C-015

主記憶上のデータに対するブロックソートを用いた圧縮法

Data Compression Based on Blocksorting for Main Memory System

東大輔 *　　金子晴彦*
Daisuke Azuma　　Haruhiko Kaneko

1 はじめに

コンピューターの主記憶の容量には限界があるため、大きなサイズのデータを扱う処理等においては、主記憶に保持できないデータを磁気ディスク等の二次記憶装置へ転送する必要がある。主記憶と二次記憶装置の動作速度には大きな差があることから、二次記憶装置への転送はコンピューターの性能を下げる要因となっている。二次記憶装置へのデータ転送量を削減するため、例えば IBM Memory Expansion Technology(IBM-MXT) ではメモリコントローラーに LZ77 圧縮・伸長回路を搭載し、主記憶へ転送されるデータを圧縮し格納することによって、主記憶の容量を仮想的に増加させ、主記憶領域の有効活用を図っている [1]。また、[3] において主記憶データの特徴に適した圧縮手法が提案され、[4] においてメモリバスを流れるデータ量をデータ圧縮によって削減する手法が提案されるなど、データ圧縮技術を主記憶に使用する様々な手法が研究されている。

本稿では、IBM-MXT において使用されている LZ77 よりも一般に圧縮率が高いとされるブロックソートを用いた圧縮法を主記憶上のデータ圧縮に使用する手法を提案する。主記憶においてはランダムアクセス性を維持するために非常に短いデータに対してデータ圧縮・伸長を行う必要があるが、従来のブロックソートを用いた圧縮法は、エントロピー符号化に Huffman 符号を用いるため、圧縮するデータのサイズが小さい場合、Huffman 木を保持するためのヘッダー部の割合が非常に大きく、圧縮率が低下してしまうといった問題点を有する。これに対し、本稿では Huffman 木を近似した確率表で保持する手法及び Huffman 符号における記号のグループ化を用いる手法を提案し、ヘッダーサイズの削減を図る。また、提案手法の評価として、圧縮率と実行時間を示す。

本稿の構成は以下の通りである。第 2 章では従来の主記憶上のデータ圧縮技術の一つである IBM-MXT と、従来のブロックソートを用いた圧縮アルゴリズムを紹介する。第 3 章ではブロックソートを用いた圧縮法における Huffman 符号のヘッダーサイズを削減する手法を提案し、第 4 章では圧縮率、圧縮時間の評価と、ヘッダーサイズの評価を行う。第 5 章で結論と今後の課題を述べる。

2 従来のデータ圧縮技術

コンピューターの主記憶容量を有効活用するために、データ圧縮技術により主記憶のデータを圧縮してから格納することによって、仮想的に主記憶領域を増やす技術が利用されている。これにより、主記憶から磁気ディスク等の二次記憶装置へのスワップを防ぎ、読み書きの遅延を防ぐことができる。

*東京工業大学 大学院情報理工学研究科, Graduate School of Information Science and Engineering, Tokyo Institute of Technology

2.1 IBM-MXT

データ圧縮技術を主記憶上のデータに適用した技術の一つである、IBM-MXT がある [1]。IBM-MXT では、CPU キャッシュと主記憶をつなぐメモリコントローラーに、データ圧縮・伸長機能を有するキャッシュを導入している。メモリコントローラーは、2 次キャッシュから主記憶へ転送されるデータを圧縮し、仮想アドレスと物理アドレスのアドレス変換テーブルを保持しつつ、主記憶への圧縮データ書き込みを行う。CPU からの読み出し命令に対しては、アドレス変換テーブルをもとに圧縮されたデータを読み込み、伸長してキャッシュへ渡す。

IBM-MXT ではデータを 1KByte のブロックに分割して圧縮を行う。この分割したブロックの大きさをブロックサイズと呼ぶ。一般にファイル圧縮に使用されるブロックサイズは 64KByte 程度であることに対し、主記憶上のデータ圧縮ではランダムアクセスを行うためにファイル圧縮よりも小さなブロックサイズを使用する。

データの圧縮を行った場合、データの長さは可変長となり主記憶へ格納するアドレスが変化する。このため、IBM-MXT では主記憶上にアドレス変換テーブルを保持している。

