高性能ストリームデータ圧縮・伸張ハードウェアの構成方式に関する研究

丸茂 孝一

2018年3月

筑波大学大学院博士課程

システム情報工学研究科博士論文

高性能ストリームデータ圧縮・伸張ハードウェアの構成方式に関する研究

丸茂 孝一

博士（工学）

（コンピュータサイエンス専攻）

指導教員　山際 伸一

2018年3月

概要

本論文では、高まる伝送系の高速化要求に対応するため、データの可逆圧縮技術に注目する。高速伝送技術の進歩に追従するため、流れ続けるデータを一切止めること無く、可逆的に圧縮・伸張することが可能なアルゴリズムを開発し、実用化が可能なハードウェアを実現する方法を述べる。既存の可逆圧縮技術をハードウェア化する際に問題となる点を明らかにし、これを回避する手法を要素技術として開発する。開発された要素技術を元にハードウェア化された圧縮・伸張システムを構築し、圧縮率や速度などの特性を報告する。さらに実用的な技術とするため、いくつかの改良を行い、従来の可逆圧縮技術では不可能だった広帯域・低遅延・省リソースな圧縮・伸張システムを実現し、伝送系の実装上の制約を超える、高速大容量のデータ伝送を支える技術の開発を行う。

目 次

図目次

表 目 次

第1章　序論

1.1　研究の背景

コンピュータ・システムでは大量のデータを可能な限り速く転送する必要があり、高速データ伝送技術の高性能化への要求は高まり続けている。今日、すでに基板上の銅配線で Gbps のオーダーに達しており、これからさらに上がっていくと予測される。また、高速化への要求に対応するため、データ伝送系の幅を増やすという手法も採られている。

例えば現在の PCI Express Generation 3 では、8Gbps × 16 レーン並列という構成が一般的に使われている。規格上は 8Gbps × 32 レーンまで策定されているが、次世代の Generation4 では 16Gbps × 64 レーンとそれぞれ倍になる。シリコン・チップ間の通信では HBM（High Bandwidth Memory）のような 1024bit バスといった、非常に幅の広い接続も存在する。しかし、伝送系の高速化は、クロストークや反射、ジッタといった実装技術の限界で制限されてしまい、性能向上には限界がある。

本研究では増え続けるデータ伝送の高性能化の要求に対応するため、圧縮技術に注目した。データの情報量は維持したまま、データの大きさを減らす、可逆圧縮の技術は従来より使われており、有名な可逆圧縮の方式としてハフマン符号化 [1][2] や LZ77[3]、LZ78[4] などが存在する。これら既存の可逆圧縮技術は、主にプロセッサによるソフトウェア実装となっている。

TCP/IP 層のドライバに LZO 圧縮をソフトウェアで実装した [5] の文献では、100M Ether では物理速度を超える転送速度が得られたが、Gbit Ether ではソフトウェアの処理による遅延がオーバヘッドとなり、かえって速度が落ちてしまうという結果が報告されている。今日では CPU の性能は向上しているが、同時に伝送系も性能向上しているため、いたちごっことなっている。また、ソフトウェアでの圧縮ではデータをある塊で処理する「ブロック処理」であるため、途切れること無く流れ続けるデータ（ストリームデータ）を扱うことができない。ブロック処理ではデータをメモリに一旦貯めるため、必ず遅延時間が発生する。伝送系ではこの遅延時間も性能向上を妨げる原因となっている。

伝送技術の進歩に追従可能な圧縮・伸張を実現するためには、ハードウェア化が不可欠であると考えられる。可逆圧縮のハードウェア化の要求は強く、例えば Intel から LZ 系の圧縮をハードウェア化した大規模な LSI が提供されている [6]。しかし、この LSI でも圧縮速度は約 20Gbps であり、LSI 自体が持つ PCIe の帯域である 50Gbps には届いていない。

既存の可逆圧縮技術は「シンボル・ルックアップ・テーブル」と呼ばれるテーブルを作成し、出現頻度の高いデータを短い「シンボル」に変換することで圧縮を行っている。この方式のハードウェア化を検討したとき、以下の問題が浮上する事がわかった。（1）処理時間が不確定、（2）テーブルのメモリサイズが予測できない、これらの問題により既存の圧縮技術

では、流れ続けるストリームデータを連続的に圧縮することが困難となっている。伝送速度の高速化に追従可能なスケーラビリティを持つ新しい可逆圧縮技術が切望されている。

1.2 本研究の目的

以上の背景から、既存の圧縮技術をハードウェア化する際の問題を解決する新しい要素技術を開発し、ストリームデータの可逆圧縮を可能とするハードウェアを設計し、新しい可逆圧縮技術を開発することを探求する。

本研究は高速伝送経路に適応可能なハードウェアベースのストリームデータ可逆圧縮技術の開発を目的として行った。

提案する方式は「LCA-DLT」と呼ばれる。LCA-DLT はデータの出現頻度の分布を、一定量に制限されたテーブル（メモリ）上で動作する動的ヒストグラム技術を元に、完全にハードウェア化可能な方式となっている。さらに、LCA-DLT では圧縮器で生成されたシンボル・ルックアップ・テーブルを伸張器側に送る必要が無いため、連続したストリームデータを途切れること無く圧縮・伸張することが可能となる。データをパイプライン的に圧縮・伸張することが可能となり、遅滞なくストリームデータを流しながら、圧縮・伸張が可能な方式となる。

1.3 本研究の貢献

動的ヒストグラムを適応させた LCA-DLT の開発

LCA アルゴリズムをハードウェア化、及び動的ヒストグラム技術を新たに開発した。これらの開発により新しいハードウェアベースのストリームデータ圧縮が可能となった。

Lazy なテーブル管理手法を適応させた圧縮技術の開発

LCA-DLT のテーブル管理において、故意に圧縮を無効にすることにより、パイプラインのストールを完全に排除する手法を発見した。これにより、ストリームデータを一切止めること無く圧縮・伸張が可能なハードウェアの開発が可能となった。

時分割マルチスレッド技術を適応させた圧縮技術の開発

LCA-DLT のパイプラインのデータハザードを回避するため、時分割マルチスレッド技術（Time-Sharing Multithreading）を適応させた。これにより、動作周波数の向上、ハードウェアリソースの使用量の減少が可能となり、より実用的なハードウェアが可能となった。

以上より、本研究の目的である、ストリームデータの可逆圧縮が可能なハードウェアの開発に必要な要素技術を解明し、全く新しい可逆圧縮技術の提案・利用が可能となった。

1.4 本論文の構成

2 章より高速伝送技術および既存の圧縮技術について言及し、ハードウェア化の妨げとなる問題点を解明する。それら問題点を回避するため、(1) 処理を一定にする手法の開発、(2) 一定メモリ量で動作する動的ヒストグラム技術の開発、が必要となることを解明する。

3 章では提案されたストリームデータ圧縮技術のハードウェアの設計・開発を行う。

4 章では 3 章で開発されたハードウェアの問題点を明らかにし、実用的なハードウェアの設計・開発を行う。この中で Lazy 方式という手法が生まれている。

5 章では 4 章で開発されたハードウェアで、高速化の妨げとなるボトルネックを探し、さらに高性能なハードウェアの設計･開発を行う。この中で時分割マルチスレッド技術（Time-Sharing Multithreading:TSM）の適応が行われる。

6 章を本論文のまとめとする。

第2章　研究の背景

本章は既存技術とその問題点について述べ、新技術に必要となる要素を特定する。

2.1　伝送系の高性能化とその問題

コンピューティング・システムの進化に従い、コンポーネント間、システム間のデータのやりとりを支えるため、高速データ伝送技術への高速化の要望が高まり続けている。

現在利用されている通信規格の代表的な例をいくつか挙げる。

- PCI Express [7]
  現在普及しているのは gen3 と呼ばれる規格となっている。gen3 はデータレート 8Gbps の伝送系を 1〜16 レーン（規格上は最大 32 レーン）束ねている。近年、次世代の gen4 がリリースされ、データレート 16Gbps、最大 64 レーンとそれぞれ倍とされている。

- Ethernet [8]
  現在、一般家庭向けではギガビット・イーサネット（1000BASE-T が主流）が普及している。1000BASE-T では 250Mbps の伝送系を 4 本使用し 1Gbps となっている。業務用では、銅線では 10 ギガビット・イーサネットが普及しており、光ファイバーを用いた 100 ギガビットイーサネットも利用されている。一般家庭への 10 ギガビットイーサネットの普及も期待されていたが、価格や発熱などの要因により進んでいない。このため、伝送速度を下げたマルチギガビット・イーサネット（2.5Gbps または 5Gbps）規格が制定されている。

- USB [9]
  現在 USB3.0（5Gbit/s）もしくは USB3.1（10Gbit/s）が利用されている。USB3.1 を 2 本束ねた USB3.2 がリリースされており、最大 20Gbit/s の速度に対応する。

- HDMI [10]
  HDMI 2.0 でいわゆる 4K60p に対応するため、伝送系の帯域が 18Gbps に引き上げられた。次世代の HDMI 2.1 では 8K60p 対応のため、48Gbps とさらに帯域が引き上げられる。

- メモリ・インターフェース [11]
  現在は DDR4 が主流となっている。速度の違いでいくつか規格があるが、DDR4-3200・64bit バスという規格で 1600MHz、25.6GB/s の速度となる。他にビデオメモリ向けの

GDDR5（伝送系の速度 8Gbps、通常 256bit バス程度）、HBM（最大 2Gbps、1024bit バス）などが様々な規格が存在する。

- その他チップ間
  CPU〜チップセット間、LSI〜LSI 間のインターフェースとして
  - QPI [12]、インテル、19.2GB/s〜25.6 GB/s
  - Hyper Transport [13]、AMD 他、バージョン 3.1 で最大 51.2 GB/s
  - NVLink [14]、nVidia,IBM 他、〜160 GB/s

  等が利用されている。それぞれ 10Gbps〜20Gbps の伝送系を複数束ねて実装されている。主にチップ間の伝送で利用されるため、数 cm〜数 10cm 程度の伝送距離を想定している。

これら多くの規格において、すでに伝送速度が、伝送系 1 ライン当たり Gbps を超えるのが当たり前となっており、規格によっては 10Gbps〜20Gbps に達している事がわかる。さらにこの高速な伝送系を、複数チャネルを束ねて帯域を稼いでいる。

このように、伝送系の高速化の要求に対応するためには、

- 伝送系の速度（周波数・変調方式）を上げる
- 伝送系の並列度を増やす

という対策が基本となっている。

しかし、これら伝送系の高速化は実装上の制約・物理限界との戦いとなっており、性能向上には限界がある。上に挙げた規格においても、伝送系 1 ライン当たりの速度は軒並み 10Gbps〜20Gbps 付近で上限となっており、コストや発熱を考慮した場合、現在の技術ではここが一つの限界となっていることが見える。なお、100Gbps クラス〜それ以上の伝送には光ファイバーが主に利用されているが、コストアップの要因となっている。

性能向上の難しさの例として PCI Express の進歩が挙げられる。

表 2.1: PCI Express の進歩

世代	登場年	リンク速度	転送帯域	レーン数	エンコード方式
gen1	2003 年	2.5GT/s	2Gbit/s	1〜32	8b/10b
gen2	2007 年	5GT/s	4Gbit/s	1〜32	8b/10b
gen3	2010 年	8GT/s	8Gbit/s	1〜32	128b/130b
gen4	2017 年	16GT/s	16Gbit/s	1〜64	128b/130b

PCI-SIG（Special Interest Group）[7] では世代毎に帯域を倍にすることを公約としており、これは守られている。しかし、gen2 → gen3 の際、リンク速度が 10GT/s になり、実装上困難

となったためエンコード方式を変更し、リンク速度を 8GT/s に抑える対策が取られた。また、それまで 3〜4 年間隔で新しい規格がリリースされていたが、gen3 → gen4 では 7 年かかっている。これらから伝送系の高速化が様々な制約により難しくなっていることが伺える。

また、10 ギガビットイーサネット規格の制定後、若干速度を落としたマルチギガビットイーサネットが制定された事からも、伝送速度の高速化による価格上昇や発熱の増加が問題となっていることが伺える。

2.2 データ圧縮技術

上記の事から、伝送速度の高速化の要求に対し、それを転送速度や並列度の向上だけで対応するのは困難が伴う。このため、データの量そのものを減らす事が不可欠であり、圧縮技術の検討へとつながった。

圧縮技術と伝送系の関係を考えるため、まずは現在利用されている圧縮技術の方式を調べる。なお、圧縮の際、元のデータを小さなデータに置き換えるが、この小さなデータは「シンボル」と呼ばれる。また、データを圧縮しサイズを小さくすることを、「圧縮」「符号化」「エンコード」などと呼ぶ。逆のデータを元に戻す作業は、「伸張」「復号化」「デコード」などと呼ばれている。

圧縮方式には大きく 2 つの方式がある。

- 非可逆圧縮
- 可逆圧縮

なお、本研究では特に断りが無い限り可逆圧縮を研究対象とする。

2.2.1 非可逆圧縮

非可逆圧縮は、圧縮されたデータを伸張したときに、得られたデータが元のデータとは異なる（劣化する）方式の圧縮である。元データの情報量は保たず、「不必要な情報」を削除することで圧縮を行う。ロッシーな圧縮、有歪みデータ圧縮とも呼ばれる。主に画像や音声など、いわゆるメディアデータの圧縮に用いられている。主な圧縮方式例を表 2.2、表 2.3 に列挙する。

非常に多くの規格が存在するが、これらの共通点として「最終的なデータの消費者が人である」という点が挙げられる。人の目や耳（及び脳）に気付かれにくい情報を「不必要な情報」として削除することで、非常に高い圧縮率を実現する。この分野の研究では、なにが人にとって不必要な情報かを調査することが一つの大きな課題となっている。

また、どこまで不必要とするかをユーザが与えることが可能で、品質の要求を与えることにより、データの圧縮率をコントロールすることが出来る。つまり、要求されるデータのサイズ（帯域）から、必要なだけ品質を落とし、要求を満足させることが可能となる。この特性により、伝送路の帯域が制限されているアプリケーションでの利用に向いている。

表 2.2: 画像圧縮の例

規格・方式	備考
JPEG	静止画圧縮方式の一つ。ブロック単位で離散コサイン変換を用いる。
MJPEG	各フレームを JPEG で圧縮する方式。
MPEG-1	ビデオ CD 等で使用された規格。フレーム間予測と離散コサイン変換を用いる。
MPEG-2	DVD や日本の地上波ディジタル等に利用されている規格。基本的な方式は MPEG-1 と同等だが、いくつか機能拡張されている。
MPEG-4 AVC/H.264	MPEG-2 の倍の圧縮率を目標に制定された規格。ネット配信などに利用されている。
H.265	H.264 の次世代機各。携帯向けの配信から 8K 動画まで広く対応する。
VP9	Google が開発したオープンな規格。H.265 に対抗して策定された。YouTube がこちらに移行中。

表 2.3: 音声圧縮の例

規格・方式	備考
MP3	正式な名称は MPEG-1 Audio Layer-3。人の聴覚特性を利用した圧縮方式。
AAC	MP3 の次の世代の規格。MPEG-2・MPEG-4 の仕様の一部として制定されている。
AC-3	ドルビーデジタルと呼ばれる方式。DVD や BD など、様々な記録媒体で利用されている。

2.2.2 可逆圧縮

圧縮データを伸張したときに、データが元のデータから変化しない圧縮を可逆圧縮と呼ぶ。ロスレス圧縮、無歪みデータ圧縮とも呼ばれる。可逆圧縮はデータが変化しては困るアプリケーションへの応用が可能であり、データの種類・目的を問わず利用可能である。

一方、多くの場合、圧縮率は非可逆圧縮のそれより悪い。元のデータが持つ情報量を削ることは出来ず、冗長性を持った情報のみが減らされるためである。また、全く冗長性の無い情報を圧縮することは不可能である。完全なランダムデータ、ホワイト・ノイズなどがこれに該当する。

可逆圧縮の代表的なアルゴリズムをいくつか挙げる。大きく分けてエントロピー符号と辞書式がある。エントロピー符号の代表的なアルゴリズムとしてハフマン符号が、辞書式の圧縮方式の代表的なアルゴリズムとして LZW が有名である。

可逆圧縮アルゴリズムの種類を簡単に表 2.4 にまとめる [15]。

表 2.4: 可逆圧縮アルゴリズムの系譜

エントロピー符号	ハフマン符号	静的ハフマン符号	
		動的ハフマン符号	
	算術符号	静的算術符号	
		動的算術符号	
辞書式	LZ77 スライド窓方式	LZR	
		LZSS	LZB
			LZH
		LZS	
		ROLZ	LZRW1
			LZP
		LZX	
		LZO	
		LZ4	
	LZ78 動的辞書方式	LZW	LZC
			LZMW
			LZWL
		LZJ	

ハフマン符号

代表的なエントロピー符号のアルゴリズム。「ハフマン木」と呼ばれるツリーを用い、出現頻度の高いデータに短いシンボルを割り当てる事でデータの圧縮を行う。静的ハフマン符号 [1] と動的ハフマン符号（適応型ハフマン符号）[2] がある。

静的ハフマン符号の基本アルゴリズムでは以下の手順で圧縮を行う。

1. まずある塊のデータを読み込み、データ毎の出現頻度をすべて求める。
2. ハフマン木の、ルートに近い出現頻度の高いデータに短いシンボルを割り当て、リーフに近い出現頻度の低いデータに長いシンボルを割り当てる。
3. ハフマン木の割り当てたそれぞれのシンボルにデータを登録する。
4. 再度データを読み込み、ハフマン木を検索する。
5. 割り当てられたシンボルに変換し、出力する。

静的ハフマン符号では必ず 2 回データの読み込みが行われるが、データ全体のエントロピーを調査するため、圧縮率を高くすることが可能となる。

動的ハフマン符号（適応型ハフマン符号）の基本アルゴリズムでは以下の手順で圧縮を行う。

1. 空のハフマン木を用意する。
2. データを一つ読み込み、シンボルを割り当てる。
3. 割り当てたシンボルにデータを登録する。同時に出現回数を記録しておく。
4. 割り当てたシンボルと元のデータを出力する。
5. 次のデータを読み込む。
6. ハフマン木を検索する。
7. 登録済みならシンボルを出力。
8. 未登録の場合は、出現頻度の低いリーフのハフマン木を生長させ、新しくシンボルを割り当てる。
9. 割り当てたシンボルにデータを登録する。同時に出現回数を記録しておく。
10. ハフマン木を再構築する。
11. 割り当てたシンボルと元のデータを出力する。
12. 5〜11 を繰り返す。

動的ハフマン符号では必ずしも出現頻度の高いデータに短いシンボルが割り当てられるとは限らないため、圧縮率は静的ハフマン符号に比べ低くなる。しかし、データを 1 度読み込むだけで符号化できるため、高速に処理する事が可能となる。

LZW

辞書式圧縮技術で代表的なアルゴリズムである、LZ78[4] の派生アルゴリズムの一つ。なお、辞書式圧縮には他にも派生アルゴリズム多く存在する。シンボル・ルックアップ・テーブルと呼ばれる辞書を用い、登録された長いデータを短いシンボルに変換する事で圧縮を行う。

辞書式圧縮の基本的な動作は以下のようになる。

1. データを読み込み、辞書を検索する。
2. データが辞書に登録済みであれば、位置や長さを圧縮シンボルとして出力する。
3. データが辞書に登録されていなかった場合は、辞書に登録する。

