修 士 論 文

# 消費電力削減に適したキャッシュブロック圧縮アルゴリズムに関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

川原 貴裕

2007年3月

修 士 論 文

# 消費電力削減に適したキャッシュブロック圧縮アルゴリズムに関する研究


指導教官　田中清史 助教授

審査委員主査　田中清史 助教授
審査委員　日比野靖 教授
審査委員　井口寧 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

510026川原 貴裕



提出年月: 2007 年 2 月

## 概 要

近年，プロセッサの消費電力が増大し，それに付随した放熱の問題や動作周波数の頭打ちが起こるなど，消費電力の削減は重要な課題となっている．そのため，プロセッサの消費電力削減を目指し，Gated-Vdd[2] や Cache Decay[3] など，数々の研究が行われている．

プロセッサの消費電力削減法のひとつであるデータ圧縮を用いた Gated-Vdd による電力削減法 [1] では，まず L2 キャッシュに格納されるブロックに対してデータ圧縮を試みる．この圧縮により，ブロックサイズの 1/2 以下にデータを圧縮できたキャッシュブロックについては圧縮した状態でデータを格納し，空いたブロックサイズの 1/2 の領域を Gated-Vdd 制御により電力供給をオフにすることで電力削減を行っている．

本研究はこの電力削減法の圧縮アルゴリズムの点に着目する．1/2 以下に圧縮できるブロック数を増やすことが出来れば，L2 キャッシュにおける消費電力をさらに削減することが可能となる．そのため，本論文ではデータ圧縮を用いた Gated-Vdd による電力削減法で用いられた圧縮アルゴリズムである Frequent Pattern Compression[4] 以外にも，Frequent Value Compression[5]，X-Match アルゴリズム [6]，X-RL アルゴリズムを紹介し，これらのアルゴリズムを用いた評価を行って，キャッシュブロックのデータに対する圧縮に有効な圧縮アルゴリズムを見つけ出す．また，ブロックの圧縮サイズの傾向に合わせ，電源制御の粒度を改善することでより多くのブロックに対して圧縮を成功させ，電力削減を行う．

本研究では，これらの評価を CPU シミュレータを用いた実験により行う．また，評価対象のプログラムとして SPECint95[7] ベンチマークを用いる．

実験の結果，評価対象プログラム全てにおいて X-RL アルゴリズムが高い圧縮率を獲得している．消費電力の観点から見ても X-RL アルゴリズムの結果が最も良く，非圧縮実行と比べて平均で約 30%の電力を削減している．

# 目 次

# 第1章　はじめに

## 1.1　背景

### プロセッサにおける消費電力問題

近年，プロセッサの消費電力が増大しモバイルコンピュータのバッテリ駆動時間や高性能プロセッサの放熱の問題から，消費電力の削減は重要な課題となっている．特にトランジスタの微細化に伴い，トランジスタがオフの場合にも電流が持続的に流れるリーク電流が増大している．キャッシュメモリが増加傾向にある高性能プロセッサでは，それに付随したリーク電流の増大が問題となっている．

また，プロセッサの処理性能自体を向上させる場合にも消費電力問題が関係してくる．プロセッサの処理性能を向上させるには，動作周波数を向上させることが最も単純である．しかし，半導体チップに用いられている CMOS プロセスは，動作周波数に比例して消費電力が増大する．さらに半導体チップは電力消費により発熱を起こすため，消費電力の増大は発熱の増大につながる．発熱が増大すると回路の動作速度が不安定となり，動作が保証できなくなる問題も出ている．

このように，消費電力問題はそのプロセッサ自身の問題だけではなく，今後のプロセッサの成長にも影響を及ぼすこととなる．

### 付随する問題

プロセッサの消費電力増大は，単にプロセッサ自身の問題に留まらない．プロセッサの消費電力が増大すれば，動作のために必要な電力要求が増大する．使用する電力が増大するということは，電気料金がそれまで以上に掛かることとなり，経費の増加につながる．また金銭的な面以外でも，供給する電力自体の生成も増やさざるを得ないという問題が発生する．生成する電力を増やすということは，それを作り出すために必要な化石燃料の量も増大することを表しており，エネルギー問題へと発展していく．化石燃料の枯渇が問題となってきている現代において，その使用を加速させる要因はなるべく取り除くべきである．さらには，化石燃料を燃焼させる際に発生する二酸化炭素の増加により，地球温暖化に拍車をかけるという環境問題にも関連してくる．

世界中で数億台の PC が使われている現代社会において，それに使われているプロセッサの消費電力増大は，付随して様々な社会問題を引き起こす原因となる．それらを防ぐ上

でも，プロセッサの低消費電力化は早急に取り掛からなければならない大きな問題であるといえる．

## 1.2 目的

プロセッサにおける消費電力問題を解決するため，様々な低消費電力化の研究が行われている．例えば，プロセッサの処理性能を向上させながら消費電力の増大を防ぐ解決策としてプロセッサのマルチコア化が挙げられる．複数のプロセッサコアに負荷を分散させることで消費電力の増大を防ぎ，かつシングルコアでは達成できないような高い性能を実現することが可能である．ただし，この解決策では半導体チップの微細化を促し，リーク電流が増大する問題がある．

また，電力削減の一つとしてキャッシュの消費電力削減が注目されている．その中にL2キャッシュブロックに格納されるデータに対してデータ圧縮を掛け，格納するデータを小さくすることによりキャッシュ内で使用される領域を削減し，空いた領域の電力をオフにする研究がある [1]．キャッシュの使用領域を削減し，電力供給をオフにすることができれば，キャッシュにおける大幅な電力削減を期待することができる．

本研究では，この研究のデータ圧縮の部分に様々なデータ圧縮アルゴリズムの適用を行う．複数の圧縮アルゴリズムを適用することで，アルゴリズムごとのデータ圧縮結果の違いが出るはずである．これにより，キャッシュに格納されるデータの圧縮に適した圧縮アルゴリズムを見つける．また，圧縮可と判断するための圧縮の粒度についても調査を行う．ブロック単位のデータに対して圧縮アルゴリズムを適用しデータサイズが一定サイズを下回った場合に圧縮可とし，圧縮された形でキャッシュへデータを格納する．先行研究ではデータサイズが 1/2 以下の場合に圧縮可とし，それ以外では圧縮できないとしていたが，本研究ではこの圧縮可能サイズについても複数段階用いる．これらについて調査を行うことで，L2 キャッシュの低消費電力化を目指す．

## 1.3 本論文の構成

本論文は 5 章からなる．第 2 章ではキャッシュ低消費電力化についての関連研究を示す．第 3 章では本研究で扱う圧縮アルゴリズムの説明，提案手法を示す．第 4 章では実際に CPU シミュレータを用いた実験方法の説明と評価について示す．第 5 章で本論文をまとめる．

