは，表 3 の短縮コード 6 のエントリ 0 から読み出すと，短縮コード 4(命令コード 620C)なので，機械語 620C が命令デコードされ，実行される．この読み出しにおいて，相対番号符号 rc = 1#3 なので，その次の PC = 02 に相当する命令コードを得るためには，表 2 の短縮コード 4 の下位表番号 1 のエントリ#3 に相当する現整列位置番号 18(16 からエントリし，17, 18)をポイントする必要がある．実際には，表 3 の圧縮符号で，短縮コード 4 欄の符号 5(0+,1#1+2)から 3 番目の符号 1#1+2 を取り出し，短縮コード 5(命令コード 6322)を実行する．このとき同時に，明示されていない相対位置を内部ではカウントしておく必要がある．具体的には，エントリ#3 からマイナス 1 した c = 3 - 1 = 2 を記憶しておく．次の PC = 03 に相当する命令コードは，表 3 の短縮コード 5 欄から 1(1+3)を読み出し，短縮コード 1(命令コード 330C)の実行となる．そして，現整列位置番号 23 からエントリし(1#1)，同時に内部カウントしている c = 2 から，プラス 2 番目の 25 に対応するので，整列前絶対位置番号 07 をポイントしていることになる．内部カウントは符号#1 より c = 4(c ← c + 2)になる．以下，同様にプログラムが復号処理されながら実行されていく．

そこで，図 4 に一般的なかたちでまとめておく．圧縮符号化された表において，短縮コード s_0 欄の s_1 を読み出したところから考える．読み出した結果 $s_1(q_1\#m_1+n_1)$なので，下位表番号 q_1 で，$m_1 + 1$ (← $\#m_1$)の相対位置にエントリする．そして，実際のアドレスは，内部カウント c の値によって補正された位置 $m_1 + 1 + c$ をポイントすることになる．しかし，短縮コード s_1 欄において，実際には途中の位置 k_1 に相当するところに次の短縮コード s_2 の $q_2\#m_2+n_2$ が書き込まれている．そこで，次の内部カウント c は，$c \leftarrow (m_1 + 1 - k_1) + c$ のように更新される．ここで，k_1 は明示的には記述されていないので，$q_2\#m_2+n_2$ に到達するまで順々にカウントして求めるのが 1 つの方法である．もし，k_1 が大きすぎて逐次カウントの手間が許容できないようであれば k_1 を明示的に，例えば，$q_1\%k_1\#m_1+n_1$ のように記述する．さて，次の短縮コードが s_2 なので，圧縮符号表の s_2 欄から $q_2\#m_2$ に従ってエントリし，同様の操作を繰り返す．これを見てわかるように，符号+n_i の示す数値 n_i が大きいほど圧縮効率を上げるのに寄与することができる．

ところで，復号操作中に分岐命令の短縮コード 7 に遭遇すると，そこにはその分岐先の相対位置が埋め込まれている必要がある．分岐先アドレスを記述している元の命令語のフィールド(機械語 89**, A***の*部分)の長さは固定だが，圧縮符号には可変長の分岐先相対位置を特定の領域を設けて別途記述することができる．そのため，同時に分岐命令の種類も区別することができる．このように，ブロックソート圧縮法を用いたコード圧縮は，ランダムに分岐した先でも容易に復号できる特徴をもつことができる．

4. VLIW コード圧縮

高性能アーキテクチャでは，最大限の並列実行性を引き出すために，VLIW(Very Long Instruction Word)形式が使われることがある[5,20,23,25]．しかし，1つの命令コード長が長いのとその中で無効になる命令(NOP: no-operation)フィールドが占める割合が大きいために，コード効率が悪く，結果としてコード・サイズが大きくなるのが欠点である．そのため，大抵は NOPs を省略して圧縮する方法が提案されている．