辞書の検索方式、辞書への登録方式に様々な工夫があり、例えば LZW では以下のような動作を行う。

1. 辞書を初期化する

2. データを読み込み、辞書を検索する。

3. 登録されていたら新たにデータを読み込み、データの長さを伸ばし再検索する。これを繰り返す。

4. 辞書に登録されていなかったら、登録済みまでのデータのシンボルを得る。

5. 新たにシンボルを割り当て、辞書に登録する。

6. 2 に戻る。

できるだけ長いデータ列を一つの圧縮シンボルに割り当てた方が効率が良くなるため、テーブルに登録されるデータ列と、テーブルのエントリー数は次第に成長する。どこまで成長させるかは任意で、最大のテーブルエントリー数（最大シンボルの長さ）などは事前に決定させておく。例えば GIF では最大のシンボルの長さを 12bit と決め、テーブルのエントリー数を 4096 までとしている。テーブルが一杯になった場合は再初期化する事も出来る。再初期化用のシンボルをクリアコードとして割り当てておく。

LCA

ここで、本研究の大きなきっかけとなった、LCA（Lowest Common Ancestor）[16] を紹介する。LCA は文法方式の圧縮とされ、非常に軽い処理で圧縮を行う事が可能となる。LCA は図 2.1 のような二本木ツリーを用いてデータ構造を表現する手法である。入力されたデータ 2 つを表す親ノード（LCA）を求めることで、LCA を圧縮シンボルとして利用することができ、データの圧縮と構造解析を同時に行うことが可能となる。一つの LCA を求める処理は、元となるデータが 2 つ固定であり、処理時間を一定とすることができ、ハードウェア化が容易となる。基本的な動作は以下のようになる。

1. 2 シンボルを 1 つのペアとし（データ・ペアと呼ぶ)、テーブルを検索する。

2. 未登録なら新たにシンボルを割り当て、ペアを登録する。

3. 登録済みならシンボルを得る。

4. 繰り返す。

さらに得られたシンボルをさらにペアにすることで、長いシンボルを短いシンボルへ変換し、圧縮を行う。図 2.1 の (1)(2)(3) のように、階層を辿ることでさらに圧縮を行うことが可能となる。

LCA の大きな特徴は、ペアとなった 2 つのデータしか検索しないため、テーブルの 1 エントリーを検索する時間を一定にすることが可能となる点である。ただし、テーブルの最大エントリー数は不明となる。

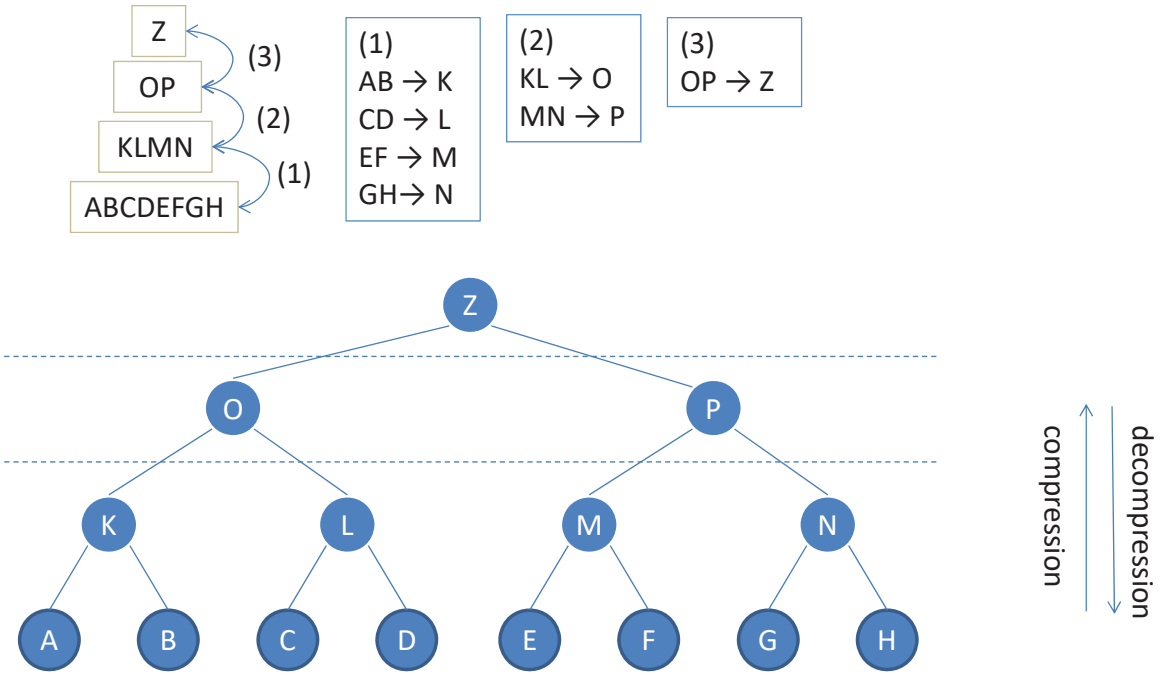


図 2.1: LCA アルゴリズム

2.2.3 LCA-SLT：初期の実装例

LCA をハードウェア圧縮装置に応用した最初の実装例が「LCA-SLT」（LCA - Static Lookup Table）である [17]。

この方式は静的ハフマン符号と同様、まず最初にデータをスキャンしヒストグラムを求め、固定長のテーブルに収まるよう出現頻度の高いヒストグラムのデータ・ペアだけをテーブルに収納し、圧縮を行う方式となっている（図 2.2）。

出現頻度（ヒストグラム）が静的なため、特定のデータ列の圧縮しか効かない。特性の異なるデータ列においては圧縮率が著しく悪化する。

限定的な手法ではあるが、実際に FPGA を用いて実装され、非常に小さな回路によるストリームデータのリアルタイム圧縮・伸張を実証している。また、この段階を踏んだことにより、LCA アルゴリズムのハードウェア化の有用性が確認され、後の動的なヒストグラム生成手法の開発につながる多くの知見が得られている。

2.2.4 ヒストグラムと圧縮

出現頻度の高いデータを、短いシンボルに置き換える事で圧縮することが可能となる。

静的ハフマン符号では、実際に圧縮を行う前にデータの出現頻度を求める。動的ハフマン符号ではハフマン木を生長させながら出現頻度を求め、その出現頻度を元にハフマン木の再構築を行う。LCA においてはデータ・ペアの出現頻度を求める。

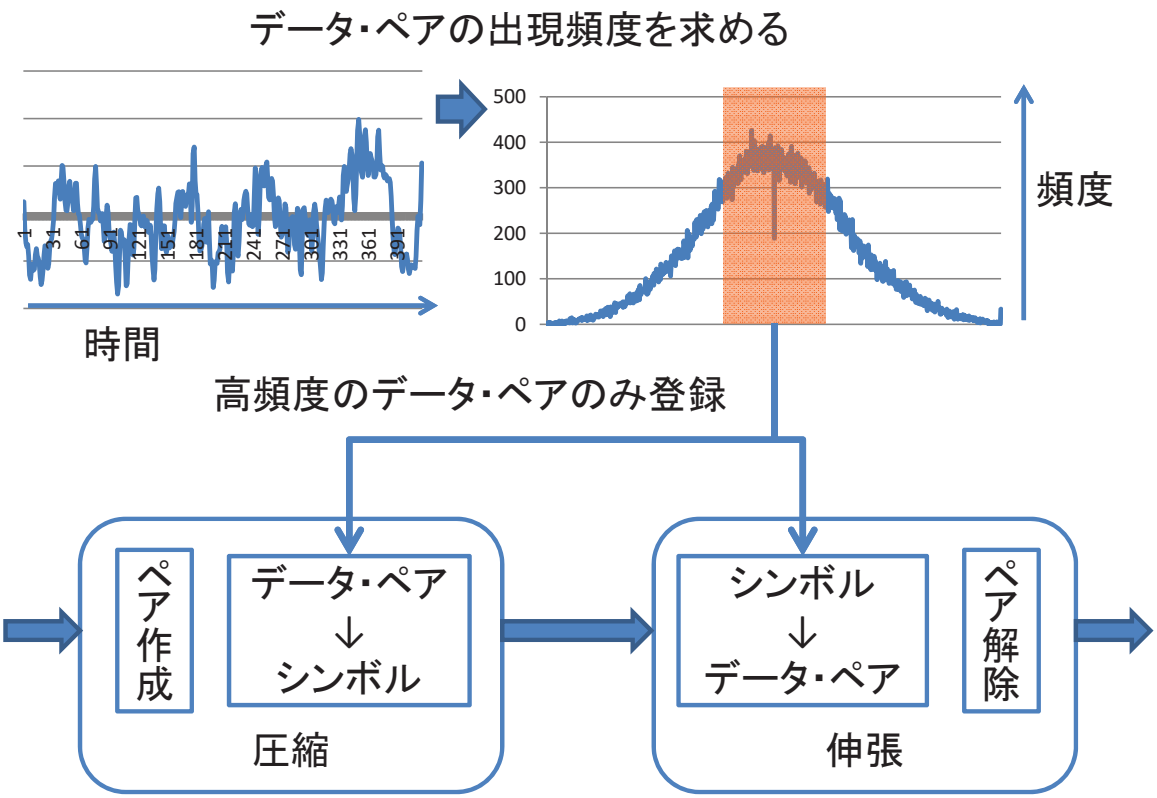


図 2.2: LCA-SLT 動作

これらの方式において、データの出現頻度の分布（ヒストグラム）を求めることが重要となる。完全なヒストグラムを採るにはデータのパターンエントリー数分のカウンターを用意し、現れたすべてのデータを出現個数を数える処理となる。しかしこの方法では、例えば 32bit データの場合、$2^{32} = 4G$ 個 のエントリーが必要となり、現実的では無い。ヒストグラムのエントリー数を減らす必要がある。さらにハードウェア化を想定した場合、エントリー数を固定にする必要がある。

また、ヒストグラムは、ある区間のデータの出現頻度を計測したものであり、時間経過と共に変化する。ヒストグラムの時間変化の概念図を図 2.3 に記す。ある波形のデータをある区間（色）で切り、それぞれの頻度（ヒストグラム）を求めた例となっている。それぞれの区間毎にヒストグラムの形は異なる事がわかる。図 2.3 において、異なる色のヒストグラムを用いて圧縮を行うと、圧縮率は悪化してしまう。

静的ハフマン符号などではデータ全体を精査し、最適なシンボル化を行うことで圧縮率を高めているが、テーブルの量が限定される場合、ヒストグラムの時間変化に追従できず、圧縮率が悪化してしまう。さらに、データを精査するためにはバッファーメモリが必須となり、リアルタイムにヒストグラムを作成することが出来ない。

ヒストグラムのエントリー数が一定の環境で、そのエントリー数を超えるパターンを持つ

データが入力された場合でも、出現頻度の高いデータのヒストグラムを保持し続ける手法の開発が必要となる。また、入力されたデータから、その場でヒストグラムを作成し、データの変化に追従可能な、リアルタイム性も必要となる。これを「動的ヒストグラム生成手法」と呼ぶ。

動的ヒストグラム生成の方法は複数種類提案されているが [18][19]、有名な方法として以下が挙げられる。

- Lossy Counting [20]
- Space Saving [21]

これらの方法を精査すると、アルゴリズムの中にポインタ操作や動的なメモリ操作が含まれていることが分かる。つまり、これらのアルゴリズムはソフトウェアでの実装が想定されており、ハードウェア化は難しい。また、リアルタイム性が考慮されておらず、本件の目的には沿わない。これらの方式を参考に、一定のメモリ量で、一定時間内に、リアルタイムでヒストグラムを作成するハードウェア向けのアルゴリズムが必須となる。

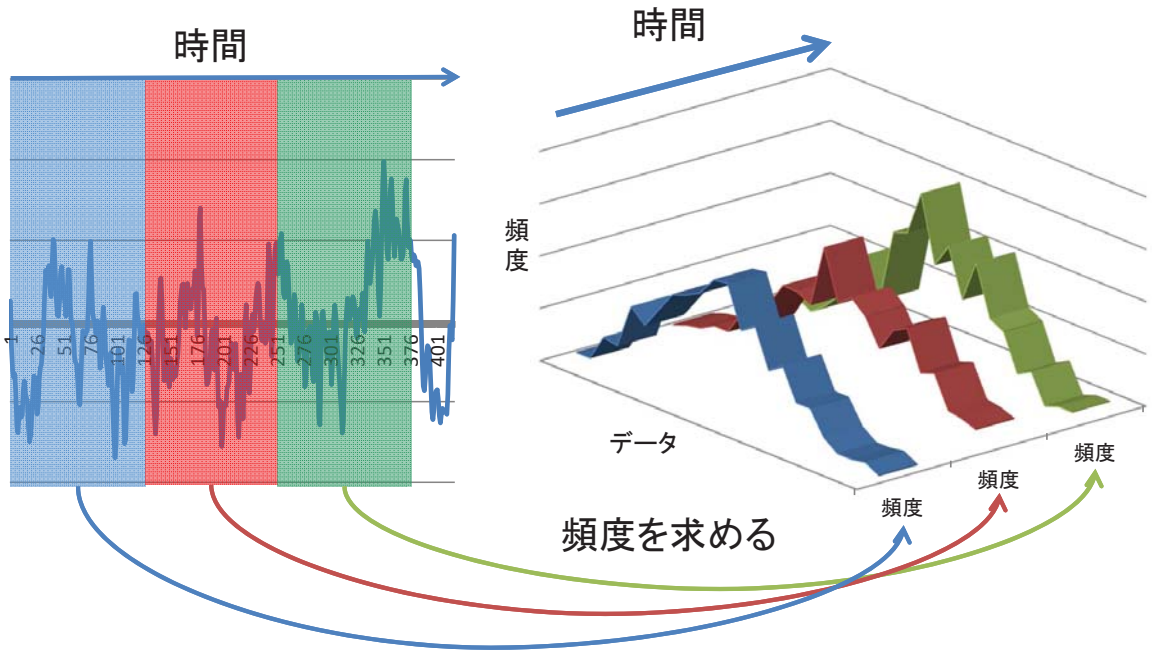


図 2.3: ヒストグラムの変化

2.3 議論

高速伝送系に圧縮技術を適応させるのに必要な要素を考える。

伝送系が高速化されつつある状況で、ネットワークのような通信のみならず、プロセッサ～メモリ間のようなコンポーネント間の伝送系にも圧縮技術を適応させるためには、ソフトウェア処理による圧縮処理では対応しきれない。例えば、プロセッサの処理能力がどんなに向上しても、プロセッサ本体が利用するデータ伝送路の帯域を上回る速度を得ることは不可能である。

ソフトウェアによる圧縮の高速化は、[22] などに例を見ることが出来る。この資料においても、ソフトウェアでは 300～400MByte/s が上限とされており、GByte/s クラスの性能をソフトウェアで達成することは困難である。

このため、伝送速度の高速化に対応するには、圧縮処理のハードウェア化が不可欠となる。

現在の圧縮アルゴリズムにおいて、ハードウェア化を考えた場合、いくつかの問題点が挙げられる。エントロピー符号の方式ではエントロピーを計算する処理の時間が不確定となる。辞書式圧縮は「辞書の大きさと内容が成長する」という特徴がある。

これらはハードウェア化する際、以下の問題点が発生する。

- 処理時間が一定では無い。
- メモリのサイズが不確定。

ソフトウェアによる処理であれば、必要な処理がすべて終わるまで、次のデータを待たせることが可能なため、処理時間を一定とする必要は無い。さらに、メモリのダイナミックな確保・開放が容易であり、メモリの初期化や成長などのタイミングで、その容量を適切に切り出すことが可能である。また、ポインタ操作により、データの入れ替えや割り当てを変えるのも容易である。これらが多くの圧縮アルゴリズムの実装においてソフトウェアをベースにしている所以となる。

性能評価としは、帯域だけでは無く、処理時間（レイテンシ）も考慮しなくてはならない。処理時間が一定では無いアルゴリズムで、流れ続けるデータ（以下ストリームデータ）を一切止めること無く処理するためには、以下のような構成が必要となる。

- 処理のワーストケースの処理時間を求める。
- ワーストケースが発生してもデータを途切れさせない、十分な量のバッファーメモリを用意する。
- ワーストケースが発生する時間の間隔を求める。
- ワーストケースの発生間隔で、用意したバッファーメモリが埋まるだけの性能を持ったハードウェアを用意する。

処理のワーストケースを、ある時間内に完了させるのに十分な性能を持ったハードウェアが必要となるが、通常の処理を行っている間はハードウェアリソースのほとんどが利用されないという状況になる。非常に効率の悪い実装となってしまう。

また、バッファーメモリを使用した場合は、データをメモリに貯める時間も増え、レイテンシが大きくなる。レイテンシの増加はシステムの性能に大きな影響を与える。[5] では、プロセッサの性能不足による性能低下だけでは無く、レイテンシの増加による通信効率の悪化も指摘している。[15] では、圧縮器をハードウェア化したにもかかわらず、レイテンシ増加による性能悪化が示されている。一方的にデータを送るアプリケーション（例えばバックアップテープなど）以外では、相互通信が行われるため、レイテンシの増加が問題となる。

ストリームデータを止めずに、かつバッファーメモリを利用せず処理するためには、処理時間が一定であることも必須となる。すなわち、ベストケースの処理時間=ワーストケースの処理時間であることが条件となる。ハードウェアおいて、処理時間が可変な処理を一定時間で処理するためには、ハードウェアの量が可変になることを意味し、実質実装不可能となる。

また、メモリのサイズが決定されない状態では、要求されるデータに対して実質無制限と呼べる量のメモリが必要となる。処理パイプラインに組み込むには必要となるメモリの量は一定でなければならず、無制限のメモリを想定することは出来ない。実際には考えうる最大のメモリを用意し、それ以上のメモリが要求された場合は初期状態に戻る、といった手法が考えられる。

すなわち、ハードウェア化のためには上記 2 点を解決する事が必須となる。

- 処理時間が一定。
- 必要なメモリのサイズが一定。

この 2 点を満足させることにより、ストリーム処理が可能なハードウェア圧縮を実現する事が可能となると考えられる。

一方、既存の圧縮技術でも上記の条件の両方、または片方を満足させ、ハードウェア化した例がいくつか存在する（表 2.5）。

表 2.5: 圧縮ハードウェアの例

実装	スループット	レイテンシ	備考
IBM velirog 実装 [23]	約 3GByte/s	17 クロック	
Altera OpenCL 実装 [24]	約 3GByte/s	87 クロック	
Intel 89xx [6]	約 20Gbps	不明	
AHA3642 [25]	約 20Gbps	不明	[26] において、可変レイテンシを課題としている。
CAST GZIP IP[27]	数 Gbps ～ 100Gbps 以上	15～2000 クロック	性能はリソースの量や圧縮方式で変化。

レイテンシ固定でストリームデータ処理が可能、パイプラインのストールが一切無い事を保証している実装ではハードウェアの量が大きく、Altera Stratix V A7（62 万ロジックエレメント、94 万フリップフロップ）のような大規模 FPGA の半分以上を占めるインプリメンテー

ション結果が報告されている [24]。ハードウェアの量を制限した実装ではレイテンシが可変となっている。

本研究では、伝送系の経路に埋め込み、圧縮・伸張を行うことが可能なハードウェアの実現を目標とする。伝送系にはある程度のレイテンシを許すアプリケーションのみならず、CPU～メモリ間のようなレイテンシに非常に敏感なアプリケーションも想定している。