# 第2章　関連研究

本章では，本研究の関連研究についての説明を行う．

## 2.1　Gated-Vdd[2]

Gated-Vdd は，キャッシュブロックに供給される電力を制御する手法である．図 2.1 に構成図を示す．キャッシュメモリに用いられる SRAM セルと GND との間に閾値の高い Gated-Vdd トランジスタを設け，これをオフにすることで電源供給を断ち，リーク電流を削減する．Gated-Vdd トランジスタは複数セルで共有することができるが，1 つあたりにどれだけ割り当てるかは，回路面積や動作速度とのトレードオフで決める．Gated-Vdd は電源供給がオフになるためリーク電流を大きく削減することができる．しかし，電源をオフにするためセル内の情報が失われ，キャッシュミスが増加し，性能ペナルティがあるという欠点を持つ．

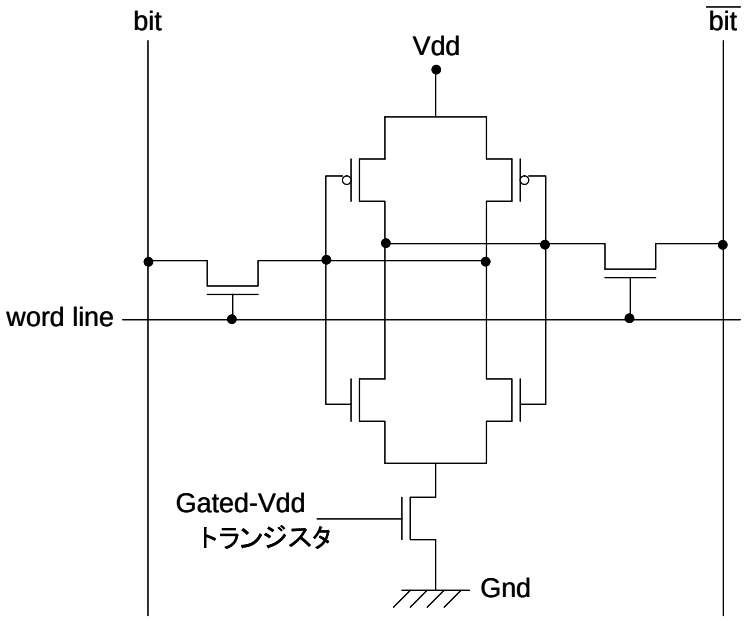


図 2.1: Gated-Vdd

## 2.2 Cache Decay[3]

Gated-Vdd を用いた研究の一つとして Cache Decay が挙げられる．Cache Decay では，あるブロックがキャッシュに格納された場合に，格納されてから最後にアクセスされるまでの時間を Live time，最後にアクセスされてからリプレース対象となりキャッシュから追い出されるまでの時間を Dead time と定義する．この Dead time に入ったキャッシュブロックに対して Gated-Vdd を用いることにより，電力供給をオフにし電力削減を行っている．図 2.2 において LH がブロック A に対する Last Access であり，ブロック B が格納される際にブロック A がリプレース対象となってキャッシュから追い出されることが分かっているならば，LH の後すぐにキャッシュの電源をオフにするのが最も効果的である．しかし実際には Last Access を知ることはできないため，Cache Decay では一定サイクルアクセスのなかったブロックに対して電力供給の遮断を行っている．そのため，まだ Live time 中であるブロックに対して Gated-Vdd が適用されていた場合，本来ならば必要なかったキャッシュミスが発生し，性能低下を引き起こす．

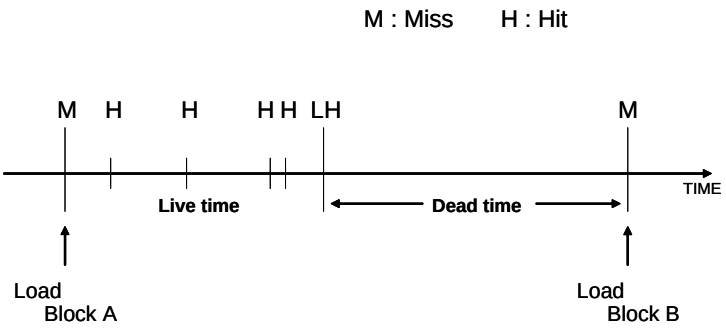


図 2.2: キャッシュ参照の流れ

## 2.3 データ圧縮を用いた Gated-Vdd による電力削減法 [1]

単に Gated-Vdd を用いてキャッシュブロックの電源をオフにすると，ブロック内のデータは損失する．すると再度そのブロックに対してのアクセスが行われた場合，本来ならば無かったはずのキャッシュミスが起こり，性能ペナルティが発生する．この問題を回避しながら消費電力を削減する手法として，L2 キャッシュブロック内のデータを圧縮する手法がある．構成図を図 2.3 に示す．この研究では，圧縮によってキャッシュブロックに格納されるデータが 1/2 以下に圧縮できた場合にはそのデータが圧縮可能であるとし，圧縮した形でデータをキャッシュに格納する．そうでない場合は圧縮不可とし，非圧縮の状態でデータをキャッシュに格納する．データが圧縮できた場合，圧縮により空いた領域に対する電力供給をオフにすることで消費電力の削減を行う．またこの方法では，キャッシュ

ブロック内のデータが損失することが無いため，データ損失によるキャッシュミス増加は発生しない．反対に，データ圧縮を用いる際に考慮すべきことは，圧縮・解凍にかかる時間とその際に必要な電力である．これらのペナルティをできるだけ少なく抑えつつ，電力削減を行う必要がある．この研究ではL2キャッシュに格納されるデータを圧縮する際，Frequent Pattern Compression[4]というデータ圧縮アルゴリズムが用いられている．