IBM-MXT では、主記憶を大きく Sector translation table（STT）と Sectored memory region（SMR）の二つに分ける。メモリコントローラーは、キャッシュからデータを受け取ると、圧縮後の符号語が以下のどれに当てはまっているかによって処理を決定する。

1. 120bit 以下に圧縮される。
2. 圧縮されるが 120bit よりも大きい。
3. 全く圧縮されない。

1 の場合、120bit の圧縮データを STT へ書き込む。

2 の場合、圧縮データを 256Byte のブロックに分け SMR に書き込み、STT に圧縮データへのポインタを書き込む。

3 の場合、非圧縮データを SMR に書き込み、STT に非圧縮データへのポインタを書き込む。

上記 3 種類のデータは、STT の先頭 8bit の Control bit field(cbf) によって区別される。伸長の際には、メモリコントローラーは STT から 128bit を読み込み、Cbf に記述されている格納状態に対応してデータを読み込み、伸長を行う。

図 1 に IBM-MXT における主記憶の構成を示す。

IBM-MXT では圧縮アルゴリズムとして LZ77 が使用されている。これにより、転送された 1KByte のワードの中に、同一系列が多く存在していた場合にはサイズを小さくすることができ、主記憶容量を有効に活用できる。実際に IBM-MXT では様々なアプリケーションにおい

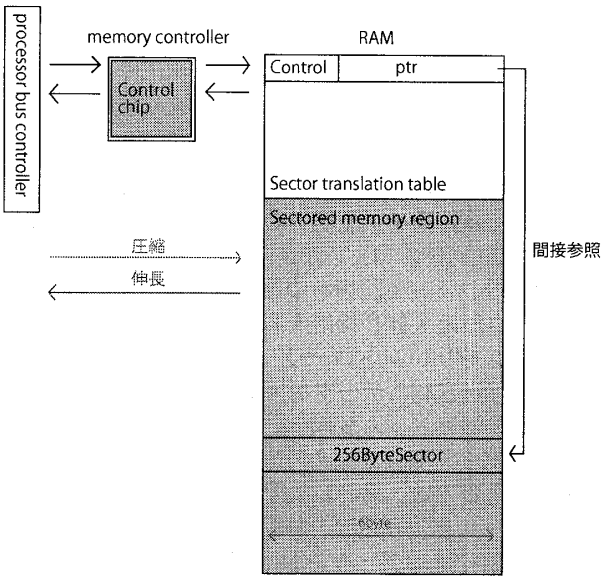

図 1　IBM-MXT における主記憶の構成

て 2〜7 倍の圧縮率が得られ、圧縮動作を行った場合でもスループットの低下は 3%程度である [1]。

2.2　ブロックソートを用いた圧縮アルゴリズム

ブロックソートを用いた圧縮は、一般に LZ77 アルゴリズムよりも圧縮率が高いとされる圧縮法であり、ブロックソート、MTF 変換、Huffman 符号を組み合わせたアルゴリズムである。圧縮はデータをブロックソート、MTF 変換、Huffman 符号による圧縮の順で符号化し、伸長は Huffman 符号による伸長、MTF 変換、ブロックソートの順で復号を行う。

ブロックソートとは、ブロックサイズで指定した長さの文字列を読み込み、図 2 のように文字列を巡回シフトしたものを辞書順にソートし、末尾の文字列を取ってきたものを符号語とするアルゴリズムである。また、復号の際に必要であるため、元の文字列がブロックソートしたデータの何番目に位置していたかを示すポインタを出力し、符号語とともに保持しておく。図 2 における"3"がそれである。

データ : aeadacab

aeadacab	0 : abaeadac
eadacaba	1 : acabaead
adacabae	2 : adacabae
dacabaea	3 : aeadacab ←元データ
acabaead	4 : baeadaca
cabaeada	5 : cabaeada
abaeadac	6 : dacabaea
baeadaca	7 : eadacaba

(1) シフト　　(2) ソート　　出力 : 3,cdebaaaa

図 2　ブロックソートによる符号化の例

ブロックソートの復号は、記号列を末尾に並べ、それらをソートすることによって先頭の文字列を復元し、文字の前後関係から元の文字列を復号するものである。復号の開始位置として、元の文字列が位置していたポインタを使用する。その復号の例を図 3 に示す。ブロックソートにより、記号の出現順序の依存関係を用いて、同じ記号を一か所に連続してまとめた記号列を生成することができる。