以上を満足する、ストリームデータを扱うことが可能な新しい圧縮方式「LCA-DLT」を提案する。以下、実装例を挙げ、性能評価を行う。

第3章　ストリームデータ圧縮技術の開発

3.1　はじめに

提案されるLCA-DLTアルゴリズムは大きく以下2つの基本技術を組み合わせて構成される。

- LCAアルゴリズム
- メモリ固定長なリアルタイム・ヒストグラム作成手法

それぞれの基本技術の設計を行い、ハードウェア実装を行う。

3.2　設計

3.2.1　LCA-DLTのアルゴリズム

LCA-DLTは、LCAに基づくシンボルの変換が行われるが、[16]のLCAアルゴリズムでは総シンボル数（=メモリの量）が未定である。これをある一定の数のメモリで処理可能とするため、動的ヒストグラム生成による選別が行われる。つまり、出現頻度の高いシンボルを残し、出現頻度が低いシンボルは削除する。これにより、ハードウェアが保持しておかなければならないシンボル数を、ある一定数に抑えることが可能となる。プロセッサのキャッシュと似た動作になるが、アドレスが存在しない、データと頻度のみで管理されるテーブルとなる。

LCA-DLTアルゴリズムで使用されるテーブルの各エントリには以下が含まれるとする。

- 「valid」: エントリが有効である事を示す1bitのフラグ
- 「s1,s0」: データ・ペア
- 「count」: データ・ペアの頻度カウンタ

ここで、各情報のbit幅をパラメータとして決める。

- 有効フラグ　　: $1[bit]$
- データ・ペアの幅　: $dw[bit]$、各データは $dw \div 2[bit]$
- 頻度カウンタの幅　: $cw[bit]$
- シンボルの幅　　: $sw[bit]$

テーブルは 2^{sw} エントリの構成となる。これをシンボル・ルックアップ・テーブルと呼ぶ。図 3.1 の構造となる。シンボル・ルックアップ・テーブルの大きさはは

$$(1 + dw + cw) \times 2^{sw}[bit] \tag{3.1}$$

となり、これが必要なメモリの bit 数となる。

典型的な設定例として $dw = 16[bit], sw = 8[bit], cw = 8[bit]$ とすると、

$$(1 + 16 + 8) \times 2^{8}[bit] = 6400[bit] \tag{3.2}$$

が必要なメモリの容量となる。

メモリのアドレス、index の値がそのまま圧縮シンボルとなる。テーブルには新規のデータ・ペアを登録する位置を指す、「登録 index」があり、新しいデータ・ペアが現れる毎に次の空いているエントリを指す。圧縮と伸張両方に同じ構成のシンボル・ルックアップ・テーブルを用意する。

頻度カウンタは $0 \sim 2^{cw} - 1$ の値の範囲を持つ。飽和演算（サチュレーション型）のカウンタとする。

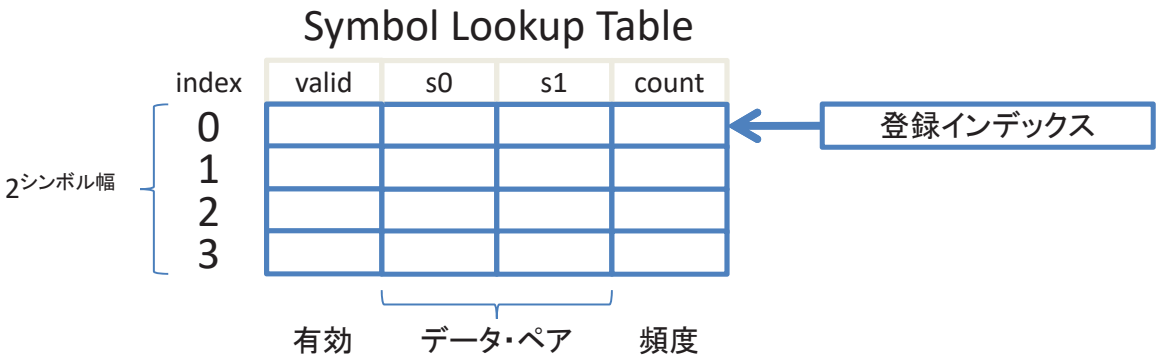


図 3.1: テーブルの構造

以下、このシンボル・ルックアップ・テーブルに圧縮と伸張のルールを割り当て、簡単なデータを流した動作例を考える。

3.2.2 圧縮動作

圧縮器ではシンボル・ルックアップ・テーブルに対し、表 3.1 のルールを割り当てるものとする。ルールを元に動作させた例を表 3.2 に示す。以下の動作となる。

1. 初期状態、テーブルが空の状態から開始。
2. 入力データ「AA」を評価。テーブルに未登録。ルール 2 を適応。登録インデックスの位置に「AA」を登録。count 値を初期化。「AA」を出力。登録インデックスを次の空きエントリに移動。

3. 入力データ「AA」を評価。テーブルに登録済み。ルール 1 を適応。count を+1。圧縮シンボル「0」を出力。

4. 入力データ「BB」を評価。テーブルに未登録。ルール 2 を適応。登録インデックスの位置に「BB」を登録。count 値を初期化。「BB」を出力。登録インデックスを次の空きエントリに移動。

5. 入力データ「CC」を評価。テーブルに未登録。ルール 2 を適応。登録インデックスの位置に「CC」を登録。count 値を初期化。「CC」を出力。登録インデックスを次の空きエントリに移動。

6. 入力データ「DD」を評価。テーブルに未登録。ルール 2 を適応。登録インデックスの位置に「DD」を登録。count 値を初期化。「DD」を出力。空きエントリが無いため、登録インデックスは不定。

7. 入力データ「EE」を評価。テーブルに未登録。テーブルに空きが無い。ルール 3 を適応。全エントリの count を-1。count が 0 となったエントリを削除。登録インデックスを次の空きエントリに移動。この間、処理が止まる（ストール）。

8. 入力データ「EE」を評価。テーブルに未登録。ルール 2 を適応。登録インデックスの位置に「EE」を登録。count 値を初期化。「EE」を出力。登録インデックスを次の空きエントリに移動。

count の値を元に出現頻度が低いシンボルが削除され、出現頻度が高く、最近出現したシンボルがテーブルに残る。テーブルに登録されたデータ・ペアが index に置き換えられ、データ圧縮が行われる。

表 3.1: 圧縮ルール

条件	動作	ルール
入力データ・ペアがテーブルにある	頻度カウンタを+1。index を出力。	ルール 1
入力データ・ペアがテーブルに無い&テーブルに空きがある	頻度カウンタを初期化。データ・ペアを登録インデックスの指す位置に登録。登録インデックスを移動。データ・ペアをそのまま出力。	ルール 2
入力データ・ペアがテーブルに無い&テーブルに空きが無い	全エントリの頻度カウンタを-1。頻度カウンタが 0 になったエントリを削除。テーブルが空くまで繰り返す。	ルール 3

表 3.2: 圧縮動作

時間	入力シンボル	登録 index	テーブル					出力シンボル
			index	valid	s0	s1	count	
1	初期状態	0	0	0			0	
			1	0			0	
			2	0			0	
			3	0			0	
2	AAAABBCCDDEE		0	1	A	A	**1**	AA
		1	1	0			0	
	（ルール 2）		2	0			0	
			3	0			0	
3	AAAABBCCDDEE		0	1	A	A	**2**	0
		1	1	0			0	
	（ルール 1）		2	0			0	
			3	0			0	
4	AAAABBCCDDEE		0	1	A	A	2	BB
			1	1	B	B	**1**	
	（ルール 2）	2	2	0			0	
			3	0			0	
5	AAAABBCCDDEE		0	1	A	A	2	CC
			1	1	B	B	1	
	（ルール 2）		2	1	C	C	**1**	
		3	3	0			0	
6	AAAABBCCDDEE	n/a	0	1	A	A	2	DD
			1	1	B	B	1	
	（ルール 2）		2	1	C	C	1	
			3	1	D	D	1	
7	AAAABBCCDDEE		0	1	A	A	**1**	<stall>
		1	1	0			**0**	
	（ルール 3）		2	0			**0**	
			3	0			**0**	
8	AAAABBCCDDEE		0	1	A	A	1	EE
			1	1	E	E	**1**	
	（ルール 2）	2	2	0			0	
			3	0			0	

3.2.3 伸張動作

伸張器ではシンボル・ルックアップ・テーブルに対し、表 3.3 のルールを割り当てるものとする。ルールを元に動作させた例を表 3.4 に示す。以下の動作となる。

1. 初期状態、テーブルが空の状態から開始。
2. 入力データ「AA」を評価。非圧縮シンボル。ルール 2 を適応。登録インデックスの位置に「AA」を登録。count 値を初期化。「AA」を出力。登録インデックスを次の空きエントリに移動。
3. 入力データ「0」を評価。圧縮シンボル。ルール 1 を適応。count を+1。登録されているデータ・ペア「AA」を出力。
4. 入力データ「BB」を評価。非圧縮シンボル。ルール 2 を適応。登録インデックスの位置に「BB」を登録。count 値を初期化。「BB」を出力。登録インデックスを次の空きエントリに移動。
5. 入力データ「CC」を評価。非圧縮シンボル。ルール 2 を適応。登録インデックスの位置に「CC」を登録。count 値を初期化。「CC」を出力。登録インデックスを次の空きエントリに移動。
6. 入力データ「DD」を評価。非圧縮シンボル。ルール 2 を適応。登録インデックスの位置に「DD」を登録。count 値を初期化。「DD」を出力。空きエントリが無いため、登録インデックスは不定。
7. 入力データ「EE」を評価。非圧縮シンボル。テーブルに空きが無い。ルール 3 を適応。全エントリの count を-1。count が 0 となったエントリを削除。登録インデックスを次の空きエントリに移動。この間、処理が止まる（ストール）。
8. 入力データ「EE」を評価。非圧縮シンボル。ルール 2 を適応。登録インデックスの位置に「EE」を登録。count 値を初期化。「EE」を出力。登録インデックスを次の空きエントリに移動。

count の値を元に出現頻度が低いシンボルが削除され、出現頻度が高く、最近出現したシンボルがテーブルに残る。圧縮されたシンボルが指す index のデータ・ペアを出力することにより、圧縮前の元データが得られる。

表 3.3: 伸張ルール

条件	動作	ルール
入力シンボルが圧縮シンボル	頻度カウンタを+1。シンボルが指す index のデータ・ペアを出力。	ルール 1
入力シンボルが非圧縮シンボル&テーブルに空きがある	頻度カウンタを初期化。データ・ペアを登録インデックスの指す位置に登録。登録インデックスを移動。データ・ペアをそのまま出力。	ルール 2
入力シンボルが非圧縮シンボル&テーブルに空きが無い	全エントリの頻度カウンタを-1。頻度カウンタが 0 になったエントリを削除。テーブルが空くまで繰り返す。	ルール 3

表 3.4: 伸張動作

時間	入力シンボル	登録 index	テーブル					出力シンボル
			index	valid	s0	s1	count	
1	初期状態	0	0	0			0	
			1	0			0	
			2	0			0	
			3	0			0	
2	AA0BBCCDDEE		0	1	A	A	**1**	AA
		1	1	0			0	
	（ルール 2）		2	0			0	
			3	0			0	
3	AA0BBCCDDEE		0	1	A	A	**2**	AA
		1	1	0			0	
	（ルール 1）		2	0			0	
			3	0			0	
4	AA0BBCCDDEE		0	1	A	A	2	BB
			1	1	B	B	**1**	
	（ルール 2）	2	2	0			0	
			3	0			0	
5	AA0BBCCDDEE		0	1	A	A	2	CC
			1	1	B	B	1	
	（ルール 2）		2	1	C	C	**1**	
		3	3	0			0	
6	AA0BBCCDDEE	n/a	0	1	A	A	2	DD
			1	1	B	B	1	
	（ルール 2）		2	1	C	C	1	
			3	1	D	D	1	
7	AA0BBCCDDEE		0	1	A	A	**1**	<stall>
		1	1	0			**0**	
	（ルール 3）		2	0			**0**	
			3	0			**0**	
8	AA0BBCCDDEE		0	1	A	A	1	EE
			1	1	E	E	**1**	
	（ルール 2）	2	2	0			0	
			3	0			0	

3.3 実装

3.3.1 シンボル変換モジュール

LCA のシンボル変換 [16] の動作をハードウェア化する。2 : 1 のシンボル変換を行うハードウェアとなる。圧縮では 2 → 1、伸張では 1 → 2 の変換となり、ハードウェアの形態が異なる。

- 圧縮（図 3.2）
  圧縮処理では入力されたデータ・ペアと、テーブルに登録されているデータ・ペアの比較回路となる。テーブルの全検索ロジックとなるため、CAM（Content-Addressable Memory）構成のハードウェアとなる。現在の FPGA では CAM が搭載されていないため、ロジックでの実装となる。データ・ペアがマッチしたテーブルのインデックスがシンボルとなる。なお、一つのデータが複数のエントリに同時にヒットする状況は無いため、シングル・マッチの CAM となる。

- 伸張（図 3.3）
  伸張処理では入力されたシンボルをデータ・ペアに変換する処理となる。シンボルがメモリのインデックスとなるため、単純なルックアップ・テーブル（RAM）となる。

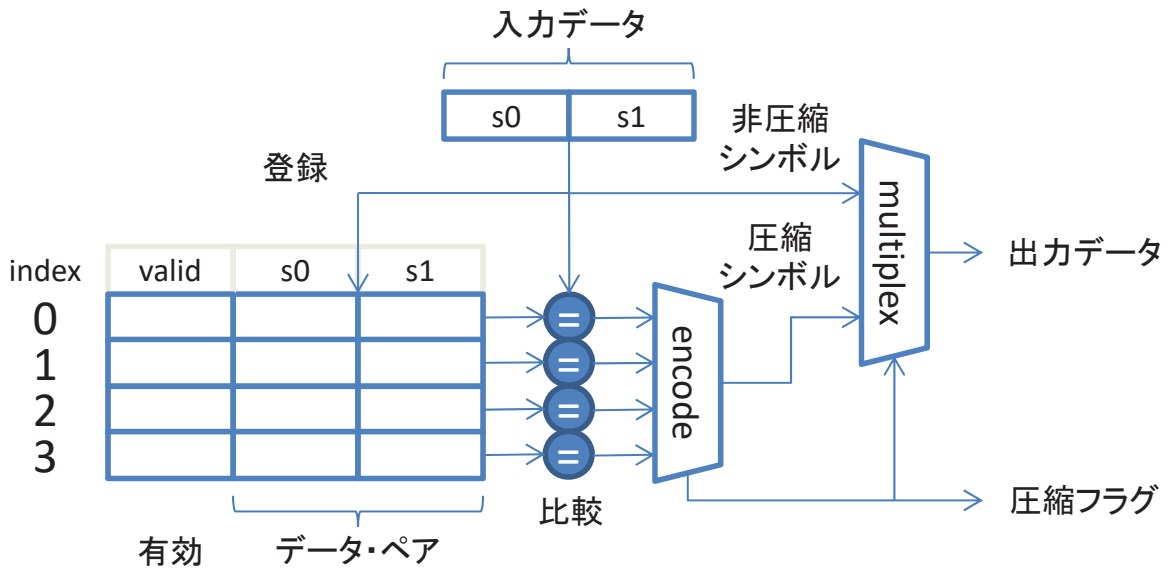


図 3.2: 圧縮テーブル・サーチ・ロジック（CAM）

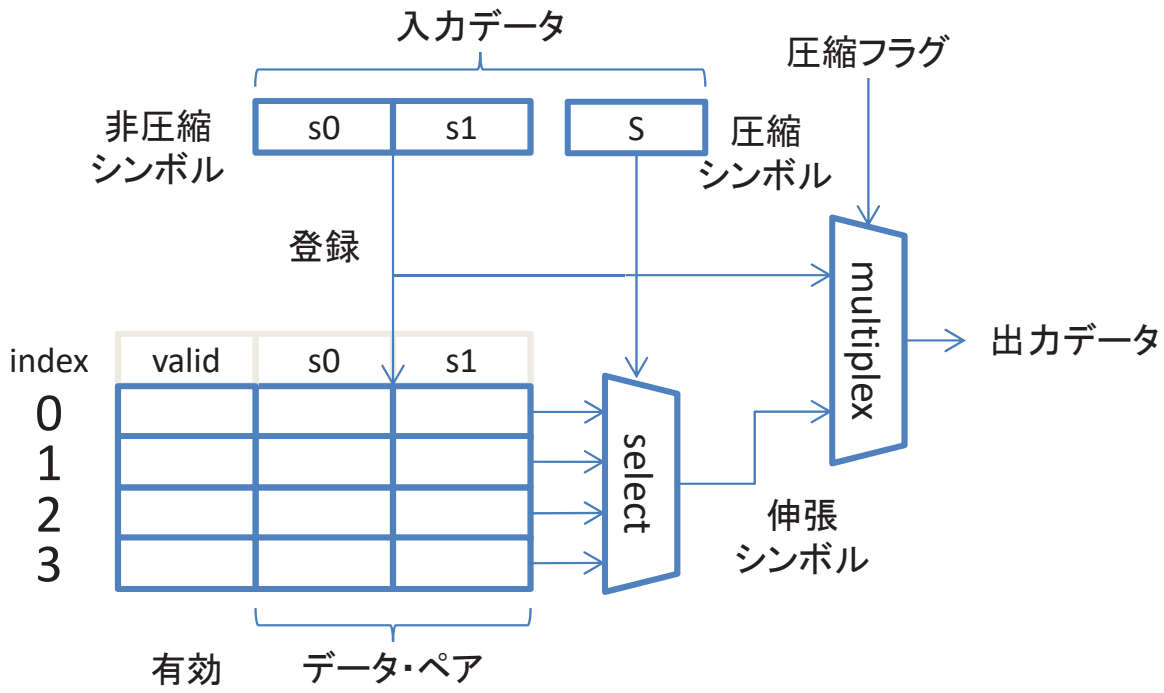


図 3.3: 伸張テーブル・ルックアップ・ロジック（RAM）

3.3.2 動的ヒストグラム生成モジュール

動的ヒストグラム生成の方法は圧縮と伸張ともに同じ動作となっている（表 3.1, 表 3.3）。圧縮・伸張の「ルール 3」（テーブルの空きを作る処理）を、逐次実行するか、並列実行するかで実装が異なる。

- 逐次実行、シリアル実装（図 3.4）
  テーブルの count を減算し、空きを探す処理を、1 エントリ毎に逐次的に実行する。1 クロックで 1 エントリの処理となる。テーブルの空きを探す時間（=ストール時間）は長くなる。ストール時間の最悪値は

$$\text{テーブルのエントリ数} \times \text{頻度カウンタの最大値} = 2^{sw} \times 2^{cw}[clock] \tag{3.3}$$