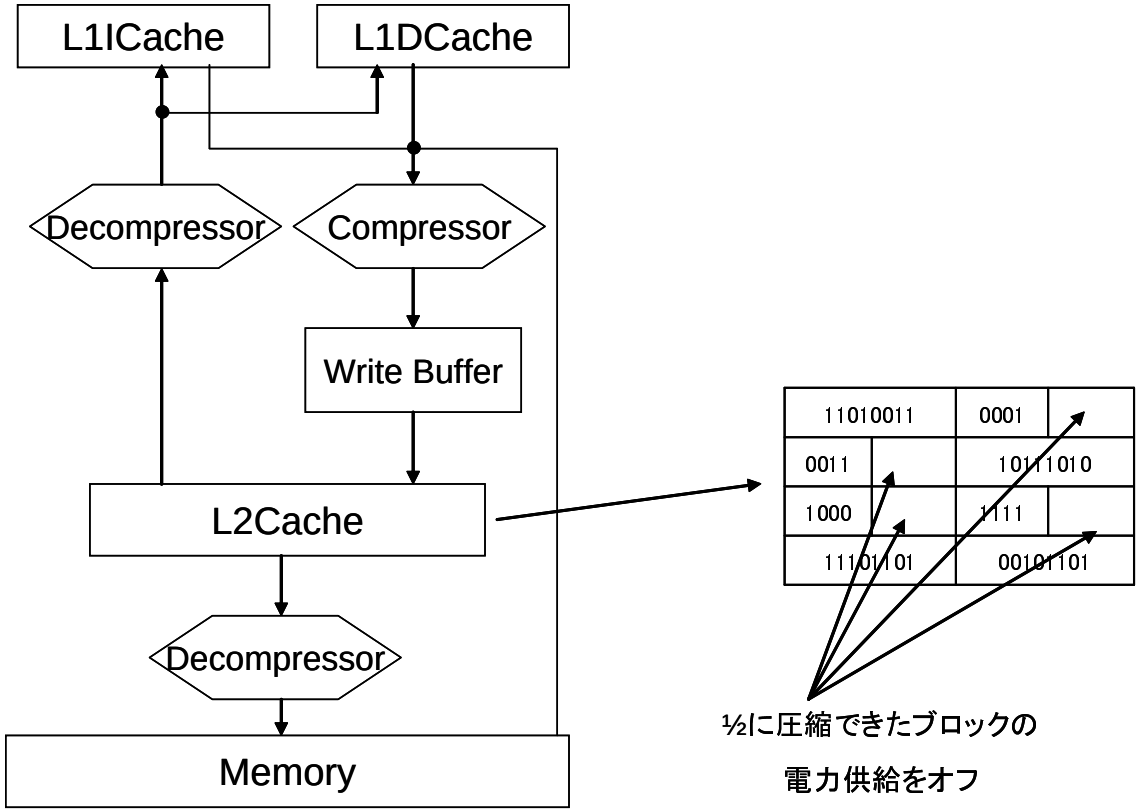


図 2.3: L2キャッシュに対するデータ圧縮

# 第3章　データ圧縮によるキャッシュメモリの低消費電力化

本章では，データ圧縮によるキャッシュメモリの低消費電力化について述べる．始めに本手法の概要について，続いてキャッシュブロック圧縮で実際に用いる各々の圧縮アルゴリズムについての説明，また実際に圧縮されたデータをどのように扱うかについての説明を行う．

## 3.1　概要

本研究ではL2キャッシュにおける電力削減を行う．Gated-Vdd方式を利用し，電力削減の対象となるL2キャッシュに格納されるデータに対して，キャッシュへのデータ格納時にデータの圧縮を試みる．データが圧縮できた場合には圧縮により空いた領域の電源供給をGated-Vddによりオフにし，圧縮できなかった場合にはそのままの形で格納する．圧縮された形でL2キャッシュに格納されているデータがL1キャッシュに格納される，または主記憶へ追い出される際にはデータの復元を行い，非圧縮のデータに戻す作業を行う．

本研究では，L2キャッシュのブロックに格納されるデータを圧縮するのにより有効な圧縮アルゴリズムを見つけることを目的とする．そのため，複数の圧縮アルゴリズムを使用し，それぞれに対しての評価を行う．

## 3.2　キャッシュブロックの圧縮に用いるアルゴリズム

L2キャッシュに格納されるデータを圧縮する際に，実際に用いる圧縮アルゴリズムについての説明を行う．本研究では評価対象の圧縮アルゴリズムとして，Frequent Pattern Compression，Frequent Value Compression，X-matchアルゴリズムとその改良版であるX-RLアルゴリズムの4つを用いる．ここでは，それぞれの圧縮方式について説明を行う．

### 3.2.1　Frequent Pattern Compression[4]

Frequent Pattern Compression(FPC)は1word(32bit)のデータに対して，ビットパターンを元につくられた8つの圧縮規則を用いて圧縮を行うアルゴリズムである．圧縮対象と

なるデータを圧縮規則と照らし合わせ，最も小さいデータに圧縮できる規則を適用して圧縮を行う．圧縮されたデータには適用した圧縮規則を表す prefix が付加される．
表 3.1 に 8 つの圧縮規則を示す.

表 3.1: Frequent Pattern Encoding

| prefix | Pattern Encoded | Data Size |
|---|---|---|
| 000 | Zero Run | 3 bits |
| 001 | 4-bit sign-extended | 4 bits |
| 010 | One byte sign-extended | 8 bits |
| 011 | halfword sign-extended | 16 bits |
| 100 | halfword padded with a zero halfword | 16 bits |
| 101 | Two halfwords, each a byte sign-extended | 16 bits |
| 110 | word consisting of repeated bytes | 8 bits |
| 111 | Uncompressed word | 32 bits |

- Zero Run
  圧縮対象の word のデータが 0 の場合にこの規則が用いられる．Data Size の 3bit はカウンタとして用いられる 3bit であり，連続 8word までの値が 0 である word をこの 3bit のカウンタで圧縮することができる.

- 4-bit sign-extended
  圧縮対象の word が 4-bit の符号拡張の場合に用いられる．Data Size の 4bit は，圧縮対象となったデータの下位 4bit を指す.

- One byte sign-extended
  圧縮対象の word が 1byte の符号拡張の場合に用いられる．Data Size の 8bit は，圧縮対象となったデータの下位 8bit を指す.

- halfword sign-extended
  圧縮対象の word が 16bit の符号拡張の場合に用いられる．Data Size の 16bit は，圧縮対象となったデータの下位 16bit を指す.

- halfword padded with a zero halfword
  圧縮対象の word の下位 16bit が 0 の場合に用いられる．Data Size の 16bit は，圧縮対象となったデータの上位 16bit を指す.

- Two halfwords, each a byte sign-extended
  圧縮対象の word の上位 16bit，下位 16bit のそれぞれが 1byte の符号拡張の場合に

用いられる．Data Size の 16bit は，圧縮対象となったデータの上位 16bit における下位 8bit と，下位 16bit における下位 8bit を指す．

- word consisting of repeated bytes
  圧縮対象の word がある 1byte データの連続であった場合に用いられる．Data Size の 8bit は，連続しているデータ 1byte を指す．

- uncompressed word
  圧縮対象の word がどの規則も適用できなかった場合，非圧縮データとなりこの規則が用いられる．Data Size の 32bit は，圧縮対象となったデータそのものを指す．

FPC による圧縮例を図 3.1 に示す．

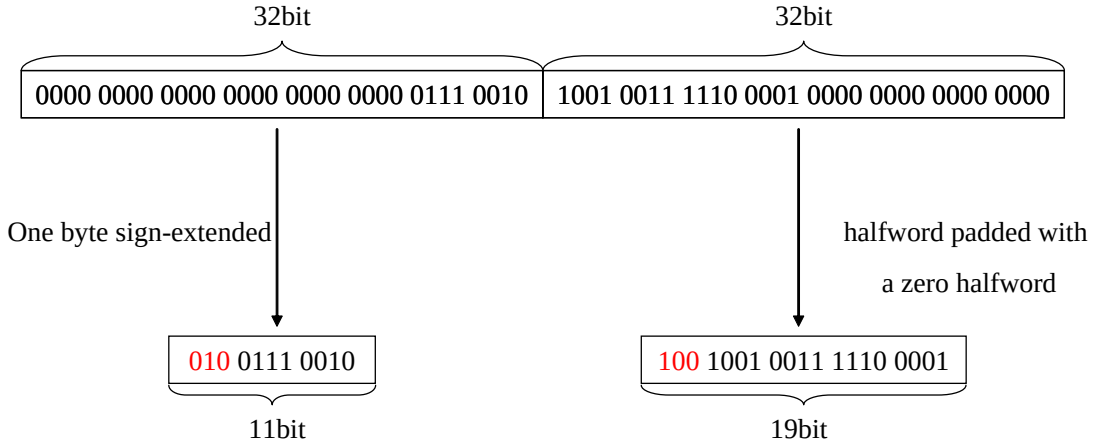


図 3.1: Frequent Pattern Compression 圧縮例

図 3.1 は 2word のデータに対して FPC を適用した例である．1word 目のデータに対しては One byte sign-extended が適用されている．圧縮後のデータは，適用規則を表す prefix 010 と圧縮対象のデータの下位 8bit により構成される．2word 目のデータに対しては halfword padded with a zero halfword が適用されている．圧縮後のデータは，適用規則を表す prefix 100 と圧縮対象のデータの上位 16bit で構成されている．元データのサイズは共に 32bit であったが，FPC の適用によりそれぞれのデータサイズが 11bit，19bit に圧縮されている．