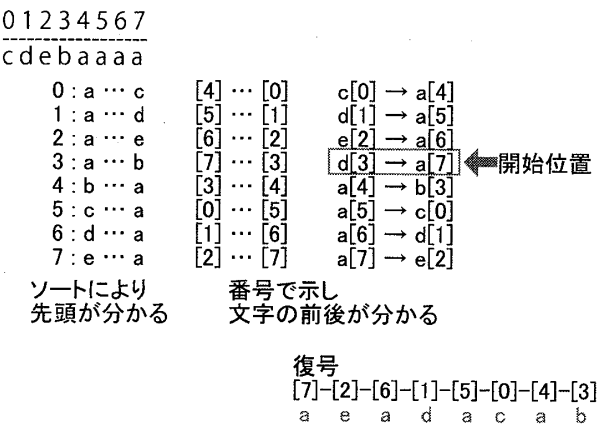

図 3　ブロックソート復号の様子

MTF(Move-to-Front) 変換とは、出現する文字を辞書順に並べた文字列を辞書として、読み取った文字を辞書の先頭に移動させながら、辞書の何番目の文字かを符号語とする変換アルゴリズムである。図 4 のように、MTF 変換により同一の記号が連続する文字列が全て"0"の列に符号化されるため、小さな文字の出現頻度が高い記号列に変換できる。MTF 変換によって、ブロックソート

文字列	辞書の変化	出力
baccddd	[a, b, c, d] -> [b, a, c, d]	1
*	* b を先頭に移動	
baccddd	[b, a, c, d] -> [a, b, c, d]	11
*	* a を先頭に移動	
baccddd	[a, b, c, d] -> [c, a, b, d]	112
*	* c を先頭に移動	
baccddd	[c, a, b, d] -> [c, a, b, d]	1120
*	* c を先頭に移動	
baccddd	[c, a, b, d] -> [d, c, a, b]	11203
*	* d を先頭に移動	
baccddd	[d, c, a, b] -> [d, c, a, b]	112030
*	* d を先頭に移動	
baccddd	[d, c, a, b] -> [d, c, a, b]	1120300
*	* d を先頭に移動	

図 4　MTF 変換の様子

によって同じ文字が連続して発生している符号語が、小さな数字の出現頻度が高い文字列へと変換される。復号も符号化と同様の辞書を用意し、読み込んだ数字に従って辞書の文字を出力し、辞書の文字の順序を変化させることにより復号する。

MTF 変換の出力をエントロピー符号化のひとつである Huffman 符号により圧縮する。Huffman 符号とは、出現頻度の高い記号を短い符号語で表現し、出現頻度の低い記号を長い符号語で表現することにより、平均符号長を短くする圧縮技術である。例えば、英文の場合、“e”は非常に多く出現し、“z”の出現は非常に少ない。そのような場合“e”は“0”と、“z”は“11111”と表現す

れば平均符号長を短くすることができる。記号の符号化表は Huffman 木という木構造で示され、これに従って圧縮・伸長する。

Huffman 木とは、出現確率の高い記号を要素にもつ葉ほど根に近く、出現頻度の低い記号を要素にもつ葉ほど根から遠くなるように位置する二分木である。Huffman 木によって、根に近い記号は短い符号語で表現され、根から遠い記号ほど長い符号語で表現される。

3 主記憶上のデータに対するブロックソートを用いた圧縮法

本節では、一般に LZ77 アルゴリズムよりも圧縮率が高いとされるブロックソートを用いた圧縮アルゴリズムによる主記憶データの圧縮法を提案する。また、主記憶データ圧縮に適した Huffman 符号として、近似した確率表で辞書を保持する手法及び記号をグループ化を行う手法を提案する。

従来のブロックソートを用いた圧縮と同様に、ブロックソート、MTF 変換、Huffman 符号を組み合わせたアルゴリズムにより、主記憶データの圧縮・伸長を行う。主記憶のデータを扱うためブロックサイズは 512Byte～4KByte 程度とする。