  となる。count を RAM で実装するため、回路規模は小さくなる。

- 並列実行、パラレル実装（図 3.5）
  テーブルの count を減算し、空きを探す処理を、全エントリ同時実行する。1 クロックで全エントリの処理となる。テーブルの空きを探す時間（=ストール時間）が短くなる。ストール時間の最悪値は

$$\text{頻度カウンタの最大値} = 2^{cw}[clock] \tag{3.4}$$

となる。count をレジスタファイル（REG）で実装し、演算ロジックを並列にするため、回路規模が大きくなる。

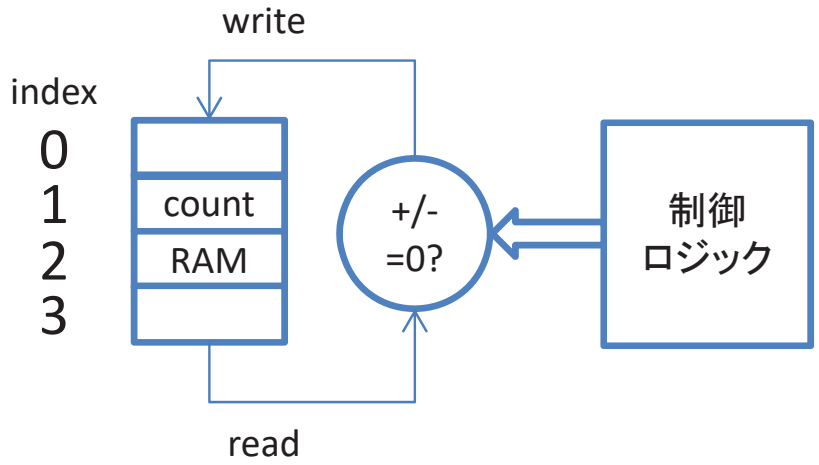


図 3.4: 逐次実行ロジック（RAM）

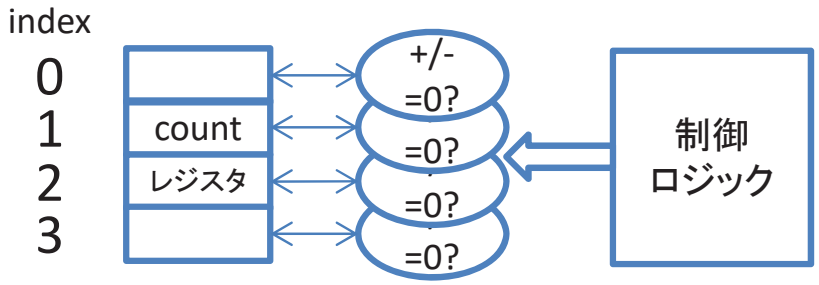


図 3.5: 並列実行ロジック（REG：レジスタファイル）

3.3.3 全体の構成

圧縮・伸張とシリアル実装・パラレル実装の組み合わせで、図 3.6、図 3.7、図 3.8、図 3.9 の 4 つの実装パターンとなる。表 3.5 にそれぞれの特徴をまとめる。CAM と REG は回路規模が大きく、RAM は小さい。REG 構成はストール時間が短いが、RAM は長い。これらの組み合わせにより、回路規模とストール時間に差が生まれる。なお、図ではパイプラインステージが追加されている。若干の周波数向上がなされている。

表 3.5: 実装の組み合わせ

種類	動的ヒストグラム生成	構成	回路規模	ストール時間	構成図
圧縮	シリアル実装（図 3.4）	CAM-RAM	中	長い	図 3.6
（図 3.2）	パラレル実装（図 3.5）	CAM-REG	大	短い	図 3.7
伸張	シリアル実装（図 3.4）	RAM-RAM	小	長い	図 3.8
（図 3.3）	パラレル実装（図 3.5）	RAM-REG	中	短い	図 3.9

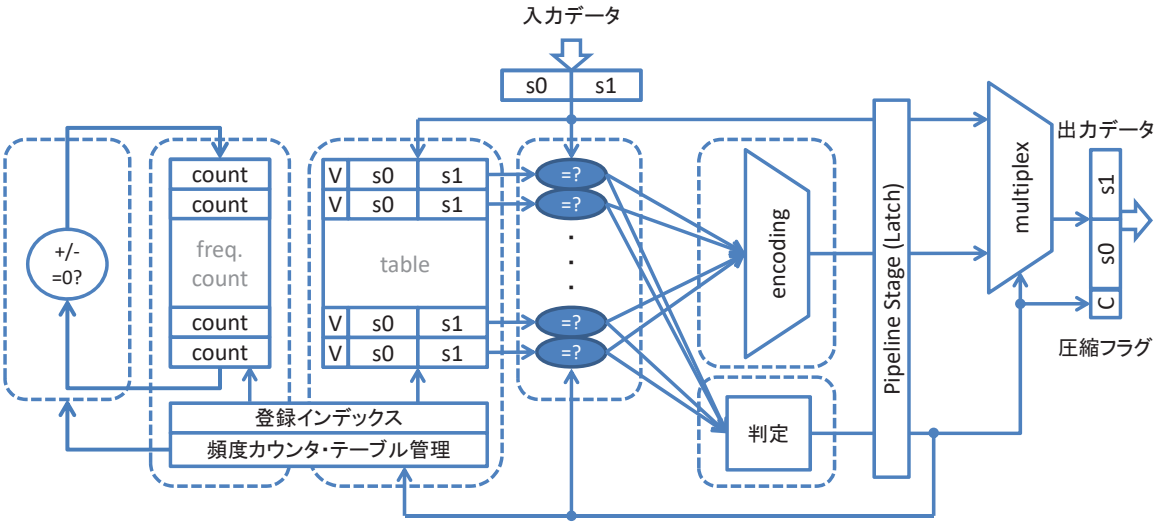


図 3.6: 圧縮 シリアル実装 CAM-RAM 構成

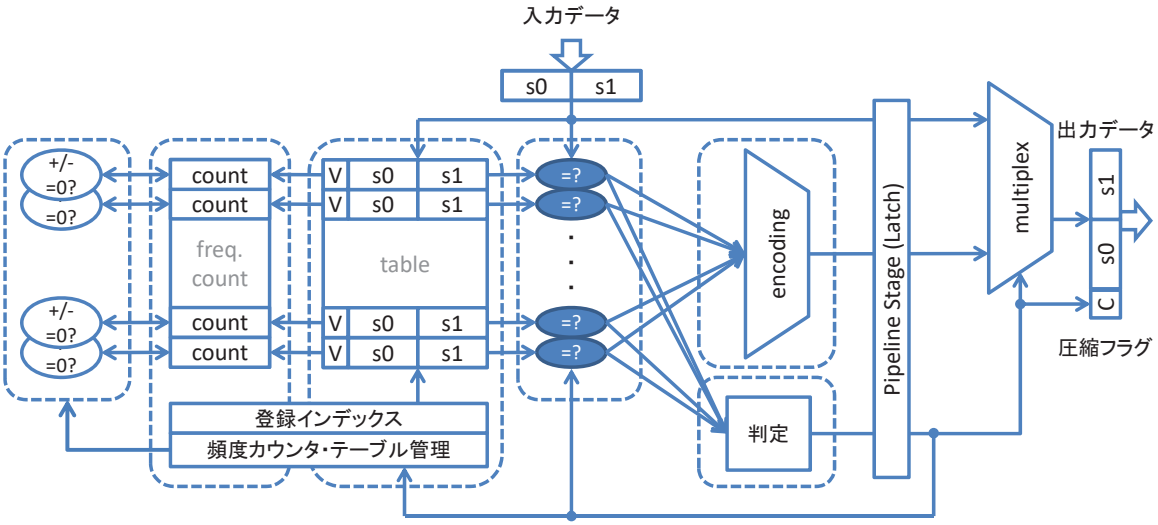


図 3.7: 圧縮 パラレル実装 CAM-REG 構成

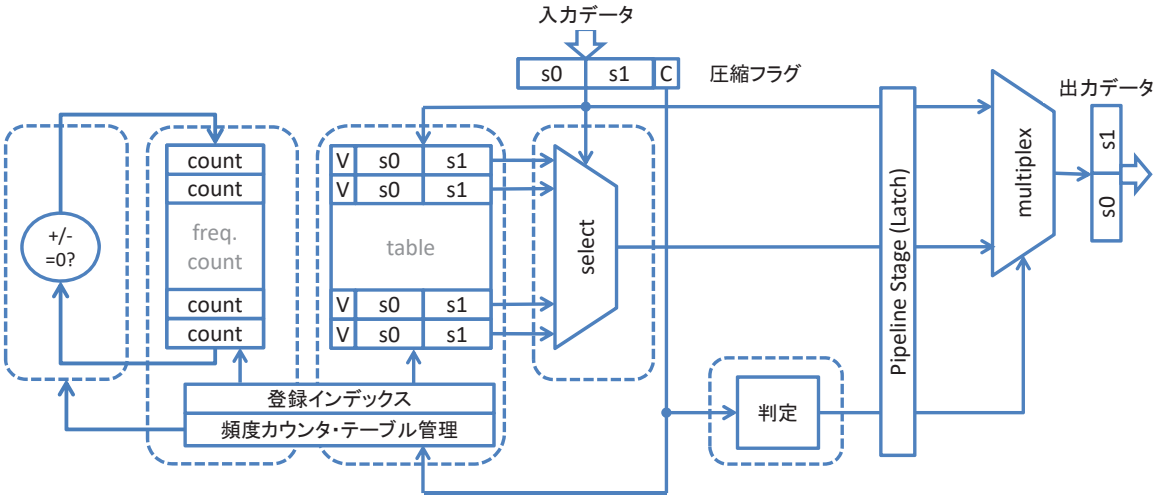


図 3.8: 伸張 シリアル実装 RAM-RAM 構成

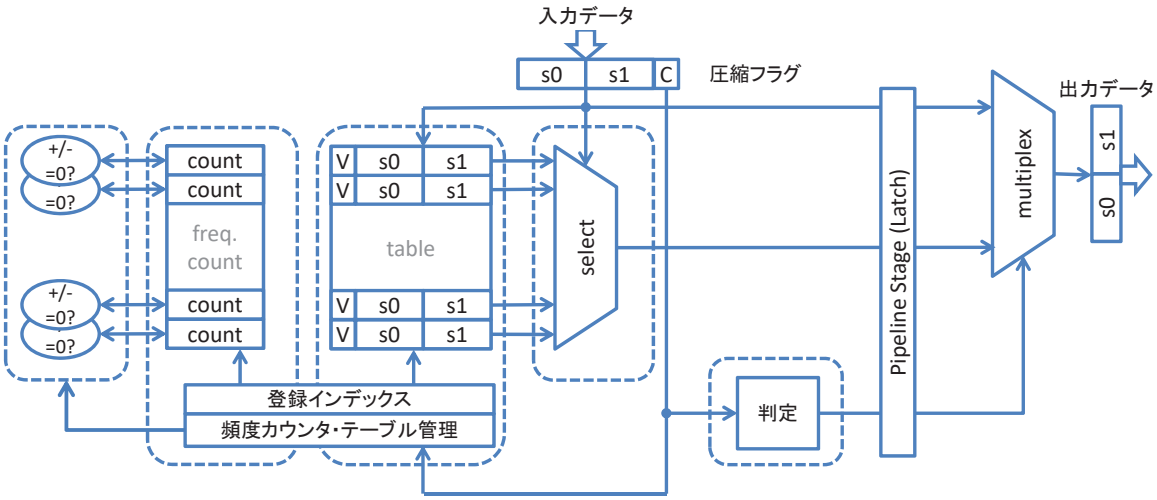


図 3.9: 伸張 パラレル実装 RAM-REG 構成

3.3.4　カスケード接続による圧縮率制御

LCA による圧縮では 2 : 1 変換のため、最高 50 %までの圧縮となる。しかし、LCA のツリー構造を辿ることで、圧縮率を高めることが可能である（図 3.10）。これにより圧縮率とハードウェアの量のバランスをユーザが選ぶことが可能となる。

出力シンボルは、未出現のシンボルでなくてはならないため、1bit をフラグ・ビットが追加される。このため、カスケード段数が増えるとシンボルの幅が増える。元のシンボルに対するフラグ・ビットの割合が増えるため、圧縮率に影響が出る。

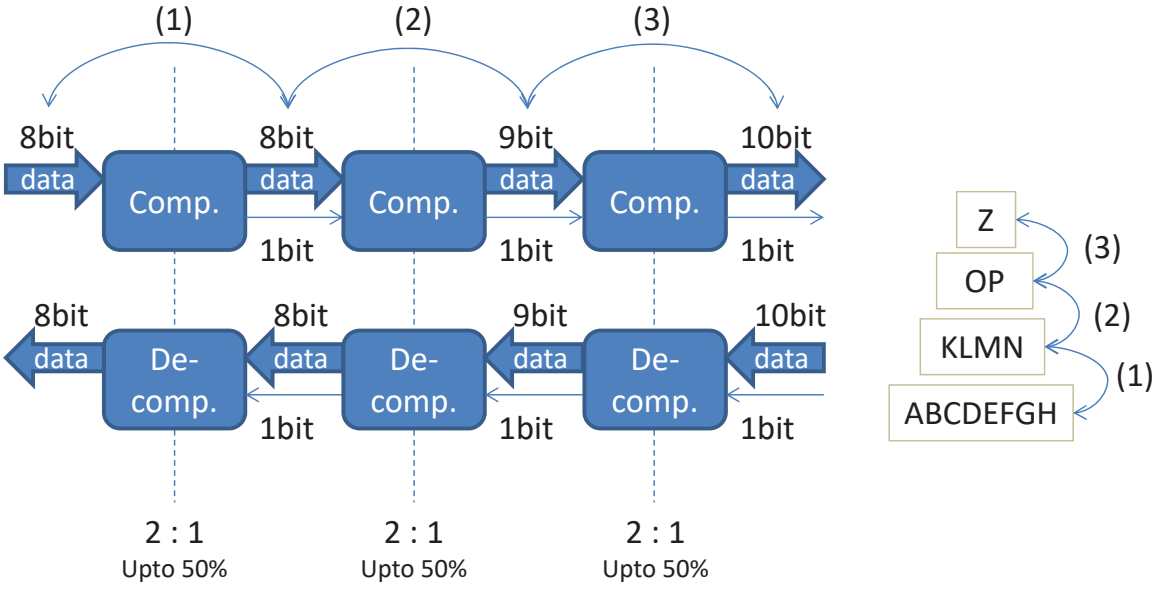


図 3.10: カスケードによる圧縮率制御

3.4　評価

評価項目として、以下の項目を評価する。
なお、パラメータは $dw = 16[bit], sw = 4, 5, 6, 7, 8[bit], cw = 8[bit]$ とした。

- 圧縮率

$$\text{圧縮率} = \frac{\text{圧縮後のサイズ}}{\text{元のサイズ}} \tag{3.5}$$

圧縮率はテーブルのエントリ数とカスケード段数の影響を受けると考えられる。代表的な数値の圧縮率を評価する。値が少ないほどよい。圧縮出来ない場合は圧縮率が 100 %を超える。

- ストール率

$$\text{ストール率} = \frac{\text{ストールしたクロック数}}{\text{全体のクロック数}} \tag{3.6}$$

  テーブルの空きを探す「ルール 3」の影響を評価する。値が少ないほどよい。処理のストールにより性能が低下する。

- ハードウェアの規模と速度
  実際の FPGA に実装した場合のコンパイル結果を評価する。
  - ターゲットデバイス：Xilinx Artix7 XC7A200T-1FBG676C
  - 開発ツール　　：Vivado2015.2

ベンチマークに使用したデータは [28] から 10MByte のファイル（XML, English Test, MIDI pitch, Protein, DNA, Linux source）データを使用する。

3.4.1 圧縮率の評価

圧縮率の実行結果を図 3.11 に示す。以下の傾向が見られる事がわかる。

- テーブルのエントリ数が増えると圧縮率は良くなる傾向がある。
- カスケード段数が増えると、一部（XML と DNA データ）では良くなるが、多くデータで悪化が見られる。

DNA のような例外的な場合では 40 %、一般的なデータでは概ね 60 %〜80 %程度の圧縮率が得られる。一方段数の追加による圧縮率の改善は見られなかった。多くのデータが局所的には 2 ペアまでの冗長性しか無かったためと考えられる。

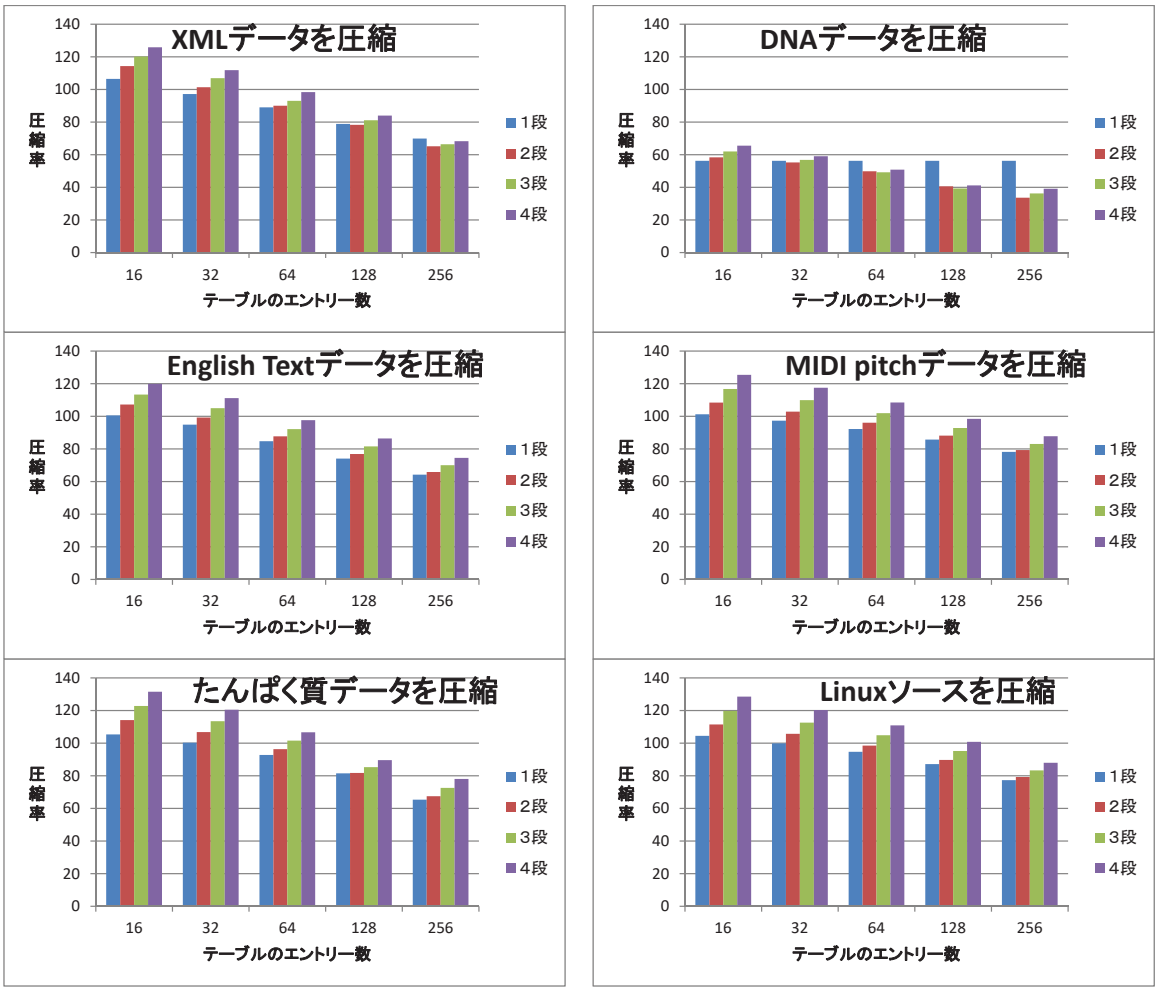


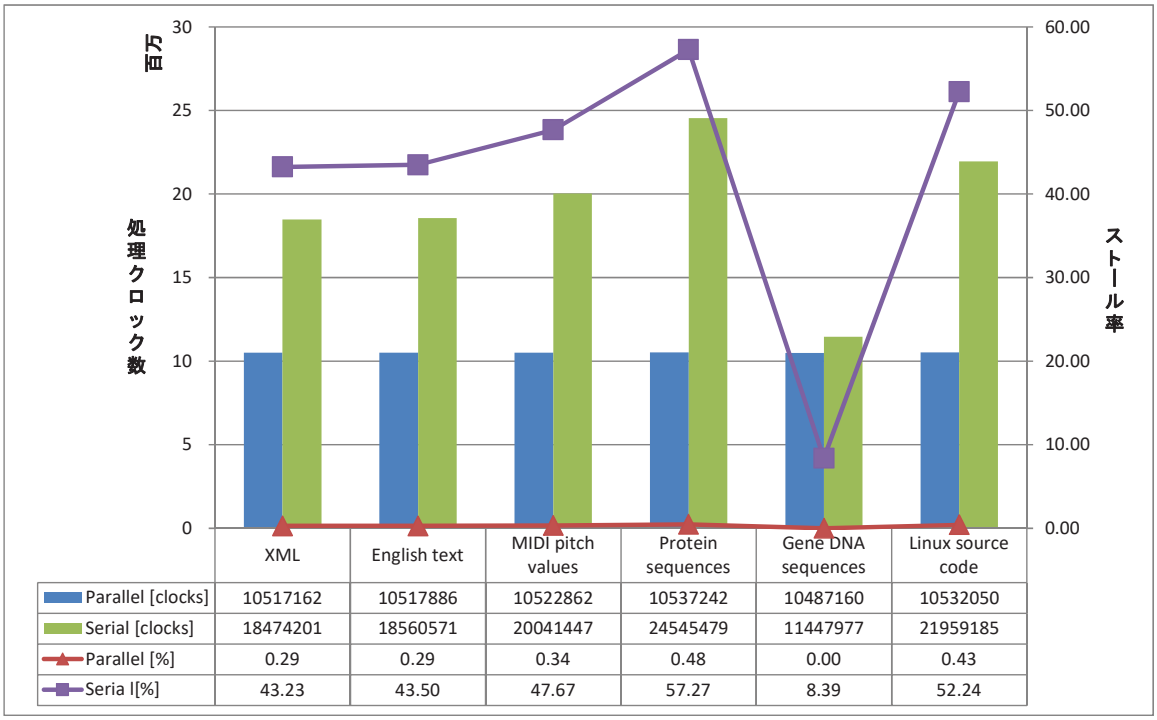
図 3.11: 圧縮率
（横軸：テーブルのエントリ数、
縦軸：圧縮率 [%]）

3.4.2 ストール率の評価

ストール率の結果は図 3.12 となった。元のデータが 10M 個なため、ストール率が 0 %の理想状態の時に、10M クロック（10,487,160 クロック）になる。