FPC により圧縮されたデータを復元するには，そのデータの先頭から 3bit の prefix を調べ，どの規則が適用されているか判別することで元データを復元することができる．

### 3.2.2 Frequent Value Compression[5]

Frequent Value Compression(FVC) は，そのプログラム中で頻繁に使われる値 (Frequent Value : FV) を用いて圧縮を行うアルゴリズムである．FVC では，プログラム実行の序盤

にメモリアクセスを監視する期間を設けることでそのプログラム中で実際に使用されたデータを記録しておき，監視期間終了時にその記録から頻繁に使われていたデータを選び出し，FV とする．この選び出された FV に対して ID を設定することで，ID を用いた圧縮を行う手法である．
FVC を用いるためには，

- プログラム実行中にメモリアクセスで用いられる値を監視する Finder
- 監視して得られた FV を実際に圧縮に用いる Encoder

の実装が必要となる．J.Yang らの論文 [5] では，この 2 つの機能を両方有する機構が提案されている．
概略図を図 3.2 に示す．

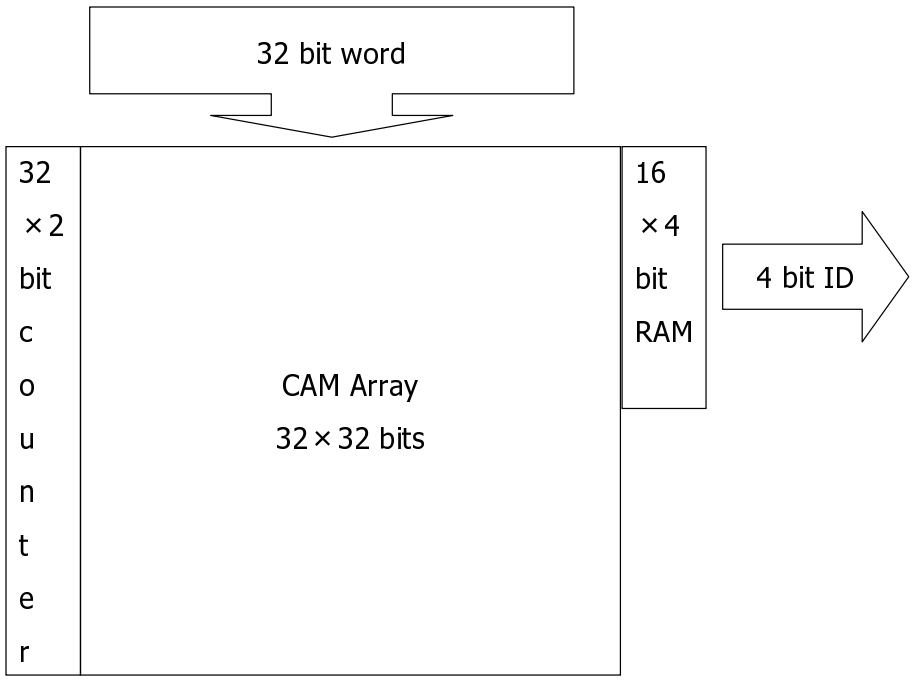


図 3.2: Finder と Encoder の複合機構

この機構は，メモリアクセスの監視期間には Finder として，監視期間終了後には Encoder として機能する．図 3.2 は，FV を 16 種類得るために必要なハードウェア構成である．この例の場合は 32 エントリの CAM Array，CAM Array のエントリと対になる 2bit counter，そして CAM Array の上位 16 エントリと対になる 4bit の RAM となる．この機構は監視期間中に以下の動作をしながら FV を決定する．

- プログラム実行中にメモリアクセスの対象となった値を CAM Array の最上位空きエントリに格納する．もし，対象となった値が CAM Array 内のエントリに既に存在していた場合は，そのエントリの counter をインクリメントする．

- counter が飽和している (counter の値が 11) エントリの値がメモリアクセス対象となった場合，そのエントリと一つ上のエントリの counter と CAM Array の内容をスワップする．

- CAM Array のエントリが全て埋まっている状態で新しい値がメモリアクセスの対象となった場合，CAM Array の下位半分 (図 3.2 では下位 16 エントリ) の中で最も counter の値が小さいエントリを追い出し，格納する．

監視期間終了時に CAM Array の上位半分 (図 3.2 では上位 16 エントリ) に存在している値を FV として扱う．CAM Array の上位半分には対となる RAM が用意されており，それぞれのエントリが一意的になるように ID をつける．図 3.2 の例では，16 個の ID が必要となるので 4bit の RAM が用意されている．
圧縮を行う場合，圧縮対象のデータを Encoder の入力とし，一致するエントリを検索する．データが Encoder のエントリと一致した場合，そのエントリに定められた ID と置換することで圧縮データとする．この方式では圧縮後のデータに prefix を 1bit 付加することになる．この prefix によりそのデータが圧縮されているデータか，非圧縮のデータかを判別する．

圧縮されたキャッシュブロックに対してメモリアクセスがあった場合は，データに付加された prefix を元に復元を行う．prefix が非圧縮データを表しているならば，それに続くデータは実際のデータそのものとなる．prefix が圧縮データを表しているのなら prefix に続くデータは ID である．一致する ID を持つエントリを検索し，対応した FV と置き換えることで復元することができる．

図 3.3 に FV の圧縮例を示す.