この節では，VLIW コード圧縮にブロックソート圧縮法を適用することについて述べる．例として，図 5 に示すように，1 つの命令コードが４つの並列ブロック: pi1，pi2，pi3，pi4(i = 1，･･･，n: プログラム・カウンタ PC)のフィールドに分割されている場合について考える．もし，４並列ブロックを 1 つの命令コード単位にまとめてブロックソート圧縮を行おうとすると，圧縮効率が悪いと考えられる．なぜならば，まず，４個の各ブロックのフィールド長が同じならば(各ブロックの有効性を示すインデックス部があるものがあるが[23,25]，それは無視する，また，フィールド長が各々異なってもダミー・フィールドを追加することですべて同じ長さにできる)，コード単位を各ブロックに分割した方がよいだろう．さらに，各ブロックの命令セットが同じものならば，なおさら分割は都合がよい．そしてこのようにすれば，コード圧縮効率がよくなるが，復号がブロック位置順に逐次になることから，４並列にした意味が薄れる．そこで，４個の各ブロックが並列に同時復号できるように，ブロック位置 j = 1，2，3，4 別に p1j，p2j，･･･，pnj のように串刺しにして繋げる．これら並びを整列させ，ブロック位置別に分けないで混在させて整列させる．すると，ブロック位置にかかわらず混在した整列が出来上がる．復号は４並列の位置番号の同時指定によって行う．ブロック位置別に分けないで混在させて整列させても，正しく復号できるのは，それぞれが次の復号位置を明確にポイント(1-1 対応)しているためである．

以上説明した並列プログラム・コードの圧縮・復号の原理は，VLIW ばかりでなく，単一命令語アーキテクチャの RISC にも応用することができる．すなわち，例えば，図 6 に示すように，シリアルなプログラム・コードを 4 命令単位ごとに区切って，4 つの並列な系列，言わば，疑似 VLIW 形式をつくる．そして，この方式において分岐を問題なく行うために，元々の分岐命令(記号: ▶)に対して，それぞれ残りの 3 系列にその後連続して実行するアドレスへ対応して分岐するダミー命令(記号: ▷)を追加する．このようにすることで，シリアルなプログラム・コードを疑似並列化することが可能となり，復号の速度を加速することができる．

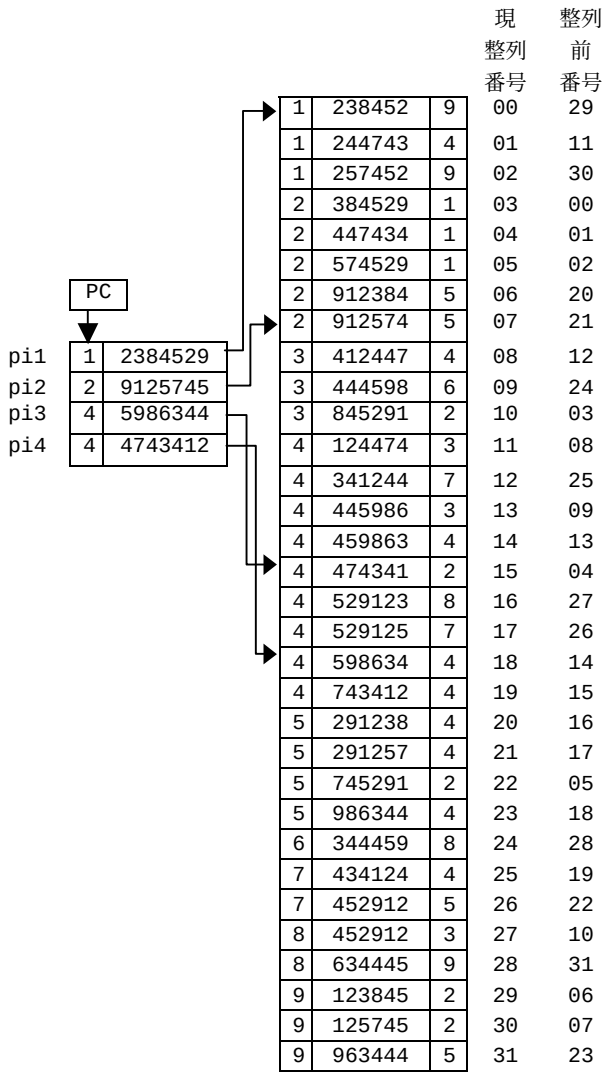


図 5 VLIW コードの混在整列
Figure 5 Merging and sorting of VLIW codes.