- シリアル実装ではストール率が高く、本来の処理時間の倍以上かかっている（ストール率が 50 %を超えている）例もある。
- パラレル実装はシリアル実装に比べ、ストール率は低い。しかし、0 %にはならない。

処理のストールが発生し、処理時間が増えていることが分かる。特にシリアル実装で顕著である。パラレル実装でも 0 では無いため、必ずデータが滞ることになる。



	XML	English text	MIDI pitch values	Protein sequences	Gene DNA sequences	Linux source code
Parallel [clocks]	10517162	10517886	10522862	10537242	10487160	10532050
Serial [clocks]	18474201	18560571	20041447	24545479	11447977	21959185
Parallel [%]	0.29	0.29	0.34	0.48	0.00	0.43
Seria l[%]	43.23	43.50	47.67	57.27	8.39	52.24

図 3.12: ストール率
（棒グラフ：縦軸左：処理クロック数 [10^6 clock]
折れ線グラフ：縦軸右：ストール率 [%]）

3.4.3 ハードウェア・リソースの評価

FPGA への実装結果は図 3.13 となった。

- パラレル実装はシリアル実装に比べ、LUT 比で約 2 倍、FlipFlops 比で約 1.5 倍のリソースを消費する。
- ハードウェア・リソース量は段数に比例する。
- 周波数は段数・実装の影響を受けない。

予想通り、シリアル実装よりもパラレル実装の方が回路規模は大きくなる。回路規模とストール率（処理時間）とのトレードオフとなる。周波数は段数・方式の影響を受けず、約 100MHz 程度となっている。これは速度を決めているボトルネックが他の所にあることを示唆している。

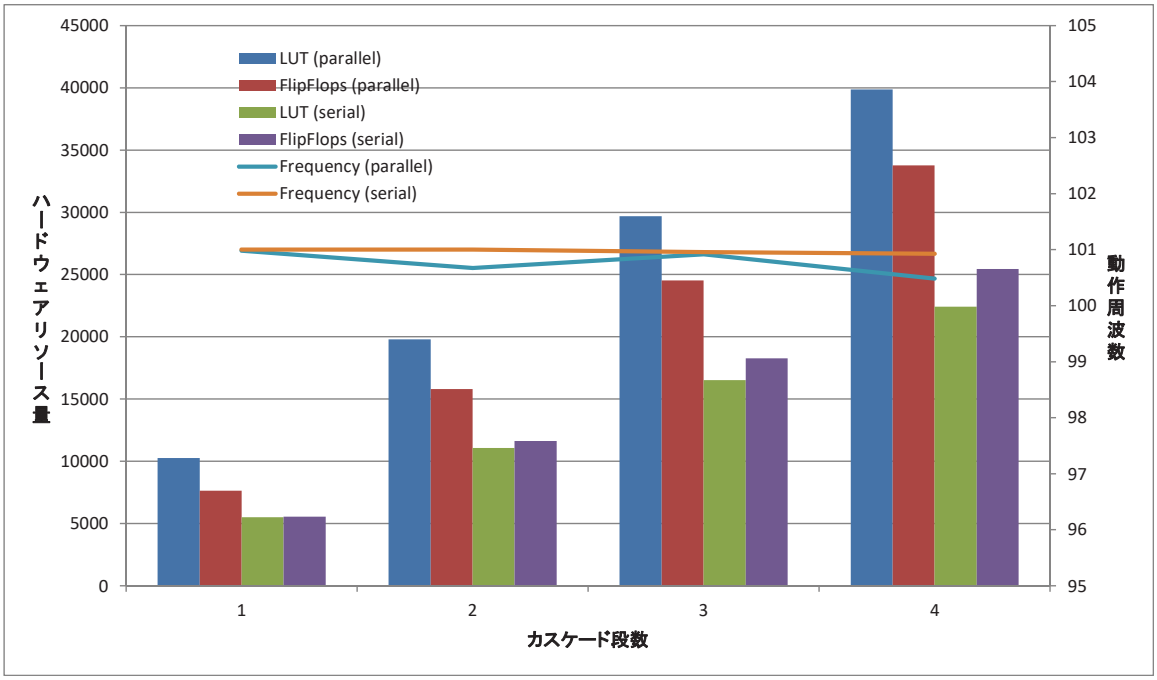


図 3.13: ハードウェア・リソース量
（横軸：段数、
棒グラフ：縦軸左：リソース量、
折れ線グラフ：縦軸右：周波数 [MHz]）

3.4.4 議論

LCA-DLT アルゴリズムにより、データのインライン圧縮の実現の可能性が見られた。スループットは約 100MHz で、100M シンボル/秒となる。データ・ペアの幅が 16bit の場合、1.6Gbps の圧縮・伸張が可能となる。レイテンシは低く抑えられ、この時点での実装では 2 クロックとなっている。

圧縮率は 60〜80 %程度となった。

一方、重大な問題が残っている。処理のストールが存在する現在のアルゴリズムでは、研究の目的の一つである、「処理時間一定」を満足しておらず、ストリームデータ処理が不可能である。この段階ではストリームデータ圧縮・伸張を実現したとは言えない。

また、100MHz という周波数は、近年の FPGA においては遅いと言える。伝送系が 10Gbps のオーダーとなっているため、高速化が必須である。

次章よりこの問題を解決する方式を提案する。

3.5 まとめ

本章では LCA-DLT の基本となるアルゴリズムの開発・実装を行った。LCA アルゴリズムを用い、シンボル変換の時間を一定とし、ハードウェア実装が可能となった。これに動的ヒストグラム生成手法を組み合わせ、LCA-DLT アルゴリズムの開発を行った。LCA-DLT アルゴリズムを用い、リアルタイム性のある可逆圧縮・伸張が可能となることを示し、圧縮率の評価を行った。ハードウェアの構成は複数種類考えられ、ハードウェアの量・性能・圧縮率の間でトレードオフが可能となった。

しかし、本章の構成では処理のストールが存在し、ストリームデータ圧縮・伸張が不可能である。また、動作周波数は現在の FPGA としては低い。これらは次章の課題となる。

第4章　Lazyなテーブル管理技法によるストリームデータ圧縮の高速化

4.1　はじめに

前章で開発された圧縮・伸張ハードウェアでは、データ処理が滞るストールが発生し、ストリームデータ圧縮・伸張が出来ない。この問題を解決し、全くデータが止まらない、真の意味のストリームデータ圧縮・伸張ハードウェアを提案する。

この章では以下の技術が提案される。

- Lazy なテーブル管理
- 削除インデックスの導入

それぞれの設計を行い、ハードウェア実装を行う。

4.2　設計

4.2.1　Lazy なテーブル管理手法

前章 LCA-DLT アルゴリズムのルールの表、表 3.1、表 3.3 において、処理のストールが発生するのは共に「ルール 3」の条件となる。入力データが新規、もしくは未圧縮で、シンボル・ルックアップ・テーブルに未登録であり、テーブルに新しく登録しなければならない状況で、かつ、テーブルに空いたエントリが無く、エントリが空くまで頻度カウンタを減算し続ける処理が行われる。

シリアル実装とパラレル実装でストールの時間が異なり、また、ストール条件が発生した時点でのテーブルの状態によってもストール時間が異なる。この不確定なストール時間により、ストリームデータ圧縮・伸張は不可能となっている。

ストールを回避するため、問題となる「ルール 3」を見直す。シンボル・ルックアップ・テーブルに空きが無い状態で、登録を行うことがストールの要因であるため、その要因そのものをなくすことを考える。すなわちルール 3 を、空きが無い状態で登録が必要な場合、登録を諦め圧縮処理を行わない、という動作に置き換える。登録待ちが発生しないため、ストールは発生しない。

この方式を「Lazy なテーブル管理手法」「Lazy 方式」と呼ぶ。

これに対し、前章の圧縮方式を、すべてのシンボルが圧縮される方式のため「Full 方式」と呼ぶこととする。

4.2.2 削除インデックスの導入

4.2.1 節の手法だけでは、ルールがすべて置換もしくは登録処理のみとなり、頻度カウンタの減算及びエントリの削除処理が行われなくなる。このままではシンボル・ルックアップ・テーブルのエントリの使用量が増える一方で、エントリの置き換えが起こらなくなってしまう。エントリの置き換えが起きなければ、入力データの傾向が変わった場合に対応できず、圧縮率が出来なくなる。

そこで、「削除インデックス」を導入し、動的にテーブルのエントリを削除する手法を考える。削除インデックスはテーブルの削除候補を指し、入力データが来る毎に以下の処理を行う。

- 頻度カウンタの減算。
- エントリの評価。頻度カウンタが 0 ならエントリを削除。
- 削除インデックスを移動。

この処理を「タスク 2」として新しく導入し、通常の置換・登録処理を「タスク 1」とする。

なお、「タスク 2」のエントリ操作は以下の条件においては実行されない。削除インデックスが単純に次のエントリに移動する。

- 削除インデックスが指すエントリが無効だった場合。
- 削除インデックスとテーブル検索が重なった場合。

全エントリを定期的に巡回させることにより、使用頻度の低いエントリが削除されていく。この場合頻度カウンタは「テーブルに存在できる時間」を表す値となる。頻度カウンタの値が $count$ だった場合、

$$\text{テーブルに存在できる時間} = 2^{sw} \times count[symbol] \tag{4.1}$$

の間テーブルに存在し続ける。この時間以内に一度も利用されなかった（登録されたデータ・ペアが現れなかった）場合は使用頻度が低いと判断され、シンボル・ルックアップ・テーブルから削除される。削除され空いたエントリは他のシンボルに再利用される。

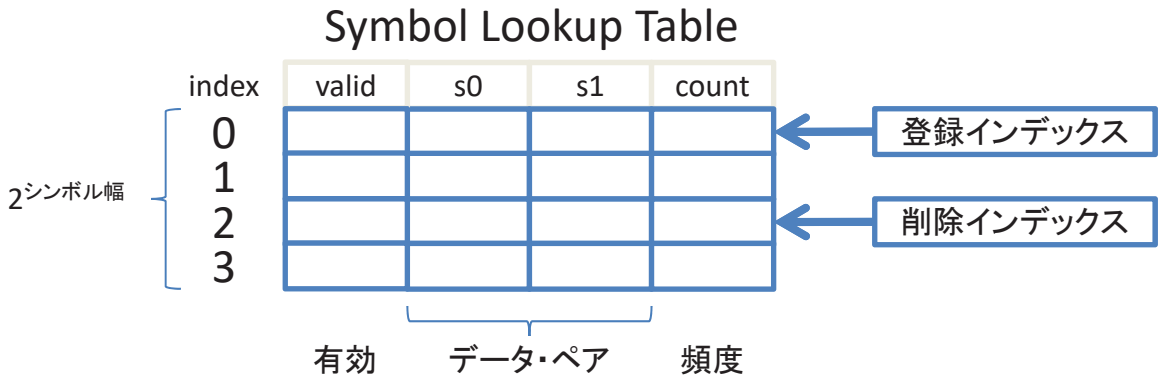


図 4.1: テーブルの構造（lazy）

4.2.3 圧縮動作

圧縮器ではシンボル・ルックアップ・テーブルに対し、表 4.1 のルールを割り当てるものとする。「タスク 1」は表 3.1 とほぼ同等だが、ストールを回避するため、ルール 3 が異なる。また、「タスク 1」と「タスク 2」は同時に実行される。ルールを元に圧縮動作させた例を表 4.2 に示す。以下の動作となる。

1. 初期状態。テーブルにデータが登録され、空きが無い状態。登録インデックスは不定。

2. 入力データ「AC」を評価。テーブルに登録済み。ルール 1 を適応。count を+1。圧縮シンボル「2」を出力。削除インデックスが指す count を-1。count が 0 なのでエントリを削除。登録インデックスを空きエントリに移動。削除インデックスを次のエントリに移動。

3. 入力データ「DC」を評価。テーブルに未登録。ルール 2 を適応。登録インデックスの位置に「DC」を登録。count 値を初期化。「DC」を出力。削除インデックスが指す count を-1。削除インデックスを次のエントリに移動。空きエントリが無いため、登録インデックスは不定。

4. 入力データ「DB」を評価。テーブルに未登録。テーブルに空きが無い。ルール 3 を適応（Lazy）。「DB」を出力。削除インデックスが指す count を-1。削除インデックスを次のエントリに移動。

5. 入力データ「DB」を評価。テーブルに未登録。削除インデックスが指す count を-1。count が 0 なのでエントリを削除。ルール 2 を適応。空いたエントリに新たに「DB」を登録。「DB」を出力。削除インデックスを次のエントリに移動。空きエントリが無いため、登録インデックスは不定。

6. 入力データ「DB」を評価。テーブルに登録済み。ルール 1 を適応。count を+1。圧縮シンボル「0」を出力。削除インデックスが指す count を-1。count が 0 なのでエントリ

を削除。登録インデックスを空きエントリに移動。削除インデックスを次のエントリに移動。

4. と 5. で同じ「DB」が出力されているが、4. はテーブルに空きが無く登録できないため、圧縮を諦めた動作となる。この動作が Lazy 方式の特徴となる。

表 4.1: 圧縮ルール (lazy)

タスク	条件	動作	ルール
タスク 1	入力データ・ペアがテーブルにある	頻度カウンタを+1。index を出力。	ルール 1
	入力データ・ペアがテーブルに無い＆テーブルに空きがある	頻度カウンタを初期化。データ・ペアを登録インデックスの指す位置に登録。登録インデックスを移動。データ・ペアをそのまま出力。	ルール 2
	入力データ・ペアがテーブルに無い＆テーブルに空きが無い	なにもしない。データ・ペアをそのまま出力。	ルール 3
タスク 2	無条件	削除 index が指す頻度カウンタを-1。頻度カウンタが 0 ならエントリを削除、テーブルに空きを作る。削除 index を移動。	

表 4.2: 圧縮動作 (lazy)

時間	入力シンボル	登録 index	テーブル					削除 index	出力シンボル
			index	valid	s0	s1	count		
1	初期状態	n/a	0	1	A	A	1		
			1	1	B	B	1	1	
			2	1	A	C	3		
			3	1	B	D	4		
2	ACDCDBDBDB		0	1	A	A	1		2
		1	1	0			**0**		
	（ルール 1）		2	1	A	C	**4**	2	
			3	1	B	D	4		
3	ACDCDBDBDB	n/a	0	1	A	A	1		DC
			1	1	D	C	**1**		
	（ルール 2）		2	1	A	C	**3**		
			3	1	B	D	4	3	
4	ACDCDBDBDB	n/a	0	1	A	A	1	0	DB
			1	1	D	C	1		
	（ルール 3,lazy）		2	1	A	C	2		(lazy)
			3	1	B	D	**3**		
5	ACDCDBDBDB	n/a	0	1	D	B	**1**		DB
			1	1	D	C	1	1	
	（ルール 2）		2	1	A	C	2		
			3	1	B	D	3		
6	ACDCDBDBDB		0	1	D	B	**2**		0
		1	1	0			**0**		
	（ルール 1）		2	1	A	C	2	2	
			3	1	B	D	3		

4.2.4 伸張動作

圧縮器ではシンボル・ルックアップ・テーブルに対し、表 4.3 のルールを割り当てるものとする。「タスク 1」は表 3.3 とほぼ同等だが、ストールを回避するため、ルール 3 が異なる。また、「タスク 1」と「タスク 2」は同時に実行される。ルールを元に伸張動作させた例を表 4.4 に示す。以下の動作となる。

1. 初期状態。テーブルにデータが登録され、空きが無い状態。登録インデックスは不定。

2. 入力データ「2」を評価。圧縮シンボル。ルール 1 を適応。count を+1。登録されているデータ・ペア「DC」を出力。削除インデックスが指す count を-1。count が 0 なのでエントリを削除。登録インデックスを空きエントリに移動。削除インデックスを次のエントリに移動。

3. 入力データ「DC」を評価。非圧縮シンボル。ルール 2 を適応。登録インデックスの位置に「DC」を登録。count 値を初期化。「DC」を出力。削除インデックスが指す count を-1。削除インデックスを次のエントリに移動。空きエントリが無いため、登録インデックスは不定。

4. 入力データ「DB」を評価。非圧縮シンボル。テーブルに空きが無い。ルール 3 を適応（Lazy）。「DB」を出力。削除インデックスが指す count を-1。削除インデックスを次のエントリに移動。

5. 入力データ「DB」を評価。非圧縮シンボル。削除インデックスが指す count を-1。count が 0 なのでエントリを削除。ルール 2 を適応。空いたエントリに新たに「DB」を登録。「DB」を出力。削除インデックスを次のエントリに移動。空きエントリが無いため、登録インデックスは不定。

6. 入力データ「0」を評価。圧縮シンボル。ルール 1 を適応。count を+1。登録されているデータ・ペア「DB」を出力削除インデックスが指す count を-1。count が 0 なのでエントリを削除。登録インデックスを空きエントリに移動。削除インデックスを次のエントリに移動。

4. と 5. で同じ「DB」が入力されているが、4. ではテーブルに空きが無く登録できないため、圧縮を諦めたデータであることが判明する。5. ではテーブルに登録され、これ以降は圧縮される。この動作が Lazy 方式の特徴となる。

表 4.3: 伸張ルール (lazy)

タスク	条件	動作	ルール
タスク 1	入力シンボルが圧縮シンボル	頻度カウンタを+1。シンボルが指す index のデータ・ペアを出力。	ルール 1
	入力シンボルが非圧縮シンボル＆テーブルに空きがある	頻度カウンタを初期化。データ・ペアを登録インデックスの指す位置に登録。登録インデックスを移動。データ・ペアをそのまま出力。	ルール 2
	入力シンボルが非圧縮シンボル＆テーブルに空きが無い	なにもしない。データ・ペアをそのまま出力。	ルール 3
タスク 2	無条件	削除 index が指す頻度カウンタを-1。頻度カウンタが 0 ならエントリを削除、テーブルに空きを作る。削除 index を移動。	