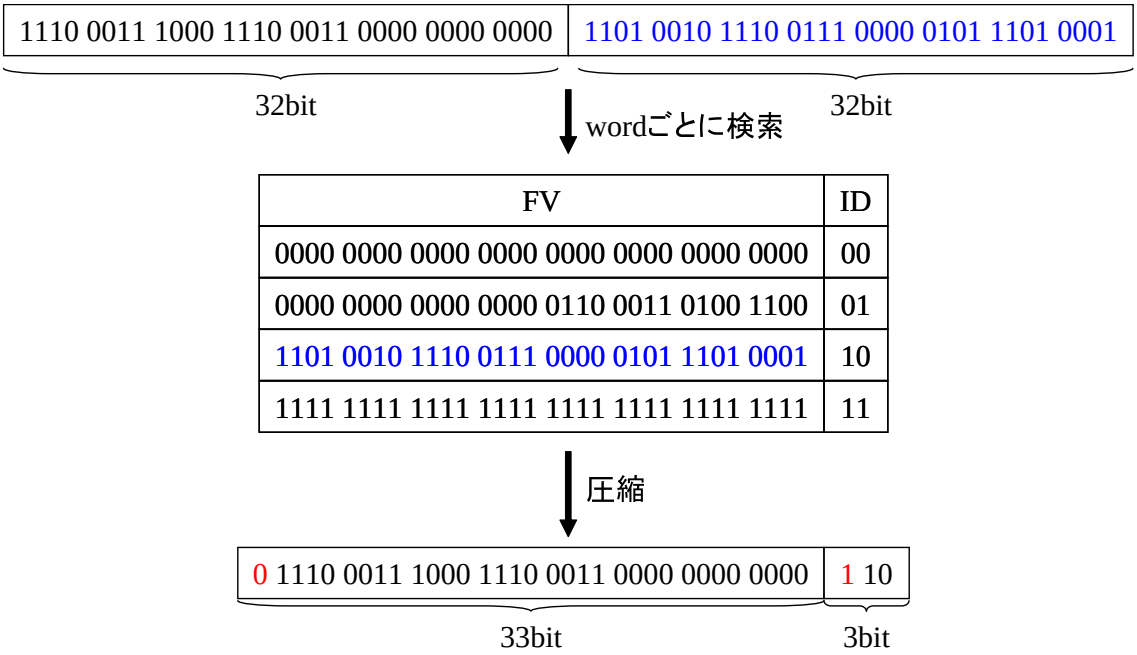


図 3.3: Frequent Value Compression 圧縮例

図 3.3 は 2word のデータに対して FVC を適用して圧縮を行った例である．まず，1word 目のデータと一致するエントリをエンコーダから検索する．この場合，エンコーダには一致するエントリが存在していないため，1word 目のデータは圧縮不可となる．結果，圧縮・非圧縮を表す prefix が 1bit 付加され，32bit のデータが 33bit になる．次に，2word 目のデータと一致するエントリをエンコーダから検索する．2word 目のデータの場合は一致するエントリがエンコーダに存在しているため，圧縮可となる．この例の場合，一致するエントリと対応した ID は 10 の 2bit なので，結果，prefix と合わせて 32bit が 3bit に圧縮できる．

選び出す FV の個数はハードウェアが許す限り，静的に変えることができるが，

- FV 数・多
  - データが圧縮できる確率が上がる
  - ID を表すのに必要な bit フィールドが大きくなるため，圧縮効率が下がる
- FV 数・少
  - データが圧縮できる確率が下がる
  - ID を表すのに必要な bit フィールドが小さくなるため，圧縮効率が上がる

ということを考慮しなくてはならない.

### 3.2.3 X-Match[6]

#### X-Match アルゴリズム

X-Match は dictionary とよばれる CAM Array を用いることで，過去に参照されたデータを用いて圧縮を行うアルゴリズムである．圧縮を行うキャッシュブロックを word ごとに順番に dictionary に入力として与え，データが一致するエントリを検索する．
データの一致には，次の 2 つの概念を用いる．

- full match
  入力データと dictionary 内のあるエントリのデータが完全に一致

- partial match
  入力データと dictionary 内のあるエントリのデータが 2byte・3byte の部分一致

データを入力として dictionary を検索し，full match もしくは partial match した場合，そのデータを圧縮した形に置き換えることになる．dictionary は毎データごとに更新される必要があり，更新方法は full match の場合と，partial match もしくは match しなかった場合の 2 通りに分かれる．

full match した場合では，full match したエントリ (match エントリと呼ぶ) を dictionary の最上位エントリへ移動させる．その時，match エントリよりも上位に存在していたエントリを 1 つずつ下位エントリに移動させることで最上位エントリを空ける．partial match・match なしの場合，dictionary 内に存在している全てのエントリを 1 つずつ下位エントリへ移動し，最上位エントリに圧縮対象のデータを挿入する．結果としてどちらも，最上位エントリが今検索に用いられたデータになるように更新している．

X-Match で扱う dictionary は，毎圧縮単位，本研究では 1 ブロックごとに内容を消去して，毎回初期状態にして圧縮を開始する．ブロック内のデータのみで圧縮を行うことで，復元もブロック単体で行うことが可能となる．

データ圧縮が成功した場合，圧縮できた場合にセットされる match フラグ，match エントリを指す Address，1word を byte ずつに分けたときにどの byte が match しているのかを表す Match Type，match していない部分の値を表す Literals が出力される．full match の場合は Literals が必要ないため，出力されない．圧縮できなかった場合の出力は，圧縮できなかったことを表す match フラグ (=0) と，データそのものである．それぞれの場合について，例を用いて説明する．

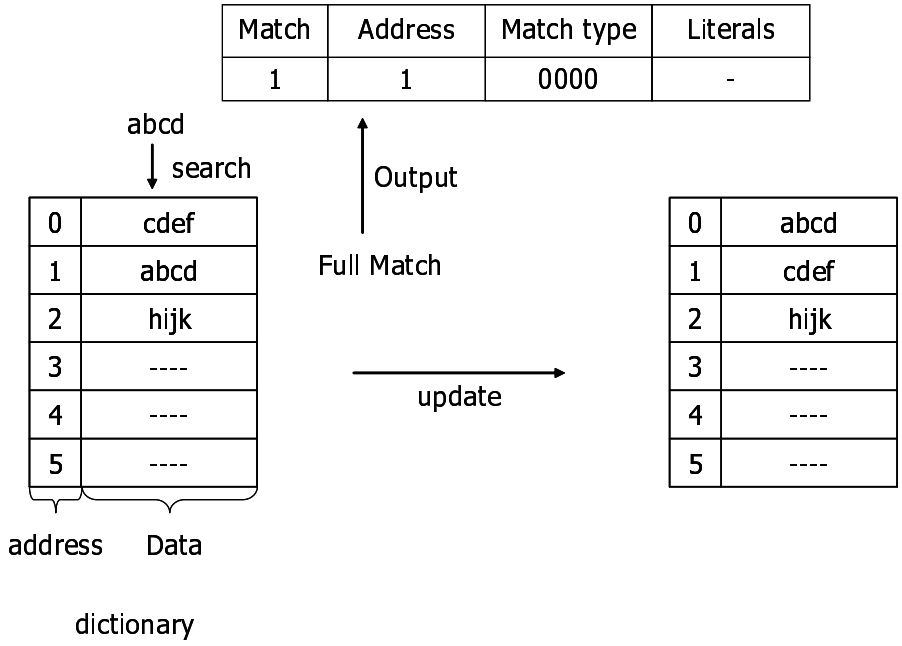


図 3.4: X-Match：full match 圧縮例

**full match**

図 3.4 は full match した場合の圧縮例である．dictionary は既にデータが 3 エントリ存在している状態である．この例では，圧縮対象である abcd を dictionary で検索した場合，Address 1 と full match する．このときの出力は，match を表す match フラグは 1，match エントリを指す Address は 1，match した byte を表す match type は 0000 となる．full match なので，Literals は出力されない．出力が確定した後，match エントリが最上位エントリになるように dictionary の更新を行う．