3.1 ブロックソートを用いた圧縮法のアルゴリズム

ブロックソートを用いた圧縮法による主記憶データの圧縮・伸長アルゴリズムを以下に示す。

圧縮アルゴリズム

ブロックソート、MTF 変換、Huffman 符号化の順で圧縮を行う。

1. 転送されてきたデータを読み込む。
2. ブロックソートを行い、復号に必要なヘッダーを保持する。
3. ブロックソートされた記号列に対して MTF 変換を行う。
4. MTF 変換された記号列に対して Huffman 木の作成及び Huffman 符号化を行う。
5. Huffman 木と Huffman 符号語の合計が元の記号列より短ければ"圧縮フラグ 1、ブロックソート復号のためのヘッダー、Huffman 木と Huffman 符号語"を書き出し、そうでない場合は"圧縮フラグ 0"と"元の記号列"を書き出し終了する。

圧縮アルゴリズムのフローチャートを図 5 に示す。

伸長アルゴリズム

圧縮とは逆に、Huffman 復号、MTF 復号、ブロックソートの順で伸長を行う。

1. 圧縮フラグを読み込み、"1"であれば 2 へ。そうでないなら非圧縮データを書き出して終了する。
2. ブロックソート復号のためのヘッダーを読み込む。
3. Huffman 木、Huffman 符号語を読み込み、木に従って符号語を伸長する。

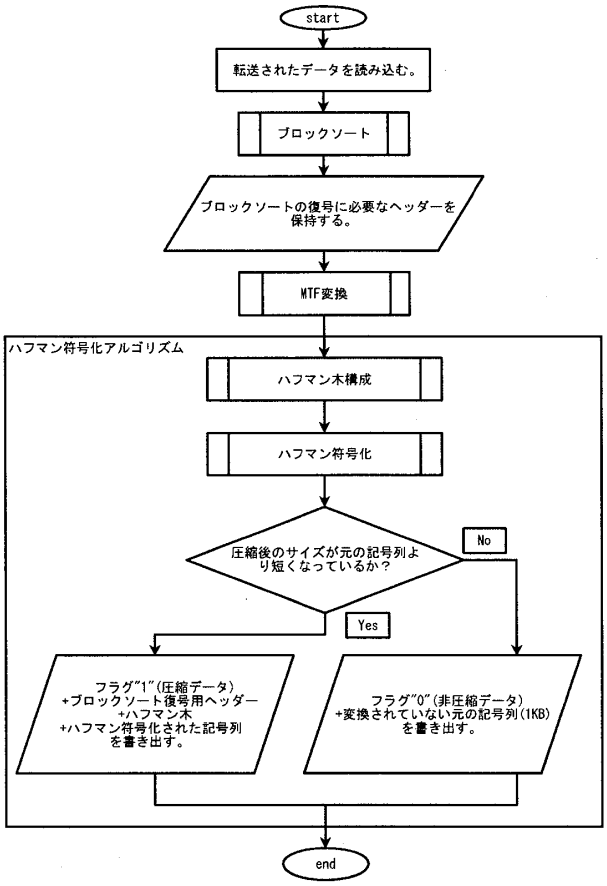

図 5 ブロックソートを用いた圧縮アルゴリズム

4. Huffman 復号された記号列に対して MTF 復号を行う。
5. 符号語をブロックソートし、1 で保持したヘッダーを使い復号し、書き出して終了する。

伸長アルゴリズムのフローチャートを図 6 に示す。

3.2 確率の近似による辞書サイズの削減

Huffman 木において、情報源記号の数が M 個である場合、葉の記号を表わすのに $M\log_2 M$ ビット、木の形状を表わすのに $2M-2$ ビットの容量が必要となる [2]。例えば情報源アルファベットの数が $M=256$ である場合、Huffman 木を表わす情報量は全部で 320Byte となる。主記憶データの圧縮においては、短いブロック単位ごとに対してこのヘッダーが必要なため、木の占める割合が大きくなり、圧縮率が低下することから、ヘッダーサイズを削減する手法を以下に示す。