表 4.4: 伸張動作 (lazy)

時間	入力シンボル	登録 index	テーブル					削除 index	出力シンボル
			index	valid	s0	s1	count		
1	初期状態	n/a	0	1	A	A	1		
			1	1	B	B	1	1	
			2	1	A	C	3		
			3	1	B	D	4		
2	2DCDBDB0	1	0	1	A	A	1		AC
			1	0			**0**		
	（ルール 1）		2	1	A	C	**4**	2	
			3	1	B	D	4		
3	2DCDBDB0	n/a	0	1	A	A	1		DC
			1	1	D	C	**1**		
	（ルール 2）		2	1	A	C	**3**		
			3	1	B	D	4	3	
4	2DCDBDB0	n/a	0	1	A	A	1	0	DB
			1	1	D	C	1		
	（ルール 3,lazy）		2	1	A	C	2		(lazy)
			3	1	B	D	**3**		
5	2DCDBDB0	n/a	0	1	D	B	**1**		DB
			1	1	D	C	1	1	
	（ルール 2）		2	1	A	C	2		
			3	1	B	D	3		
6	2DCDBDB0	1	0	1	D	B	**2**		DB
			1	0			**0**		
	（ルール 1）		2	1	A	C	2	2	
			3	1	B	D	3		

4.3 実装

Lazy 方式の実装は、削除インデックスが増え、シンボル・ルックアップ・テーブルの制御方法が若干変わるが、3.3.3 節から大きく変化は無い。しかし、頻度カウンタへのアクセスが、タスク 1 と 2 から同時に行われるため、シリアル実装（図 3.4）が FPGA では困難となる。

頻度カウンタのアクセスは 2 ポートリード、2 ポートライトの 4 ポートアクセスが発生する。FPGA の RAM は、2 リード *or* 2 ライト *or* 1 リード 1 ライト、いずれかのアクセスを想定した構成となっており、4 ポートアクセスメモリの RAM への実装は出来ない。このため、頻度カウンタはパラレル実装（図 3.5）となり、ハードウェア・リソース量や周波数は図 3.13 のパラレル実装の結果に準ずる。

Lazy 方式の LCA-DLT ハードウェアの構成は、圧縮器が図 4.2、伸張器が図 4.3 となる。図中、削除インデックスが追加されている。

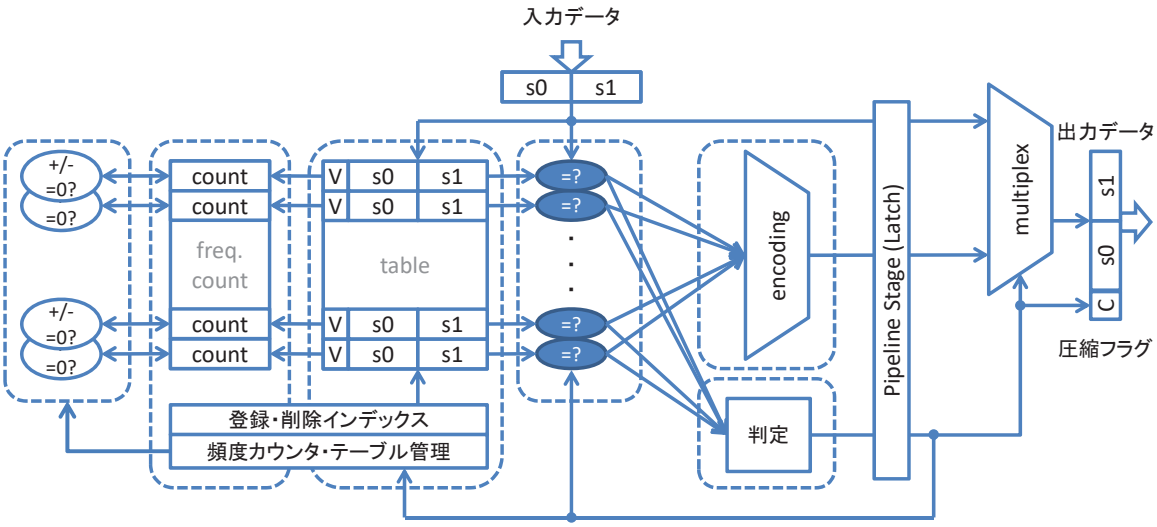


図 4.2: Lazy 方式の圧縮ハードウェア構成

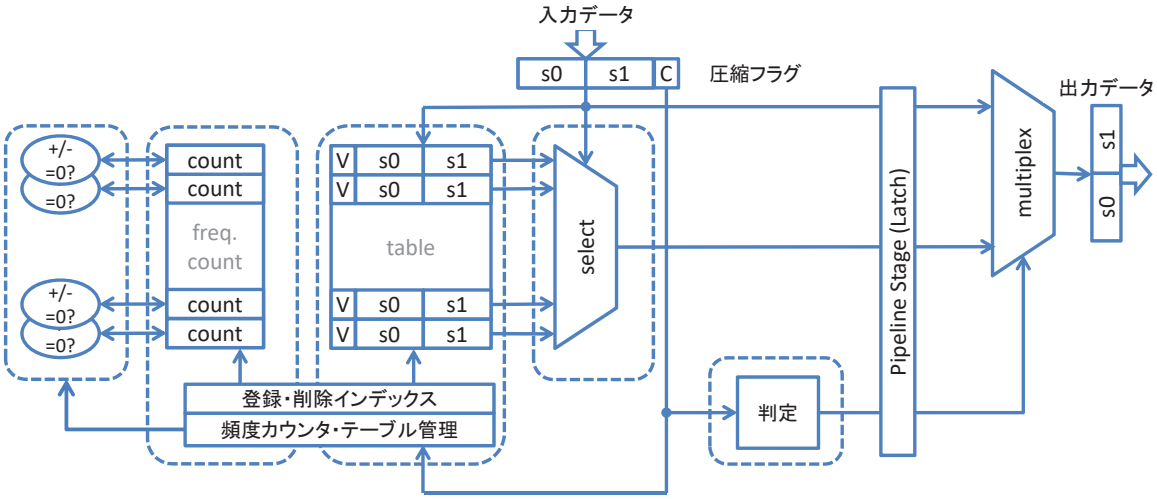


図 4.3: Lazy 方式の伸張ハードウェア構成

4.4 評価

評価項目として、以下の項目を評価する。
なお、パラメータは $dw = 16[bit], sw = 4, 5, 6, 7, 8[bit], cw = 8[bit]$ とする。

- 圧縮率
  得られた圧縮率を、前章の Full 方式の圧縮率の結果と比較する。Lazy 方式では本来圧縮出来たはずのシンボルを、圧縮せずに処理するため、圧縮率は悪化すると予測される。
  この評価のため、

$$\text{圧縮率変化} = \frac{Lazy\ \text{方式での圧縮率}}{Full\ \text{方式での圧縮率}} \tag{4.2}$$

  で比較を行う。1.0 よりも大きくなれば Lazy 方式では圧縮率が悪化したと判断される。

以下は評価対象外とする。

- ストール率
  Lazy 方式の圧縮ではストールは発生しない。ストール率は常に 0 %となる。
- ハードウェアの規模と速度
  パラレル実装（3.4.3 節）の結果に準ずる。動作周波数も 100MHz 付近で変化は無い。

ベンチマークに使用したデータは、前章と同様、[28] から 10MByte のファイル（XML, English Test, MIDI pitch, Protein, DNA, Linux source）データを使用する。

4.4.1 圧縮率の評価

圧縮率を比較した結果を図 4.4 に示す。

Lazy 方式では圧縮率が悪化すると予測されたが、実際には DNA 及び English Text 以外で Lazy 方式の方が高い圧縮率が得られるという結果が得られた。また、Full 方式では 2 段目以降圧縮が効かないという結果が出たが、Lazy 方式では圧縮されるという傾向がいくつか見られる。

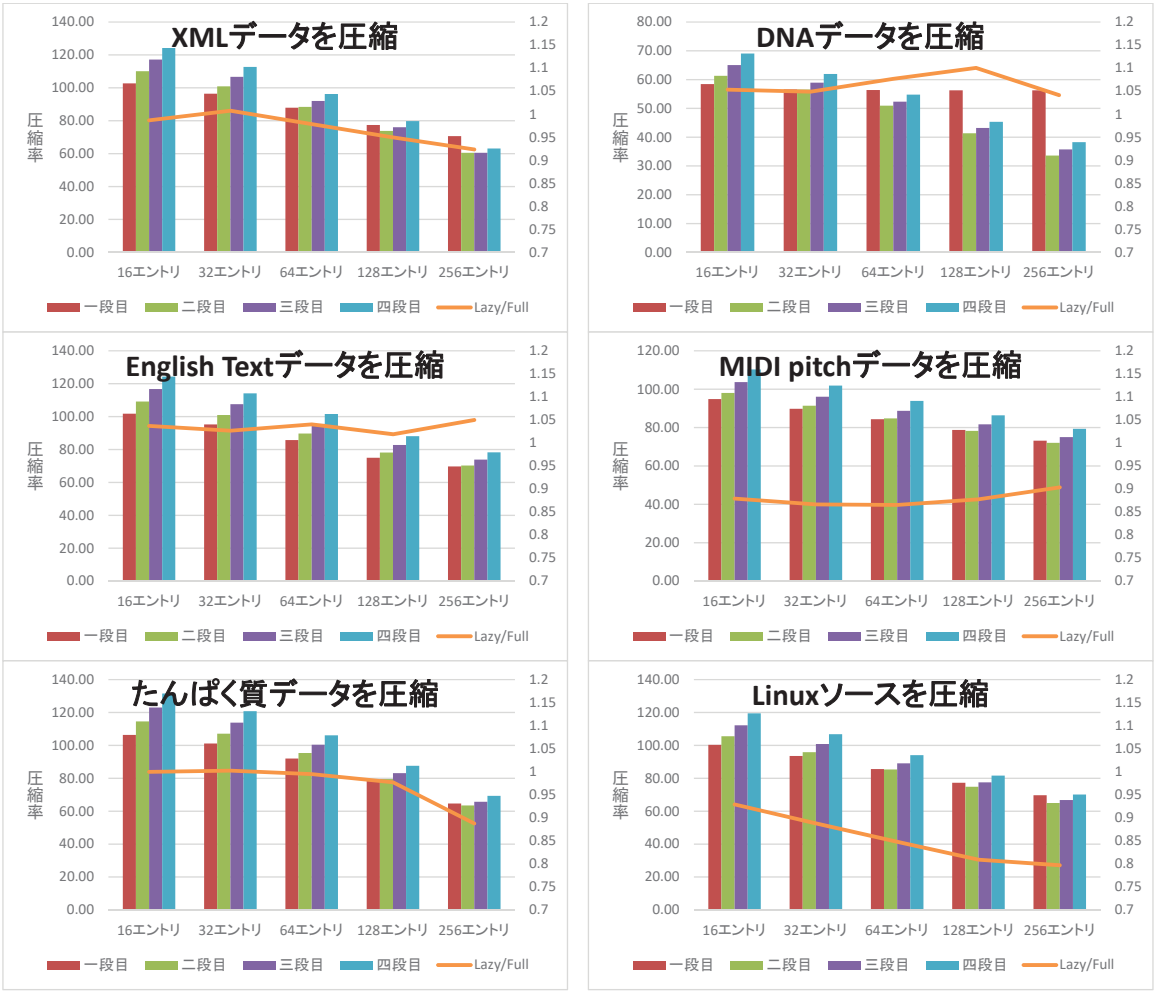


図 4.4: 圧縮率（Lazy 方式）
（横軸：テーブルのエントリー数、
棒グラフ：縦軸左：圧縮率、
折れ線グラフ：縦軸右：圧縮率の比較）

4.4.2 議論

Lazy 方式の LCA-DLT アルゴリズムにより、完全なストリームデータ圧縮・伸張ハードウェアを実現することが出来た。このハードウェアでは、100MHz でスループット 100M シンボル/s、16bit データで 1.6Gbps を常に維持することが可能となる。また、レイテンシは低く押さえられ（Full 方式と同じ 2 クロック）、伝送系に挿入した際の影響を最低限に抑えることができる。

圧縮率は Full 方式に比べ改善された例もあった。常に Full 方式より良いとは言えないが、同等の圧縮率を持つことが判る。Full 方式と同様、テーブルのエントリー数とカスケード段数により、圧縮率とハードウェア量のトレードオフを行うことが出来る。

本章ではストールの排除を目的としたため、動作周波数の問題は依然残る。次章ではこの問題について言及する。

Lazy 方式により、実用レベルのストリームデータ圧縮・伸張ハードウェアが可能となったため、本章のハードウェアを Altera Stratix V A5 FPGA 評価ボードへ実装する実証実験を行った。その結果と、他の圧縮ハードウェアの実装との比較例を表 4.5 に挙げる。比較対象として、IBM の実装例 [23] と Altera の実装例 [24] を用いる。[24] では Altera Stratix V A7 デバイスが使用されている。

比較は圧縮器を用いて行った。この実験の時点で 5 章の高速化の一部（5.2.3 節のフォワーディングパス）を実装しており、動作周波数が向上している。圧縮器単体の実装例と、圧縮器を 8 並列実装した実装例を列挙する。8 並列実装回路には、出力されたデータ列のビット結合ロジック（アライメント処理）が含まれる。図 4.5 は本方式 8 並列のフロアプラン結果となる。

IBM と Altera の実装では DEFLATE アルゴリズムを採用しており、50 %程度の圧縮率が得られている。本方式は 60 %〜80 %程度であり、圧縮率では及ばない。しかし、回路が単純なためリソースの使用量が少なく、高いスループット性能を実現したハードウェア実装となる。Lazy 方式により、表 4.5 の低レイテンシ・高スループットを常に維持することが可能となった。

表 4.5: 圧縮ハードウェアの比較

実装例	Logic	RAM	動作周波数 [MHz]	レイテンシ [クロック]	スループット [Gbit/s]
IBM の実装	279,000†	17.5Mbit†	200	17	25.6
Altera の実装	291,000†	35Mbit†	193	87	24.7
本方式（単体）	9,306	0	284	6	4.5
本方式（8 並列）	76,605‡	0	221	13‡	28.3

†:[24] の数値から換算。
‡:ビット結合ロジック含む。

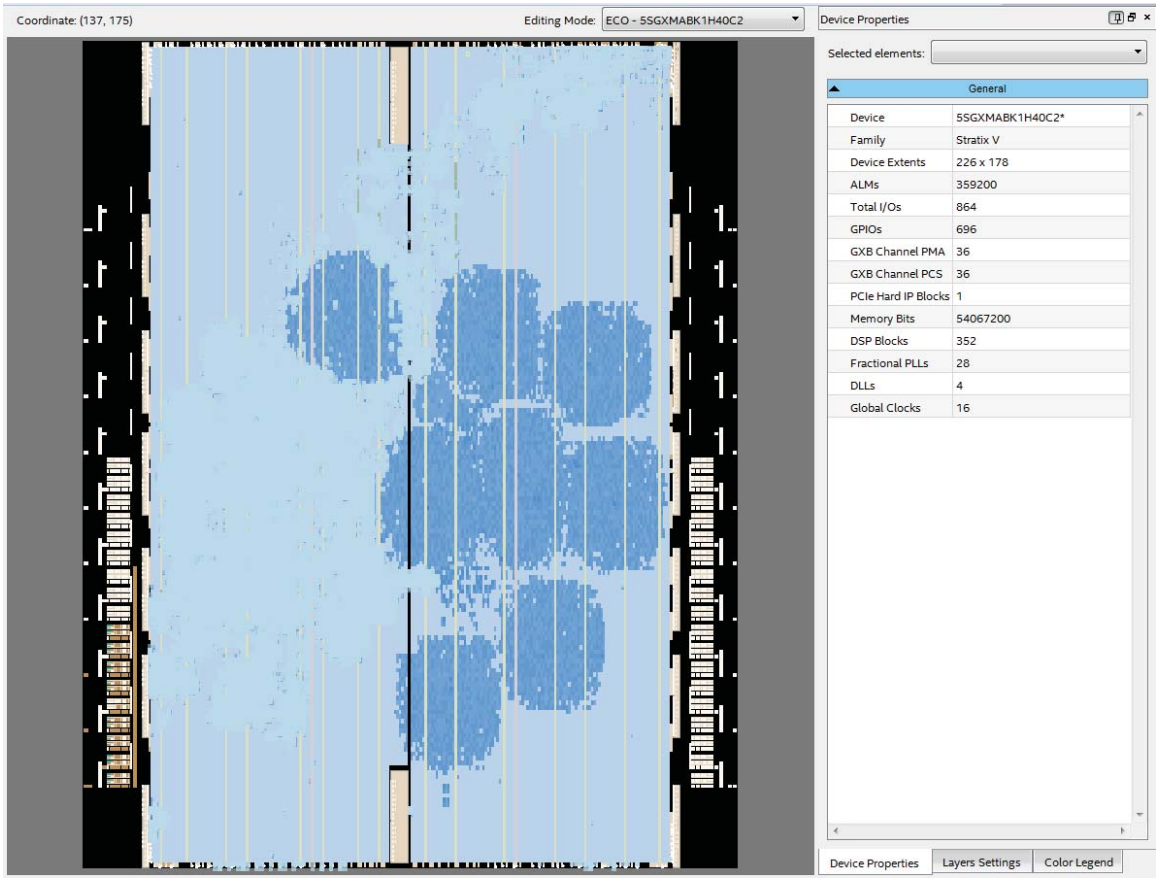


図 4.5: フロアプラン結果（8 並列）
色の濃い部分が圧縮ロジック及びビット結合ロジック

4.5 まとめ

本章では 3 章で問題となっていた、処理のストールを回避するための解決策を提案した。Lazy なテーブル管理手法、および削除インデックスの導入による動的なテーブル削除手法を提案・開発し、LCA-DLT アルゴリズムの拡張を行った。この拡張により、処理のストールが全く発生しない、真の意味でのストリームデータ圧縮・伸張ハードウェアが実現可能となった。

しかし、本章の構成では動作周波数の改善は考慮されておらず、次章の課題となる。

第5章　時分割マルチスレッド技術を適応させたストリームデータ圧縮の高速化

5.1　はじめに

前章でストリームデータを一切止めること無く、連続的に圧縮・伸張可能な、かつレイテンシが非常に短いハードウェアを実現した。しかし、動作周波数が 100MHz 程度と、現在の FPGA 実装としては低いと言わざるを得ない。

本章では、まず動作周波数を制限しているボトルネック（クリティカルパス）を探し、その解決策を考える。クリティカルパスを解決する段階で、パイプライン化が行われるが、データハザードが発生し、スループットを維持することが難しいことが判明する。発生したデータハザードを回避するため、時分割マルチスレッド技術を導入し、性能を上げる手法を提案する。以下、時分割マルチスレッド技術を TSM（Time-Sharing Multithreading）とも記述する。