5. おわりに

組み込み型プロセッサでは，プログラム・コードのサイズ効率がよいことが要求されるので，高効率な圧縮プログラミング法を検討した．プログラムは任意アドレスへ分岐してそこから復号することが要求されるという特殊事情があるため，従来は，シリアル・データ圧縮を対象に提案された LZ 圧縮法，Huffman 符号化法や算術符号化法をベースにしてプログラム・コード用に改良されていた．しかし，処理が複雑になるので方式に制限を加えるため，また命令語特有のオペコー

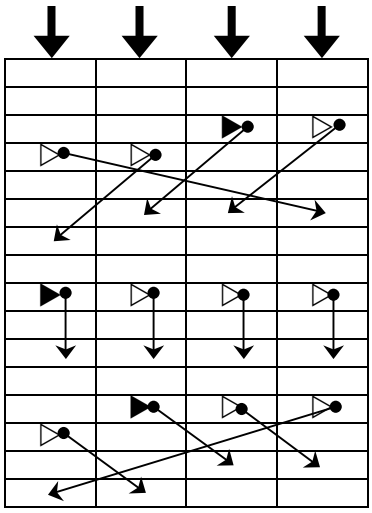


図 6　シングル命令コードの擬似 VLIW 化
Figure 6　Pseudo-VLIW coding of single instruction codes.

ドとオペランド部の繰り返しパターン相関の捩れのため，シリアル・データの場合よりも圧縮効率の低下を招いていた．

そこで，最近，LZ 圧縮法に匹敵する圧縮率が得られるということで注目されているブロックソート圧縮アルゴリズムがあり，その応用がほとんど検討されていなかったので，そのアルゴリズムを用いてどこへ分岐しても即復号できる性質をもつようなコード圧縮方式を提案した．本提案においては，これら条件を満たすしくみを中心に考えて展開した．連続する符号部分が多ければ圧縮効率が上がるのは当然なので，具体的に圧縮効率を向上させるには，s(q%k#m+n)のような圧縮符号化(s：短縮コード，q：下位表番号，%k：カウント補正数，#m：エントリ位置，+n：連続数)を最適コード化して実現する方法を採用し，それとともに実データを評価する必要がある．そのため，実際のプログラムのコンパイラ出力コードを入力とする圧縮処理系を作成して，データを収集する準備をすすめている．復号速度の向上対策としては，まずパラレルな VLIW コードの圧縮法を提案し，シングルな RISC コードを擬似 VLIW 化することで並列復号する方式を提案した．