**partial match**

図 3.5 は 3byte partial match の例である．図 3.4 と同様，dictionary は既にデータが 3 エントリ存在している状態である．圧縮対象である abce を dictionary で検索で検索した場合，Address 1 と partial match する．このときの出力は，match を表す match フラグは 1，match エントリを指す Address は 1，match type は match していない byte に対応している bit がセットされた形の 0001，そして match していない値 e が Literals として出力される．出力確定後，検索したデータが最上位エントリになるように dictionary の更新を行う．

データの復元は，dictionary を用いて圧縮時の dictionary を再生成することで可能となる．復元の流れを説明する．まず圧縮データの match フラグの値を見ることで，セットであれば match しているデータ，そうでなければ match しなかったデータであることが

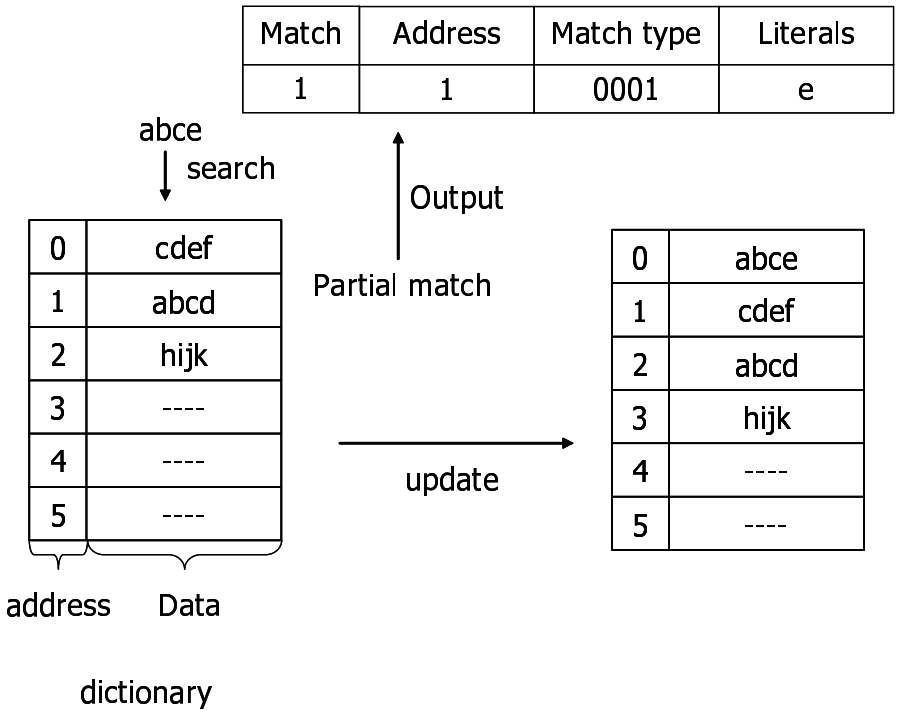


図 3.5: X-Match：partial match 圧縮例

分かる．match しなかったデータの場合，flag に続くデータが元データそのものなので，その word に関しての復元は終了となる．このとき match しなかったデータが dictionary の最上位エントリになるように dictionary の更新を行う．match フラグの値から match したデータと判断された場合，次に続くデータは match したエントリを指す Address となる．続く match type で full match・partial match の区別が可能となる．full match の場合，Address の指すエントリのデータが復元するデータとなり復元終了となる．このときに圧縮時と同様，match エントリを dictionary の最上位エントリとなるように更新する．match type が partial match を表していた場合，続くデータが match エントリと一致していない部分を保持した Literals となるため，match エントリと Literals により復元が可能となる．partial match の場合，復元したデータが dictionary の最上位エントリとなるように更新を行う．以上が復元時の流れとなる．

**Phasing in binary codes**

$x$ 個エントリの Address を表すのに必要な bit 数は，通常 $\lceil \log_2 x \rceil$bit である．例えば，9 エントリしかない場合に全てのエントリを 4bit を使用して表すと 6 通りの組み合わせが使われないことになり，無駄が生じる．有効エントリが毎圧縮ごとに可変となる X-Match アルゴリズムでは，phasing in binary codes という方法を用いてなるべく無駄がでないように Address 値の変換を行っている．

ある値 $i(0 \leq i < \rho)$ を表すのに通常必要な bit 数は，$k = \lceil \log_2 \rho \rceil$bit である．このとき，phasing in binary codes では $i < 2^k - \rho$ のとき，$k-1$bit で，そうでない場合は $k$bit で表す．$i$ が $k-1$bit で表すことが出来る場合はそのままのコード値，$k$bit で表す場合コード値は $i + 2^k - \rho$ となる．X-Match アルゴリズムにおいて，$i$ は match エントリの Address，$\rho$ はそのときの有効エントリ数となる．
図 3.6 は，0 から 8 を表す phasing in binary codes 例である

| Integer | Code |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 1110 |
| 8 | 1111 |

図 3.6: phasing in binary codes 例

### X-RL アルゴリズム

X-RL アルゴリズムは，X-Match アルゴリズムに機能を追加したものである．追加された機能とは，圧縮中の dictionary への検索対象にゼロが連続して現れた場合，その個数を圧縮に用いるという仕組みである．この仕組みを説明するにあたって，X-Match からの変更点を説明する．

一つ目の変更点は，圧縮を開始する際の dictionary の初期状態である．X-Match における dictionary の初期状態は全てのエントリの内容が空，すなわち意味のあるエントリが存在しない状態を指し，有効エントリ数は 0 である．これに対して X-RL では，最上位エントリにゼロをセット，次のエントリを reserved エントリとし有効エントリ数を 2 とする．最上位エントリにゼロをセットしておくことで，ゼロが入力として現れた場合に即座に full match することができるため，圧縮効率が良くなる．しかし，ゼロが圧縮対象データに存在しなかった場合にはエントリ数の増加により address が増えるため，圧縮効率が悪

くなる．

二つ目の変更点は，Run Length Internal counter(RLI counter) の存在である．この counter は値がゼロの入力が連続した回数をカウントするために用意される．

RLI counter を使った動作は，dictionary の最上位エントリと full match をしたとき，そのエントリのデータがゼロである場合に開始する．図 3.7 にその例を示す．