これらにより、より実用的な圧縮・伸張ハードウェアを設計・実装する。

5.2　設計

5.2.1　クリティカルパスの検討

図 4.2 を図 5.1 に再掲する。

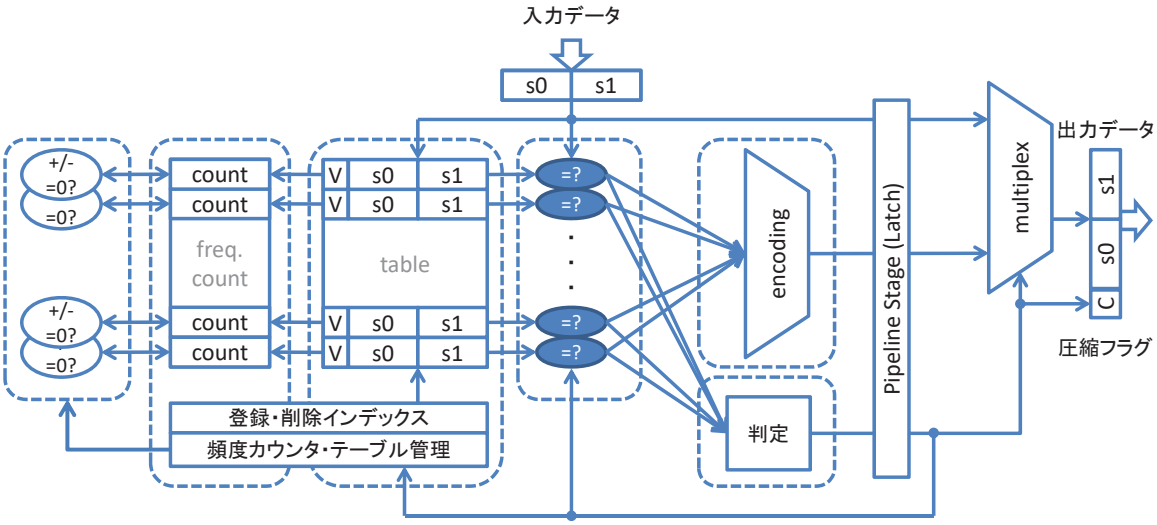


図 5.1: Lazy 方式の圧縮ハードウェア構成（再掲）

この処理の中でクリティカル・パスとなっているのは、全検索～登録判定までの組み合わせ回路となっている。なお、このクリティカル・パスは圧縮ハードウェアにのみ存在し、伸張ハードウェアには存在しない。

図 5.2 にクリティカル・パスの部分を抜粋する。

- テーブルのデータ・ペアと入力のデータ・ペアの比較、及び有効フラグの評価。
- 全エントリを評価し、いずれかのエントリにヒットしたかを示す、hit フラグの生成。

という 2 つの処理から成る。

シンボル化を行う「encoding」のロジックも複雑であるが、フィードバックが無いため、任意の段数でパイプライン化が可能である。このため、フィードバックを持つ登録判定ロジックがクリティカル・パスとなる。

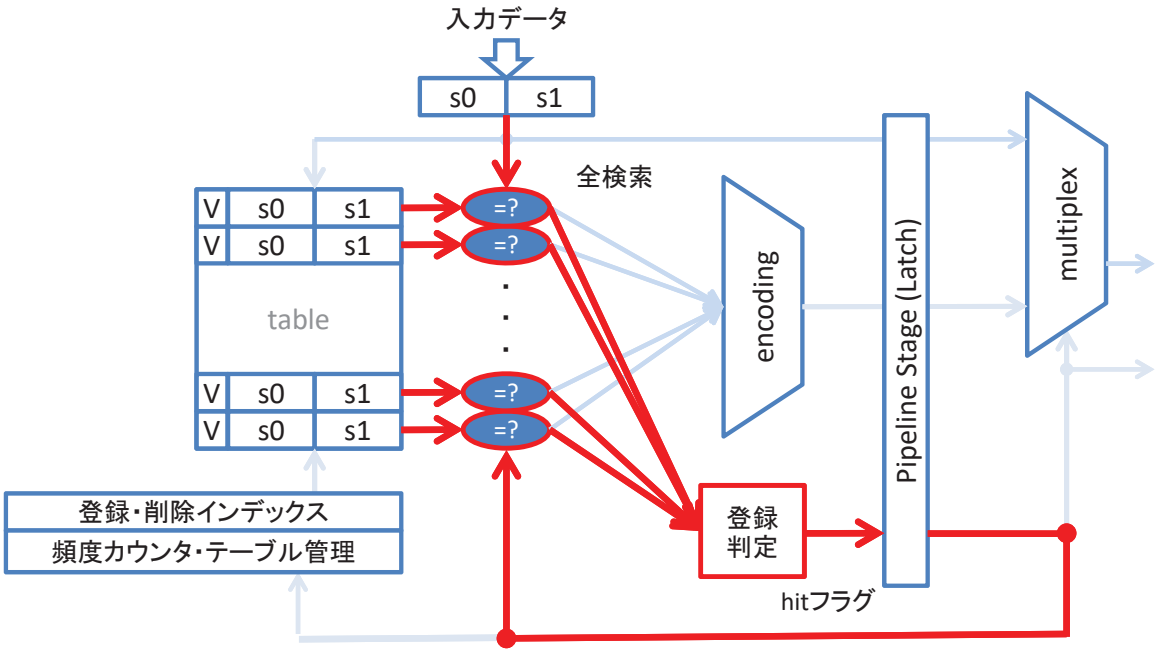


図 5.2: クリティカル・パスを抜粋

この組み合わせ回路に関連する信号の本数（=複雑さ）は以下の式で求めることが出来る。

$$(1+1+dw+dw)\times 2^{sw}[\text{本}] \tag{5.1}$$

具体例として、$dw=16[bit], sw=8[bit]$ とした場合の信号の本数を求める。

$$(1+1+16+16)\times 2^{8}=8,704[\text{本}] \tag{5.2}$$

すなわち、8,704 本の入力から hit フラグ 1 本の結果を求める、巨大な組み合わせ回路となることが分かる。求めた hit フラグの結果が次のデータの判定に使用される、フィードバック・ロジックとなっている。

5.2.2 パイプライン化

このような複雑な組み合わせ回路の速度を上げる手法としては、パイプライン化が定石である。

図 5.2 の回路において、全検索結果（比較ロジック）と、登録判定（hit フラグ生成ロジック）の間に、パイプラインステージを挿入する。すると、前段が各 $(1+1+dw+dw)$ 本、後段が 2^{sw} 本 に分割され、組み合わせ回路が単純化される。図 5.3 に構成を示す。具体例では、前段が $(1+1+16+16)=34$ 本、後段が $2^{8}=256$ 本 と、パイプライン化する前に比べ少なくなる。

しかし、hit フラグはフィードバックされ、比較ロジックに入力されている。パイプライン化される前は、1 クロックで値を評価することが出来るが、パイプラインが挿入されたことで、評価まで 2 クロックかかる。このため、RAW（Read-After-Write）タイプのデータハザードが発生し、パイプラインにバブルが発生する。パイプラインにバブルが発生すると、連続してデータを処理することが出来なくなり、スループットは半分になる。

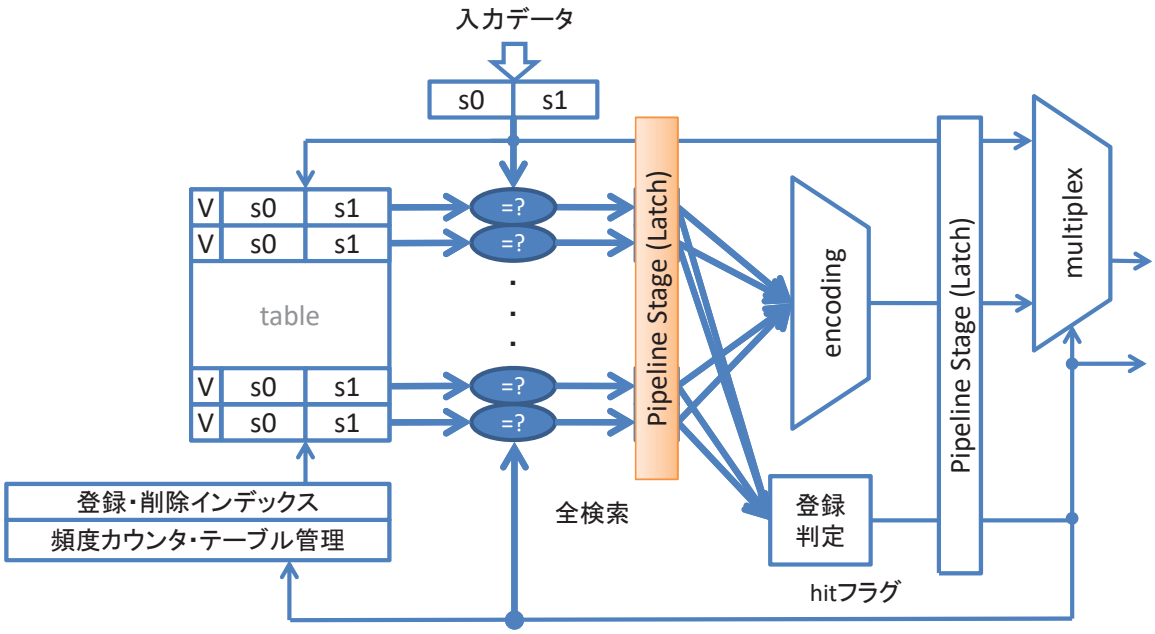


図 5.3: パイプラインステージを挿入

5.2.3 データハザードの回避

RAW タイプのデータハザードを回避するため、フォワーディングパスを設ける手法がある [29]。しかし、フォワーディングパスにすると、hit フラグが組み合わせ回路の出力となり、これが新たなクリティカル・パスとなる（図 5.4）。図の Pipeline Stage 出力により、hit フラグが生成されるが、これがレジスタ化されずに全検索回路に入力され、Pipeline Stage の入力となる。経路がすべて組み合わせ回路であり、クリティカル・パスとなる。

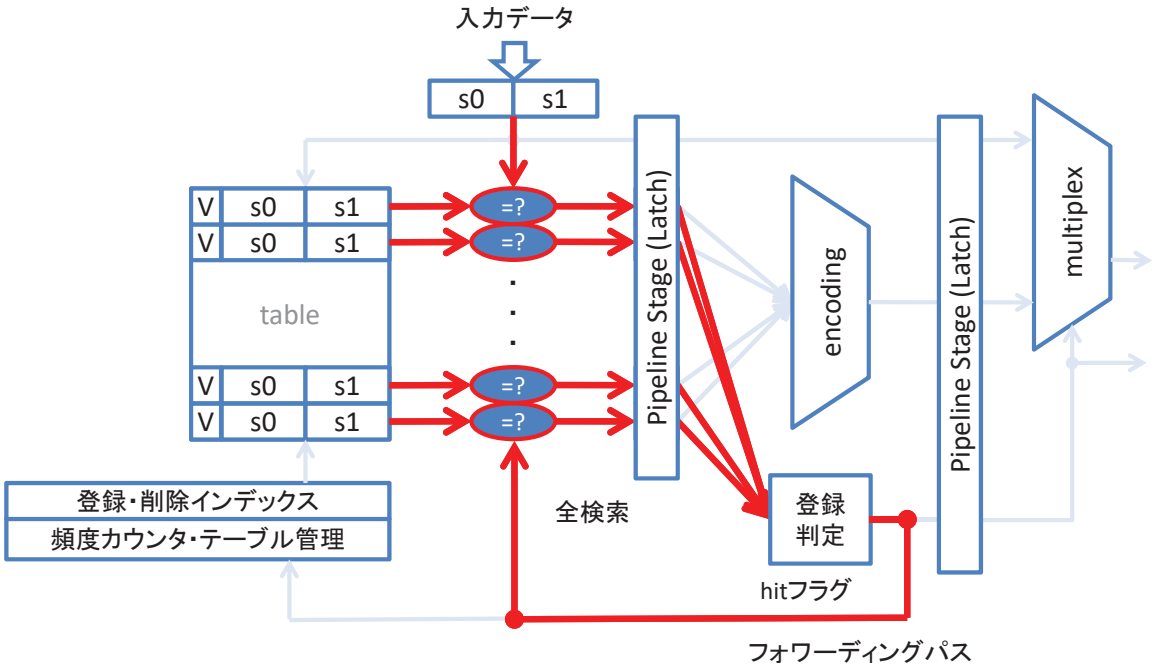


図 5.4: フォワーディングパスがクリティカル・パス

このクリティカル・パスを回避するためには、hit フラグをレジスタ出力（パイプライン化）にする必要がある。

5.2.4 時分割マルチスレッド技術の適応

5.2.2 節と 5.2.3 節は、それぞれ動作速度向上に貢献するが、相容れない条件となっている。この 2 節を両立させ、さらに動作速度を向上させるために、時分割マルチスレッド技術、TSM（Time-Sharing Multithreading）の適応を考える。

5.2.2 節によりパイプライン化を行い、hit フラグもレジスタ化されたものとする。hit フラグはフィードバックされているため、RAW ハザードが発生し、パイプラインにバブルが発生する。TSM ではこの発生したバブルに、依存関係の無い別のデータストリームを流し込むテクニックである。

図 5.5 は TSM を適応させた際のパイプラインの模式図である。hit フラグの依存関係があるため、各スレッドは連続処理は出来ない（RAW ハザード）。そのため、それぞれのスレッドは 2 ステージかけてデータを処理する。「判定」を行っている間、「検索」は空いているため、依存関係の無い別のスレッド 2 を差し込むことが可能となる。

相互のスレッド間には依存関係が無く、交互にデータを処理している状態となる。スレッド間には依存関係が無いため、hit フラグの参照までに 2 クロックの余裕が出来る。これにより hit フラグのレジスタ化が可能になり、速度向上が期待出来る。

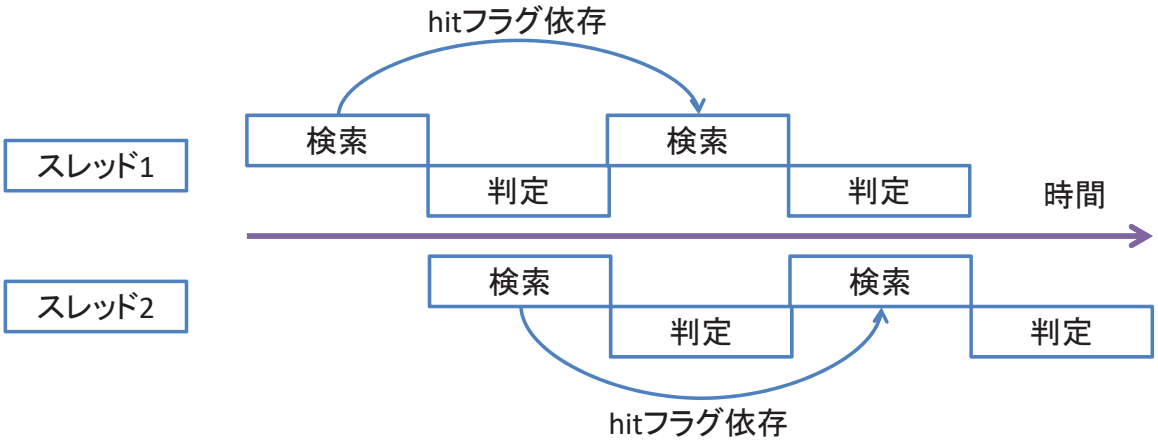


図 5.5: パイプラインステージ模式図（TSM 適応）

一方、それぞれのスレッドのスループットは $\frac{\text{動作周波数}}{2}$ となる。全体のスループットは動作周波数となるため、周波数が上がれば全体の性能は上げることが可能となる。

5.3 実装

TSM を適応させた LCA-DLT ハードウェアの構成を、図 5.6 図 5.7 に示す。伸張ハードウェアには問題となるクリティカル・パスは無いが、圧縮・伸張ハードウェアは対照である必要があるため、同時に設計する。

状態を記憶する素子、テーブルのメモリ、頻度カウンタ、登録インデックス、削除インデックス、などはスレッド毎に必要となり、倍のリソースを必要とする。しかし、処理を行う組み合わせ回路はスレッド間で共有することが可能となり、倍のリソースとはならない。判定処理の出力（hit フラグ）がレジスタ化され、クリティカル・パスが分割されていることが分かる。

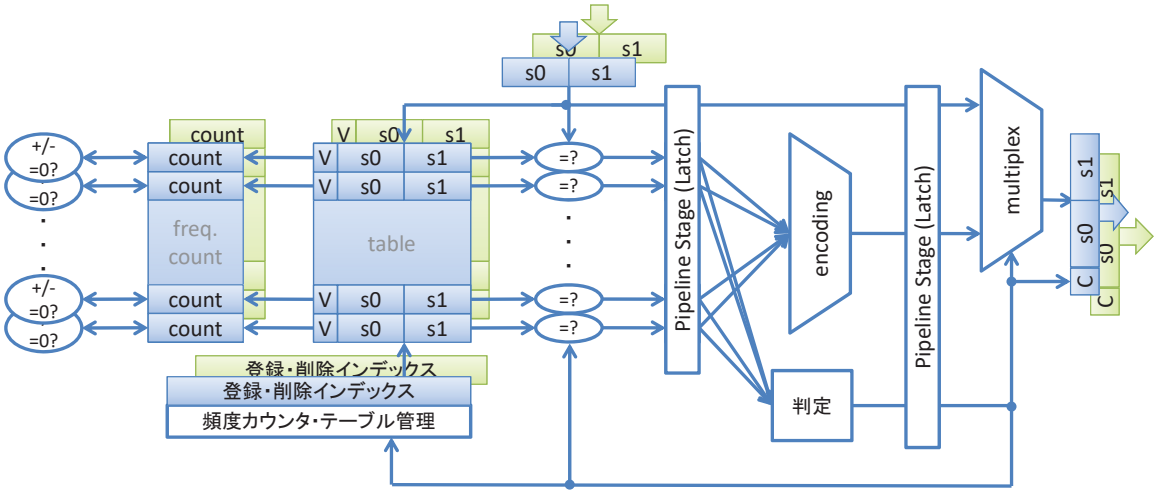


図 5.6: 時分割マルチスレッド対応、圧縮ハードウェア

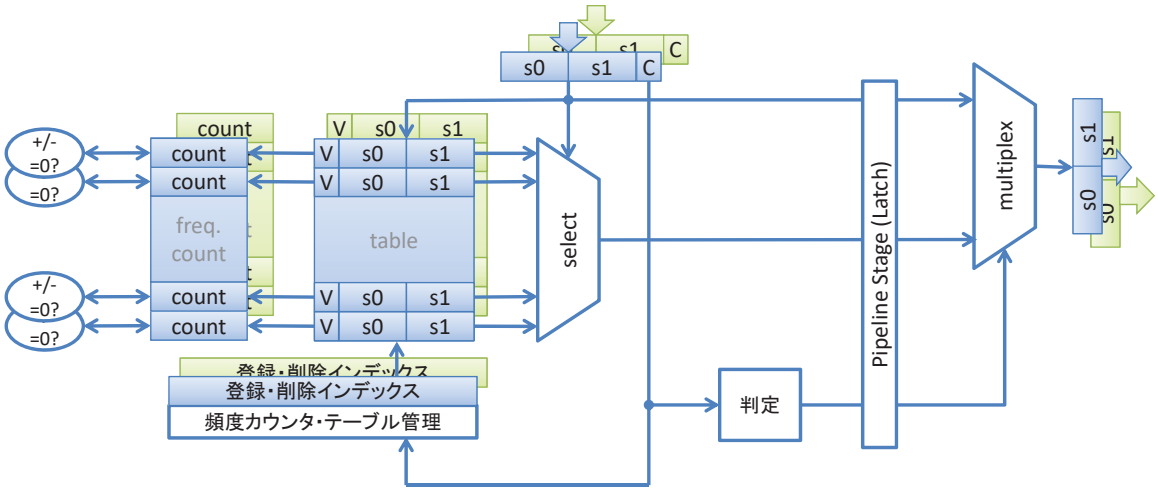


図 5.7: 時分割マルチスレッド対応、伸張ハードウェア

5.4 評価

評価項目として、以下の項目を評価する。
なお、パラメータは $dw = 16[bit], sw = 8[bit], cw = 8[bit]$ とする。

- ハードウェアの規模と速度
  TSM を有効/無効した場合の、ハードウェアの使用リソース変化と、動作周波数の変化を評価する。
  実装環境は以下になる。前章とは異なるが、TSM の有効無効の比較を目的とする。