参考文献

1) Araujo, G., Centoducatte, P. and Cortes, M.：Code Compression Based on Operand Factorization, 31th Ann. International Symposium on Microarchitecture, pp. 194-201, 1998.
2) Balkenhol, B. and Kurtz, S.: Universal Data Cmpression Based on the Burrows-Wheeler Transformation: Theory and Practice, IEEE Trans. on Computers, Vol. 49, No. 10, pp. 1043-1053, 2000.
3) Benes, M., Nowick, S.M. and Wolfe, A.：A Fast Asynchronous Huffman Decoder for Compressed-Code Embedded Processors，Proc. of IEEE International Symp. On Advanced Research in Asynchronous Circuits and Systems 1998.
4) Burrows, M. and Wheeler, D.J.: A block-sorting lossless data compression algorithm, SRC Research Report, 124 May 1994.
5) Conte, T. M.: Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings, Proc. of Microarchitecture'96, pp. 201-211, 1996.
6) Cooper, K.D.: Enhanced Code Compression for Embedded RISC Processors, Proc. of ACM SIGPLAN'97(PLDI), pp. 139-149, May. 1999.
7) Ernst, J., Evans, W., Fraser, C.W., Lucco, S. and Proebsting, T.A.: Code Compression, Proc. of ACM PLDI'97, pp. 358-365, Jun. 1997.
8) Franz, M. and Kistler, T.： Slim Binaries，Communications of the ACM, Vo1. 40, No. 12, pp. 87-94, 1997.
9) Interface: ファイル&画像圧縮技術の原理と応用，pp. 99-160, CQ 出版, Sep. 1996.
10) Kozuch, M. and Wolfe, A.: Compression of Embedded System Programs, Proc. of ICCD'94, pp. 270-277, 1994.
11) Lefurgy, C., Bird, P., Chen, I.C. and Mudge, T.: Improving Code Density Using Compression Techniques, 30th Ann. International Symposium on Microarchitecture, pp. 194-203, 1997.
12) Lefurgy, C., Piccininni, E. and Mudge, T.: Evaluation of a High Performance Code Compression Method, 32th Ann. International Symposium on Microarchitecture, pp. 93-102, 1999.
13) Lekatsas, H. and Wolf, W.: Code Compression for Embedded Systems, Proc of DAC'98, pp. 516-521, Jun. 1998.
14) Lekatsas, H. and Wolf, W.: Random Access Decompression using Binary Arithmetic Coding, Proc. of IEEE DCC'99, pp. 306-315, Mar. 1999.
15) Lekatsas, H. and Wolf, W.: SAMC: A Code Compression Algorithm for Embedded Processors, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems., vol. 18, no. 12, pp. 1689-1701, Dec. 1999.
16) Liao, S., Devadas, S. and Keutzer, K.: A Text-Compression-Based Method for Code Size Minimization in Embedded Systems, ACM Trans. on Design Automation of Electronic Systems, vol. 4, no. 1, pp. 12-38, Jan. 1999.

17) Lucco, S.: Split-Stream Dictionary Program Compression, Proc. of ACM PLDI 2000, pp. 27-34, 2000.
18) Manzini, G.: An Analysis of the Burrows-Wheeler Transform, Journal of the ACM, Vol. 48, No. 3, pp. 407-430, May 2001.
19) 門前淳, 安浦寛人: Byte Pair 符号化を用いた命令 ROM 圧縮, 信学技法, VLD2001-1, pp. 1-6, 2001.
20) Nam, S-J., Park, I-C. and Kyung, C-M.: Improving Dictionary-Based Code Compression in VLIW Architectures, IEICE Trans. Fundamentals, Vol. E82-A, No. 11, pp. 2318-2324, 1999.
21) Salomon, D.: Data Compression, Springer-Verlark, New York 1998.
22) Segars, S., Clarke, K. and Goudge, L.: Embedded control problems, Thumb and the ARM7TDMI, IEEE Micro, vol. 15, no. 5, pp. 22-30, Oct. 1995.
23) Suzuki, H, Makino, H. and Matsuda, Y.: Novel VLIW Code Compaction Method for a 3D Geometry Processor, Proc. of CICC 2000, pp. 555-558, 2000.
24) Tonomura, M.: Searching for Block Sorted Compression Data, 情報処理学会研究報告, アルゴリズム 76-2, pp. 9-16, 2001.
25) Tremblay, M.: The MAJC Architecture: A Synthesis of Parallelism and Scalability, IEEE Micro, Vol. 20 No. 6, pp. 12-25, Nov-Dec. 2000.
26) Witten, I.H., Moffat, A., and Bell. T.C.: Managing Gigabytes; Compressing and Indexing Documents and Images, second edition, Morgan Kaufmann Publishers, San Francisco 1999.
27) Wolfe, A. and Chanin, A.: Executing Compressed Programs on An Embedded RISC Architecture, Proc. 25th Ann. International Symposium on Microarchitecture, pp. 81-91, 1992.
28) Yoshida, Y., Song, B.Y., Okuhata, H., Onoye, T. and Shirakawa, I.: An Object Code Compression Approach to Embedded Processors, Proc. ACM ISLPED'97, pp. 265-268, 1997.
29) Yu, T.L.: Data Compression for PC Software Distribution, Software-Practice and Experience, vol. 26(11), pp. 1181-1195, Nov. 1996.
30) Ziv, J. and Lempel, A.: A Universal Algorithm for Sequential Data Compression, IEEE Trans. on Inform. Theory, vol. IT-23, no. 3, pp. 337-349, May 1977.