Huffman 符号において、Huffman 木を構成するために使用した確率表を保持すれば、符号化の際と同じ Huffman 木を復号の際に構成でき、伸長することができる。Huffman 木を構成するために必要な確率表を、近似した値で保持することによって、そのサイズを削減することができる。

主記憶データ圧縮においては、短いブロックサイズを使用しているため、各記号の生起回数はそれほど大きくはない。例えば生起確率として各記号の生起回数を

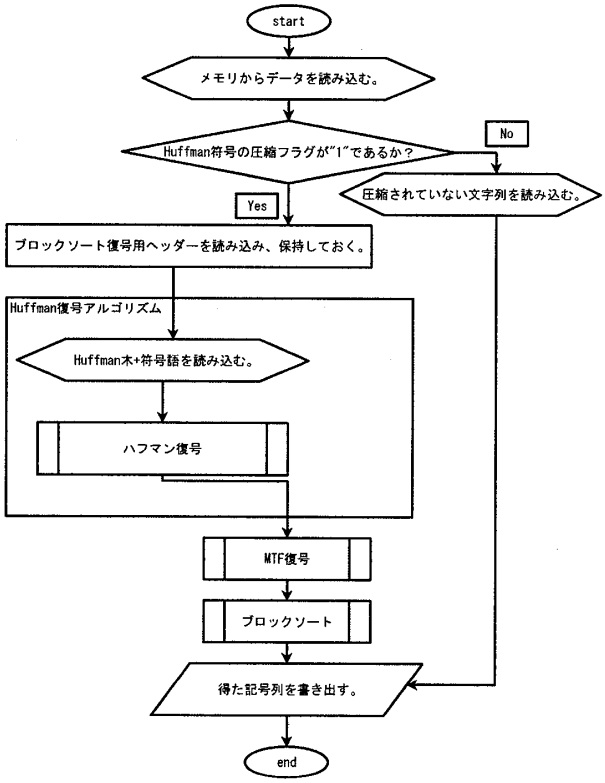

図 6　ブロックソートを用いた伸長アルゴリズム

1Byte で示せば、生起回数が 255 回を超える記号が複数現れない限り大きな誤差は生じない。従って、各記号の生起回数を 1Byte 以内に近似し保持することによって復号の際に Huffman 木を再構成することとする。これにより、本来 320Byte 必要であった Huffman 木を保持するためのヘッダーサイズを 256Byte に抑えることができる。本手法の圧縮データの構成を図 7 に示す。

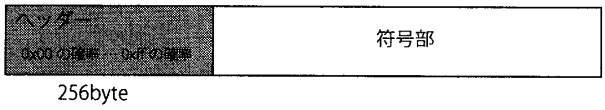

図 7　確率の近似をした圧縮データの構成

この方法を用い、ブロックソート、MTF 変換、Huffman 符号化の順に圧縮を行うアルゴリズムをアルゴリズム A とする。

3.3　記号のグループ化による辞書サイズの削減

Huffman 木は、符号化する対象の情報源記号の数が多いほど大きなサイズになる。したがって、Huffman 符号化する対象の数を少なくすれば Huffman 木のサイズを削減することができる。情報源記号をいくつかのグループにまとめ、各記号の出現確率の合計をグループの出現確率とし、グループ番号に対して Huffman 符号化を行う。これにより、Huffman 符号化の対象となる記号の数が少なくなり、Huffman 木を小さくすることができる。数値を復元するために、グループ番号を Huffman 符号化したものに加えて、グループ内の何番目の数値であるかを付加ビットとして保存する。

MTF 変換した記号列は小さな数値の生起確率が大きく、大きな数値の生起確率が小さいため、図 8 のようにグループ分けを行う。

グループ番号	要素	付加ビット長
0	0	0
1	1 2	1
2	3 4 5 6	2
3	7 8 9 10 11 12 13 14	3
4	15 16 17 18 19 20 …	4
5		5

図 8　Huffman 符号のグループ化

本手法に加え前述した確率の近似による辞書サイズの削減を使用すれば、情報源アルファベットの数が $M = 256$ であるから、Huffman 符号化の対象となるグループ数は 9 となり、辞書サイズは 9Byte となる。本手法での圧縮データの構成を図 9 に示す。