- ターゲットデバイス：Xilinx Kintex UltraScale XCKU025-FFVA1156-1-C
- 開発ツール　　：Vivado2017.2

以下は評価対象外とする。

- 圧縮率
  Lazy 方式と同じアルゴリズムであるため、圧縮率は 4.4.1 節と全く同じとなる。
- ストール率
  Lazy 方式の圧縮なのでストールは発生しない。

5.4.1　ハードウェアの評価

圧縮器の FPGA への実装結果は、図 5.8 となった。TSM により、周波数は 277MHz → 342MHz へと約 23 %向上し、組み合わせ回路は約 22 %減少した。レジスタは約 32 %増加し、RAM も使用されるようになった。これは TSM により 2 スレッド分のメモリが必要となり、記憶素子が倍となったためである。

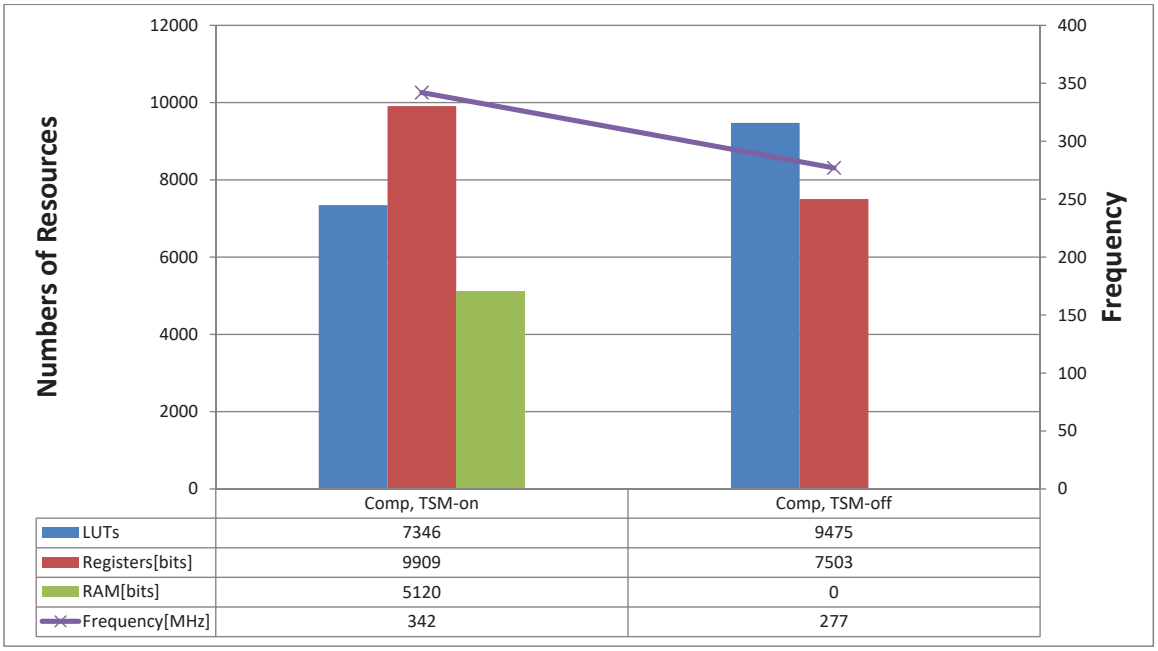


図 5.8: 圧縮、TSM の効果
（棒グラフ：縦軸左：ハードウェアリソース量、
折れ線グラフ：縦軸右：周波数）

伸張器の FPGA への実装結果は、図 5.9 となった。TSM により、周波数は 328MHz → 353MHz へと約 7.6 %向上し、組み合わせ回路は約 65 %減少した。レジスタは約 67 %減少したが、RAM

の使用量は 4.3 倍となった。レジスタを利用していた記憶素子の大半が RAM に移行した事になる。

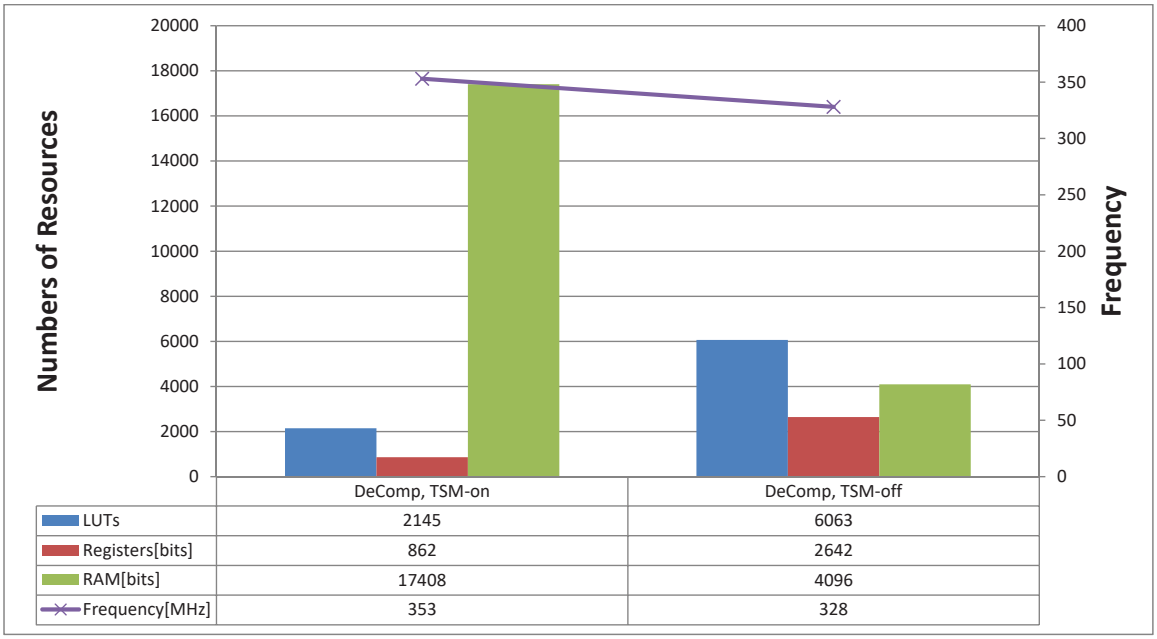


図 5.9: 伸張、TSM の効果
（棒グラフ：縦軸左：ハードウェアリソース量、
折れ線グラフ：縦軸右：周波数）

5.4.2 議論

周波数は特に圧縮ハードウェアで大きな向上が見られた。ストリームデータ圧縮・伸張では同等のスループットを維持する必要があるが、TSM が無効の状態では、圧縮ハードウェアの速度が伸張ハードウェアの速度に比べ、20 %ほど遅く、アンバランスとなっている。TSM を有効にすることにより、圧縮・伸張ハードウェア共に 340MHz 以上の速度が確保できた。これにより、データの幅が 16bit の場合、5.4Gbps のスループットを維持することが可能となり、大幅な高速化が実現している。

一方、レイテンシは 6 クロックとなっている（パイプライン化により 2 → 3 クロック、TSM により ×2）。前章のハードウェアからは増加しているが、低いレイテンシを維持している。レイテンシを重視する場合は、周波数を犠牲にして TSM を無効にするというオプションをユーザが選択することも可能となる。

TSM を有効にすることにより、2 スレッド分のリソースがとなっていることに注意が必要である。記憶素子の量が 2 スレッド分となり、倍の容量を必要とするはずである。

圧縮器では、表 5.1 となり、概ね倍となっていることが分かる。レジスタの一部が RAM に

移動したため、レジスタの使用量は倍とはならない。

表 5.1: 圧縮器の記憶素子の総数

TSM	レジスタの bit 数	RAM の bit 数	総 bit 数
無効	7,503	0	7,503
有効	9,909	5,120	15,029

伸張器では、表 5.2 となり、この関係は崩れ、倍以上の容量が利用されている。これは RAM の出力データを複数箇所で利用する、マルチ・リード・ポート構成となったため、RAM ブロックが複数個利用されたためである。ASIC などでポート数の多い RAM が利用できる場合はこの現象は起きないと考えられる。レジスタの多くが RAM に移動したため、レジスタの使用量は減少している。

表 5.2: 伸張器の記憶素子の総数

TSM	レジスタの bit 数	RAM の bit 数	総 bit 数
無効	2,642	4,096	6,738
有効	862	17,408	18,270

圧縮・伸張ハードウェア共に組み合わせ回路が大きく減少している。これは、TSM により頻度カウンタへのアクセスが分散したため、Lazy 方式のハードウェアにおいてパラレル実装（図 3.5）となっていた動的ヒストグラム生成ロジックを、シリアル実装（図 3.4）に置き換える事が可能となったためである。

現在の FPGA アーキテクチャでは、レジスタ（フリップフロップ）が組み合わせ回路より豊富にある構成となっており、RAM も大容量となっている。例えば本章で使用した Xilinx の FPGA では、組み合わせ回路：フリップフロップ=1:2 の比率となっている [30]。RAM は 12.7Mbit 搭載されており、伸張ハードウェアが使用した 17Kbit は 0.2 %未満の使用率となる。

このように現在の FPGA アーキテクチャはレジスタリッチな構成であり、TSM 化は今日の FPGA に適した構成と言える。

5.5 まとめ

本章では、3 章、4 章で問題となっていた、動作周波数の改善を行った。パイプライン化、および時分割マルチスレッド技術（TSM）の導入により、動作周波数の改善が見られた。また、TSM の採用により、ハードウェアリソースのバランスが改善され、特に FPGA に向いた構成が可能となった。

第6章　結論

本研究は、データストリームを一切止めること無く、圧縮・伸張処理が可能なハードウェアの実現を目標に開始された、データストリームにインラインで組み込めるハードウェアを理想とし、小さく、処理時間（レイテンシ）の短いハードウェアを念頭に置き、開発が行われた。

これを実現するため、新しい圧縮アルゴリズムである、LCA-DLT アルゴリズムを提案し、設計・実装を行った。段階的に問題点を解決し、実用的なハードウェアとして一つの完成形を得ている。

LCA-DLT 圧縮・伸張技術は以下の 4 つの要素技術から構成される。

1. LCA アルゴリズム
2. 動的ヒストグラム生成手法
3. Lazy なテーブル管理手法
4. 動的なテーブル管理

これら 4 つの技術が組み合わさり、完全なストリームデータ圧縮・伸張を、一定量のハードウェアで実装することが可能となった。

既存の圧縮技術である LZ 系と比較しても、省リソースで高いスループットを維持できる方式となった。ハードウェアを複数並べ、並列に圧縮処理を行うことも可能であり、100Gbps 以上の伝送系への応用も可能となる。また、レイテンシの短さを生かし、CPU〜メモリ間など、レイテンシに敏感なアプリケーションへの応用も期待出来る。ハードウェアのリソース量も、1 コアあたり 100 万トランジスタ程度であり、現在のシリコン集積度（数億〜数十億トランジスタ）からは小さいと言える。あらゆる伝送系の送受信ロジックにインラインで埋め込み、データ転送を圧縮し、転送量を減らす事が可能となる。

現在、可逆圧縮技術の伝送系への応用は限定的である。アプリケーションで工夫することが多く（例：ZIP で圧縮してメールに添付する）、基本的にオフラインで圧縮されたデータをメモリやファイルに貯め、送受信を行うという形式を取る。インターネットでは HTTP が標準で圧縮転送を持つが、これもサーバー側で事前に圧縮したデータを用意しておく、オフライン形式である。

データを貯めない、オンライン形式でデータを圧縮し、転送するアプリケーションの例として、バックアップテープが挙げられる。LTO（Linear Tape-Open）[31] では標準で LTO-DC

と呼ばれる圧縮をサポートしており、テープのバックアップ容量及び速度を約 2 倍としている。2018 年現在、LTO は 8 世代目となっており、圧縮時の転送速度が 750MB/s まで向上している。バックアップテープのデータ流れが一方方向で、スループットさえ確保できればレイテンシの増加は問題とならないため、圧縮技術が利用可能となっている。また、バックアップテープは業務での利用が主であり、本体価格よりランニングコストが優先されるため、コストの高いハードウェアを投入可能である、という背景もある。

既存の圧縮技術では、圧縮にかかる「コスト」（CPU パワー、価格、ハードウェアリソース量、時間など、様々のものを含む）が高いため、応用が限定的となってしまっている。

一方、今後データの転送要求量は増え続け、[32] の予測では 2020 年に 194EB（エクサバイト）/月に達するとされている。特にクラウド・サービスの登場、IoT 化がデータ量爆発に拍車をかけている状況となっている。エッジ・コンピューティングなどによる再分散化によりデータ伝送量の増加率を下げようという試みはあるが、基本、伝送系の大容量化という対策になり、破綻が懸念されている。

こういった状況の中、圧縮の「コスト」を軽減した本方式をあらゆる伝送系に適応させ、転送データ量そのものを減らす対策を提案する。IoT 機器が生成するデータ、クラウドに転送するデータ、サーバ内で保存されるデータ、これらすべてを圧縮することにより、データ量爆発に対抗する。

また、データ転送にかかるエネルギ削減にもつながる。転送方式によって異なるが、データ転送には 数 $[nJ/bit]$〜数 $[pJ/bit]$ オーダーのエネルギが必要であり、データ量が増えれば莫大なエネルギが必要となる。圧縮により転送データ量が削減できれば、その分必要なエネルギも削減可能となる。

これらの実現のためには、標準化・ASIC/SoC 化が不可欠となる。標準化によりメーカの異なる機器間の接続が保証され、様々な機器が自由につながる状況とならなければ普及は難しい。また、本研究での FPGA 実装では消費電力面で不利であり、圧縮でデータ転送に必要なエネルギが減っても、圧縮処理の消費電力がそれを上回ってしまっては本末転倒である。価格面でも ASIC/SoC 化されなければやはり普及は困難である。今後このような普及のための努力が必要となると考えられる。

研究・技術面ではいくつかの方向性が考えられる。本方式の圧縮率は既存の LZ 系などには及ばず、改善が求められる。圧縮率の改善には、符号化の最適化や前処理フィルタの導入、データに適応させた学習方法の開発などが考えられる。また、圧縮処理の計算量（≒ハードウェアリソース量）が他方式に比べ少ないとは言え、まだ削減する余地がある。計算量が削減できればハードウェアリソース量の削減、消費電力の削減につながり、IoT 機器への実装が容易となる。通信系への応用を考慮した場合、現在の本方式ではエラー対策の欠落が問題となる。エラー検知・訂正機構の組み込みなども今後の課題となる。

これらの研究・改良を行い、標準化を働きかけ、圧縮技術がすべてのデータ通信で汎用的に利用されるような将来を考えている。

謝辞

本研究の機会を与えていただき、ご指導いただいた筑波大学システム情報工学研究科　准教授　山際伸一先生に深く感謝いたします。

研究の貴重な助言をいただき、ご指導いただいた、筑波大学 システム情報工学研究科教授 和田耕一先生、同教授 安永守利先生、同准教授 山口佳樹先生、九州工業大学 大学院情報工学研究院教授 坂本比呂志先生に深く感謝いたします。

ツールやデバイスの使い方など、支援いただいたメーカーの皆様に感謝の意を表したいと思います。

共に研究を進めて頂いた筑波大学 システム情報工学研究科修士 森田隆太氏に感謝いたします。

最後に、短い期間でしたが、同じ研究室の方々に心より感謝いたします。

参考文献

[1] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, Vol. 40, No. 9, pp. 1098–1101, September 1952.

[2] Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes. *J. ACM*, Vol. 34, No. 4, pp. 825–845, October 1987.

[3] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337–343, May 1977.

[4] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. 24, No. 5, pp. 530–536, September 1978.

[5] Shinichi Yamagiwa, Keiichi Aoki, and Koichi Wada. Performance enhancement of inter-cluster communication with software-based data compression in link layer. In *Proceedings of IASTED PDCS 2005*, pp. 325–332, 2005.

[6] Intel©QuickAssist テクノロジー. https://ark.intel.com/ja/products/79483/Intel-QuickAssist-Adapter-8950.

[7] PCI SIG. https://pcisig.com/.

[8] ethernet alliance. https://ethernetalliance.org/.

[9] Universal Serial Bus. http://www.usb.org/home.

[10] HDMI. https://www.hdmi.org/.

[11] JEDEC. https://www.jedec.org/.

[12] An Introduction to the Intel©QuickPath Interconnect. https://www.intel.co.jp/content/www/jp/ja/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[13] HyperTransport Consortium. https://www.hypertransport.org/.

[14] Inside Pascal: NVIDIA’s Newest Computing Platform. https://devblogs.nvidia.com/inside-pascal/.

[15] Jing Zhang. Real-time lossless compression of soc trace data. Master's thesis, Department of Electrical and Information Technology, 2015.

[16] Shirou Maruyama, Hiroshi Sakamoto, and Masayuki Takeda. An online algorithm for lightweight grammar-based compression. *Algorithms*, Vol. 5, No. 2, pp. 214–235, 2012.

[17] 山際伸一. 物理的な伝送限界を越える超高速ストリームデータ圧縮技術の開発. http://www.science-academy.jp/showcase/13/pdf/P-074_showcase2014.pdf.

[18] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, Vol. 312, No. 1, pp. 3 – 15, 2004. Automata, Languages and Programming.

[19] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, Vol. 28, No. 1, pp. 51–55, March 2003.

[20] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pp. 346–357. VLDB Endowment, August 2002.

[21] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.*, Vol. 31, No. 3, pp. 1095–1133, September 2006.

[22] High Performance DEFLATE on Intel Architecture Processors. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-deflate-compression-paper.pdf.

[23] S. Rigler, W. Bishop, and A. Kennings. Fpga-based lossless data compression using huffman and lz77 algorithms. In *2007 Canadian Conference on Electrical and Computer Engineering*, pp. 1235–1238, April 2007.

[24] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pp. 4:1–4:9, New York, NY, USA, May 2014. ACM.

[25] AHA3642. http://www.aha.com/DrawProducts.aspx?Action=GetProductDetails&ProductID=38.

[26] Tony Summers. Comtech AHA, Lossless Data Compression in Storage Networks, October, 2006. http://aha.com/show_pub.php?id=238.

[27] CAST GZIP IP. http://www.cast-inc.com/ip-cores/data/zipaccel-c/index.html.

[28] Compressed Indexes and their Testbeds. http://pizzachili.dcc.uchile.cl.

[29] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[30] 7 Series FPGAs Configurable Logic Block. https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[31] LTO:Linear Tape-Open. https://www.lto.org/.

[32] 総務省　情報通信統計データベース. http://www.soumu.go.jp/johotsusintokei/whitepaper/h29.html.

論文リスト

[1] S. Yamagiwa, K. Marumo, and H. Sakamoto, ”Stream-based Lossless Data Compression Hardware using Adaptive Frequency Table Management”, In Proceedings of the VERY LARGE DATA BASES / BPOE 2015, Lecture Note in Computer Science 9495, pp.133-146 . Springer, 2015.

[2] K.Marumo, S.Yamagiwa, R.Morita and H.Sakamoto, ”Lazy Management for Frequency Table on Hardware-Based Stream Lossless Data Compression”, MDPI : Information,7 (4), 63 (2016-10)

[3] K.Marumo and S.Yamagiwa, ”Time-sharing Multithreading on Stream-based Lossless Data Compression”, In Proceeding of CANDAR’17 / CSA’17, pp.305-310, Japan, November 19-22, 2017.