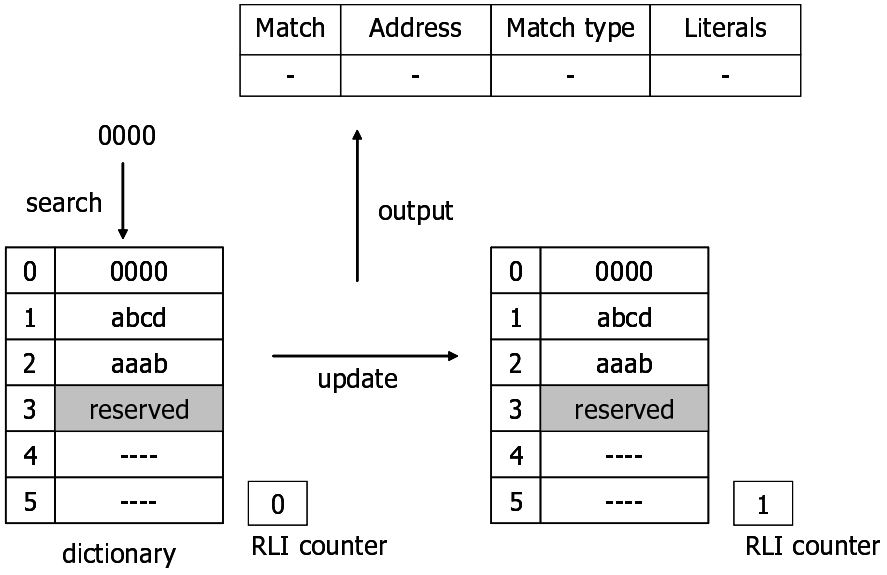


図 3.7: RLI counter 使用例

最上位エントリがゼロであり，かつ圧縮対象の値もゼロのとき，RLI counter のカウントを開始する．このときには圧縮結果の出力は無く，RLI counter がインクリメントされるのみとなる．ゼロが圧縮対象として連続する限り最上位エントリとの full match となり，dictionary の更新はされずに RLI counter がインクリメントされ続ける．

0 以外の値が圧縮対象となったら RLI counter のカウントが終了し，RLI counter の値を利用して出力が生成される．図 3.8 では RLI counter が 3 で終了しているので，この出力で 3word 分の 0 を表すことになる．まず，match を表す match フラグを出力する．続く出力は Address であるが，通常の full match の場合は full match しているエントリを指す．しかし，この場合は RLI counter を使用していることを表すため最下位有効エントリに用意された reserved エントリを指すようにする．reserved エントリのデータは不定でも良いが，通常は 0 として用意しておく．最後に RLI counter でカウントされたデータを出力 (Run Length フィールド) して圧縮終了となる．圧縮が終了したら，RLI counter の値は初期化される．

圧縮データの復元方法は基本的には X-Match と同様であるが，セットの match フラグを持つ圧縮データの Address がそのときの dictionary の最下位有効エントリを指している場合，続くデータは RLI counter によるカウントとなる．復元時に Run Length フィールド

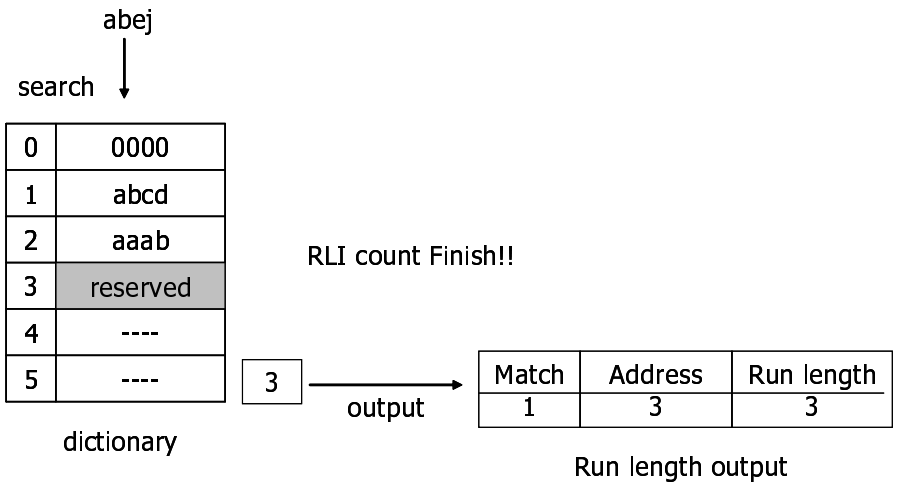


図 3.8: RLI counter 使用時の出力例

を取得する際，何 bit で構成されているフィールドであるかを予め知っていなければ，復元が困難となる．そのため，Run Length フィールドが何 bit で構成されているか，すなわち何 word まで 1 度に表すことが出来るかを決めておくことが必要である．この値は圧縮効率を考えた上でのトレードオフとなる．

## 3.3 電源制御

### 3.3.1 データ圧縮の粒度

データ圧縮を用いた Gated-Vdd 制御 [1] ではキャッシュブロックデータを L2 キャッシュに格納する際，ブロックサイズの 1/2 以下に圧縮できた場合は圧縮した形で格納し，ブロックの空き領域 1/2 に対して Gated-Vdd により電源供給をオフとし，圧縮したデータがブロックサイズの 1/2 以上になった場合は非圧縮の形で格納していた．

図 3.9 は，SPECint95[7] の 1 つである 099.go を 20 億命令実行して得たデータである．このグラフは L2 キャッシュに格納されるブロックデータに対して Frequent Pattern Compression を行い，圧縮後のデータサイズの分布 (2Byte ずつ) を表したものである．ここで 1 ブロックのサイズは 32Byte としている．

このグラフより，ブロックサイズの 1/2 以下，つまり 16Byte 以下に圧縮できたブロックは全体の約 50%である．しかし，このグラフを見ると 16Byte から 20Byte 間のブロック数が約 45%存在している．また，1/2 以下に圧縮されている範囲に着目すると，ブロックサイズが 4Byte 未満，つまりブロックサイズの 1/8 以下に圧縮されているブロックが全体の約 10%存在している．