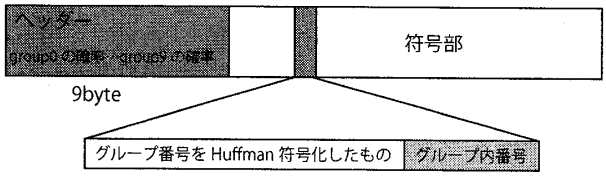

図 9　記号をグループ化した Huffman 符号の圧縮データ構成

本手法では記号ごとに適切な Huffman 木が構成されていないため、記号の出現頻度によっては最適符号長から大きくずれた圧縮データが発生する可能性があり、符号部の圧縮率は通常の Huffman 符号よりも低くなることが予想される。

ブロックソート、MTF 変換、記号のグループ化を適用した Huffman 符号の順に圧縮するアルゴリズムをアルゴリズム B とする。

4　評価

本節では、従来の主記憶データ圧縮に使用されている LZ77 アルゴリズムと、提案したアルゴリズム A、アルゴリズム B の 3 つに対し、ブロックサイズを 1KByte としたときの各ファイルの圧縮率及びヘッダーサイズの比較を行う。また、ブロックサイズを変化させながら、各アルゴリズムの圧縮率の変化及び圧縮伸長に要する実行時間を評価する。LZ77 は圧縮語に冗長性があるため、本稿では LZ77 の冗長性が取り除かれた LZSS アルゴリズムを使用し評価する。圧縮するファイルの対象として、Cargary Corpus を使用する。圧縮対象はメインメモリ上のデータであることが望ましいが、今回は初期的な実験のため、ファイルを使用して測定する。また、測定に使用したプログラムは、LZSS、ブロックソート圧縮を行う自作のプログラムである。

4.1　圧縮率

ブロックサイズを 1Kbyte としたときの圧縮率を表 1 に示す。圧縮率とは、以下の式に定義するように、圧縮後のファイルサイズに対する圧縮前のファイルサイズの比である。

$$圧縮率 = \frac{圧縮前のサイズ}{圧縮後のサイズ}$$

表 1　各アルゴリズムによる圧縮率

ファイル名	サイズ	LZSS	アルゴリズムA	アルゴリズムB
bib	111261	1.431	1.263	1.771
book1	768771	1.380	1.267	1.750
book2	610856	1.493	1.312	1.851
geo	102400	1.169	1.057	1.366
news	377109	1.369	1.218	1.675
obj1	21504	1.556	1.270	1.737
obj2	246814	1.734	1.392	2.003
paper1	53161	1.497	1.298	1.826
paper2	82199	1.473	1.306	1.837
paper3	46526	1.439	1.284	1.795
paper4	13286	1.495	1.303	1.840
paper5	11954	1.511	1.296	1.846
paper6	38105	1.542	1.324	1.886
pic	513216	3.256	2.209	4.573
progc	39611	1.625	1.348	1.950
progl	71646	1.881	1.517	2.318
progp	49379	1.916	1.512	2.310
trans	93695	1.656	1.383	2.023
平均圧縮率		1.635	1.364	2.020

評価の結果、アルゴリズム A の圧縮率は LZSS よりも低くなり、アルゴリズム B の圧縮率は LZSS よりも高くなった。アルゴリズム A の圧縮率が LZSS よりも低くなった原因は、辞書として保存しているヘッダーサイズが大きいためである。アルゴリズム A で用いているヘッダー 256Byte を取り除くと、その圧縮率はおよそ 0.5 となることから、アルゴリズム A の符号部の圧縮率が LZSS よりも高いこと、またヘッダーのサイズが非常に大きく圧縮率を低下させる原因となっていることがわかる。一方、ヘッダーサイズを削減したアルゴリズム B は従来の LZSS よりも優れた圧縮率を有することが分かる。

4.2　ヘッダーサイズの比較

それぞれアルゴリズムのヘッダー部と符号部を図 10 に示す。

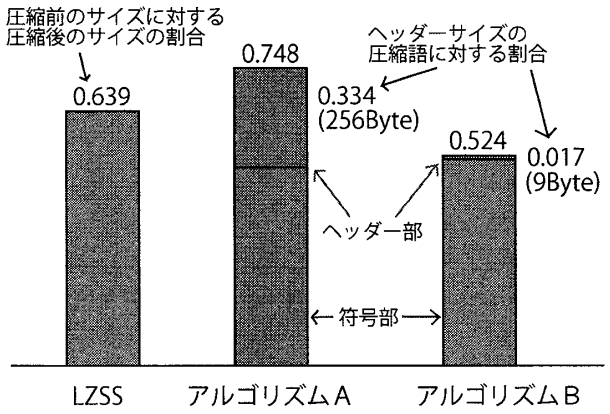

図 10　圧縮データにおけるヘッダーの割合

図 10 より、アルゴリズム A ではヘッダーサイズが非常に大きく圧縮率の低下が起きているが、アルゴリズム B では圧縮率が大きく低下することのないほどヘッダーサイズが小さいことがわかる。一方アルゴリズム B は記号ごとのエントロピー符号化でないため、アルゴリズム A よりも符号部の圧縮率が下がっている。

4.3　実行時間

各アルゴリズムの実行時間を計測する。Cargary Corpus を圧縮対象とし、各アルゴリズムの圧縮・伸長時間を測定する。時間は、Cargary Corpus の 18 ファイルをすべて圧縮・伸長するのに要した時間である。測定環境は CPU Celeron 2.80GHz, RAM 768MB, OS WindowsXP である。IBM-MXT で使用されていたブロックサイズ 1Kbyte の他に、512Byte、2KByte、4KByte のブロックサイズを用いて、実行時間の変化を調べる。またその際の平均圧縮率も評価する。平均圧縮率は、Cargary Corpus すべての圧縮率の平均である。表 2 に各アルゴリズムの圧縮・伸長時間と平均圧縮率を示す。また、図 11 にブロックサイズと圧縮時間の関係を、図 12 にブロックサイズと伸長時間の関係を示す。図 13 にブロックサイズと圧縮率の関係を示す。

表 2　各ブロックサイズにおける実行時間と平均圧縮率

		圧縮	伸長	
	圧縮単位	実行時間(s)	実行時間(s)	平均圧縮率
LZSS	512	103.125	1.843	1.496
	1024	235.146	2.031	1.600
	2048	560.578	1.937	1.695
	4096	1292.093	1.859	1.783
アルゴリズムA	512	65.265	17.765	1.053
	1024	137.343	36.640	1.379
	2048	239.718	66.328	1.733
	4096	442.250	125.828	2.033
アルゴリズムB	512	48.500	19.062	1.815
	1024	118.484	36.750	1.992
	2048	222.453	67.234	2.158
	4096	428.468	128.281	2.321

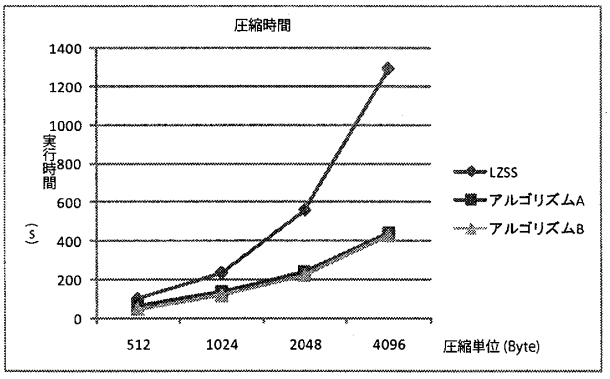

図 11　圧縮時間の比較

圧縮時間に関しては、各ブロックサイズともアルゴリズム A,B が LZSS より短いことがわかる。またブロックサイズを増加させた場合、LZSS の圧縮時間の増加率はアルゴリズム A,B よりも大きいため、大きなブロックサイズにおいては LZSS は適さないことがわかる。

逆に伸長時間に関しては、LZSS がアルゴリズム A,B より短く、またブロックサイズを変えても実行時間がほとんど増加しないことがわかる。ブロックソートを用いたアルゴリズム A,B は伸長速度が遅く、またブロックサイズを大きくすると実行時間が増加しているため、伸長時間においては LZSS が優れていることが示される。

また、各ブロックサイズにおける圧縮率について、ア

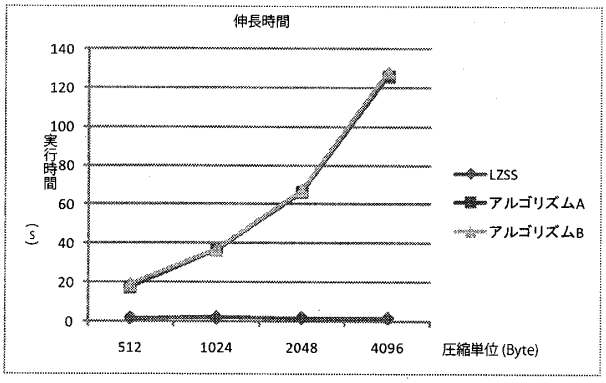

図 12　伸長時間の比較

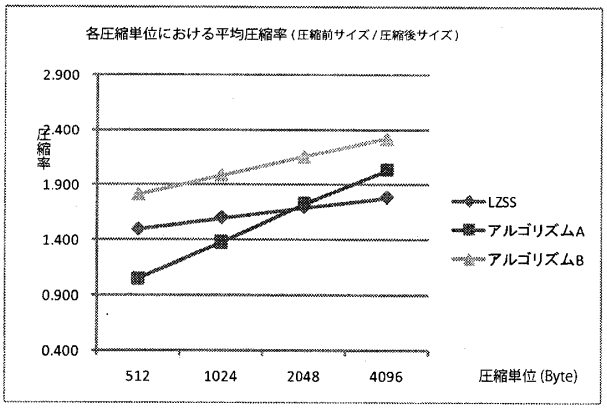

図 13　圧縮率の比較

ルゴリズム B が LZSS を常に上回っており、ブロックサイズが変化しても従来手法に劣ることはないことが示された。さらにアルゴリズム A は、小さなブロックサイズではそのヘッダーサイズが圧縮率を下げる要因となっており、他 2 つの手法に劣るが、ブロックサイズが大きくなるとそのヘッダーサイズの割合が小さくなり、圧縮率が高くなる。ブロックサイズが 2KByte を超えると LZSS よりも圧縮率が良くなることが示されている。

5　おわりに

本稿では、主記憶上のデータ圧縮にブロックソートを用いる手法を提案した。主記憶においては短い長さのデータを独立に圧縮する必要があるため、従来の手法では Huffman 符号のヘッダーサイズが非常に大きく、圧縮率が低下する原因となる。本稿ではヘッダーサイズを小さくする手法として、Huffman 符号化において記号のグループ化を行う手法を提案した。

Cargary Corpus 対して圧縮率の評価を行った結果、ブロックサイズ 1KByte においては LZSS の平均圧縮率が 1.635 であるのに対し、提案手法では平均圧縮率が 2.020 となり、従来の主記憶データ圧縮の手法よりも圧縮率が向上することを示した。また、実行時間の評価を行い、圧縮時間は提案手法が LZSS より短く、圧縮率と圧縮時間ともに提案手法が優れていることが示されたが、伸長時間は LZSS が提案手法よりも短いことを示した。

今後の課題として、提案した主記憶データ圧縮手法をハードウェアに実装し、実際のメインメモリ上のデータに対する圧縮評価を行うこと、また圧縮・伸長を導入したことによるメモリ動作の遅延時間を測定することがあげられる。

謝辞

本研究を行うに当たり, 有益な助言をいただきました藤原英二氏に心から感謝申し上げます.

参考文献

[1] P. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expantion Technology (MXT)," IBM J. Res. & Dev., Vol.45, No.2, pp.271–285, March 2001.

[2] 植松友彦, 文書データ圧縮アルゴリズム入門, CQ 出版株式会社, 1994.

[3] M. Ekman and P. Stenstrom, "A Robust Main-Memory Compression Scheme," Proc. 32nd Int. Symp. on Computer Architecture, pp.74–85, June 2005.

[4] 滝田裕, 坂井修一, 田中英彦, "チップマルチプロセッサにおけるキャッシュ-メインメモリ間動的データ圧縮の評価," 情報処理学会研究報告 計算機アーキテクチャ研究会報告, Vol.22, pp.73–78, March 2001.