これらの圧縮率を有効に活用するためには，1/2 単位での制御では不十分であると．そ

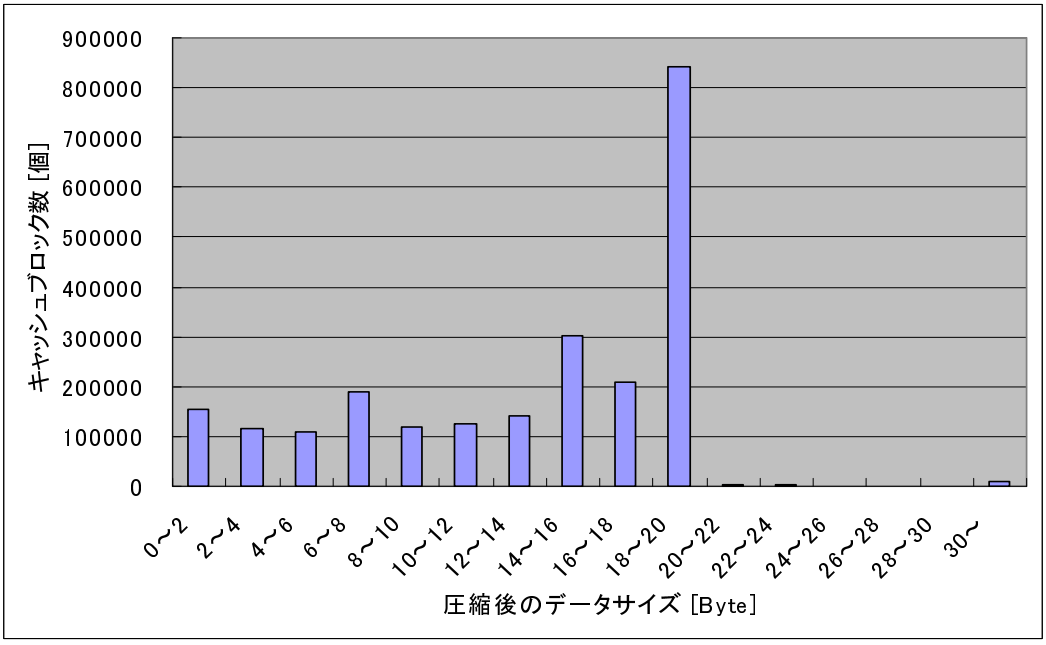


図 3.9: 圧縮されたブロックのサイズ別分布図

こで本研究では，追加ハードウェアとの兼ね合いを考え，1/4 単位での圧縮・電源制御を用いることにする．

### 3.3.2 Gated-Vdd による電源制御

本研究では L2 キャッシュへ格納されるキャッシュブロックデータに対してデータ圧縮を掛ける際，圧縮可能とするサイズをブロックサイズの 3/4 以下・1/2 以下・1/4 以下の 3 段階用い，圧縮サイズを考慮したキャッシュブロックの電源制御を行う．図 3.10 は本研究で用いる電源制御法を実現するのに必要な構成である．キャッシュブロックに格納されるデータサイズは，ブロックサイズの 3/4 以下に圧縮できずに非圧縮の状態で格納されるデータ，そして 3/4 以下・1/2 以下・1/4 以下の計 4 サイズである．L2 キャッシュタグに 2bit の compression control field を用意することで，L2 キャッシュに現在格納されているデータが圧縮されているデータであるかどうかを知るとともに，各々の圧縮サイズに応じた電源制御を行う．表 3.2 は compression control field とデータサイズの対応表である．

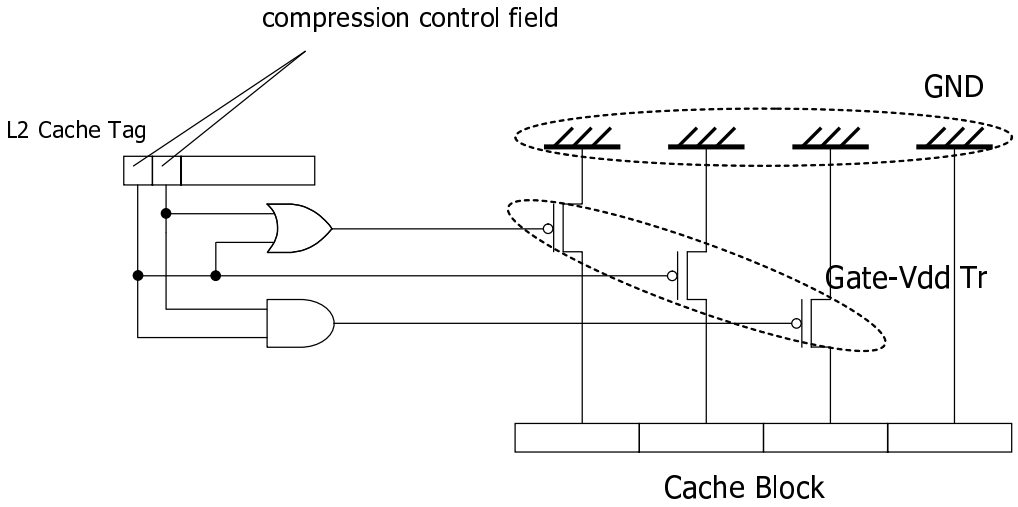


図 3.10: Gated-Vdd による電源制御

表 3.2: compression control field とデータサイズの対応

| compression control field | データサイズ |
|---|---|
| 00 | 非圧縮 |
| 01 | 3/4 |
| 10 | 1/2 |
| 11 | 1/4 |

# 第4章　評価

本章では，各々の圧縮アルゴリズムを用いて圧縮したキャッシュブロックに対しての電源制御による電力削減結果を示すため，CPU シミュレータによるシミュレーション評価を行う．

## 4.1　ベンチマークプログラム

本研究では，評価対象として SPECint95 ベンチマーク [7] を用いる．表 4.1 に評価に用いた SPECint95 のプログラムをまとめる．

表 4.1: SPECint95 Benchmarks

| Benchmark | Detail | Input File |
|---|---|---|
| 099.go | An internationally ranked go-playing program | ref |
| 124.m88ksim | A chip simulator for the Motorola 88100 microprocessor | ref |
| 126.gcc | Based on the GNU C compiler version 2.5.3 | ref |
| 129.compress | A in-memory version of the common UNIX utility | ref |
| 130.li | Xlisp interpreter | ref |
| 132.ijpeg | Image compression/decompression on in-memory images | ref |
| 134.perl | An interpreter for the Perl language | ref |
| 147.vortex | An object oriented database | ref |

プログラム実行の開始 10 億命令を初期化の期間と考え，10 億から 20 億までの 10 億命令のデータを実験結果として扱う．126.gcc のみ実行方法の違いから 20 億命令までのデータを取ることが困難なため，10 億から 18 億までの 8 億命令を結果として扱う．

## 4.2　評価環境

評価に用いる CPU シミュレータは SPARC V9[8] 命令セットアーキテクチャを対象としている．シミュレータは，C 言語で記述された SPECint95 のプログラムをコンパイル

して生成されたバイナリコードを入力とし，プログラム実行を行う．シミュレーションで用いるキャッシュのパラメータを表 4.2 に示す．

表 4.2: CPU シミュレータのキャッシュパラメータ

| | cache size | way | block size |
|---|---|---|---|
| L1 I-Cache | 64KB | 2-way | 32B |
| L1 D-Cache | 64KB | 2-way | 32B |
| L2 Cache | 1MB | 2-way | 32B |

また，L2 キャッシュアクセスレイテンシは 10cycle，主記憶へのアクセスレイテンシは 100 cycle とした．L1 キャッシュから L2 キャッシュへの書き戻し，L2 キャッシュから主記憶への書き戻し共に write back 方式を採用し，置き換え対象のブロックは LRU 法によって選択する．

L2 キャッシュブロックへ格納されるデータを圧縮する，または L2 キャッシュブロック内で圧縮されていたデータを復元する際，圧縮・復元によるレイテンシが発生する．ここで，本研究で定めた各圧縮アルゴリズムのパラメータ表 4.3 に示す．

表 4.3: 圧縮アルゴリズム別パラメータ

| 圧縮アルゴリズム | 圧縮レイテンシ | 復元レイテンシ | 備考 |
|---|---|---|---|
| Frequent Pattern | 10cycle | 10cycle | なし |
| Frequent Value | 10cycle | 10cycle | ・FV 監視期間はプログラム開始 1 億命令<br>・FV 数は 16 個 |
| X-Match | 10cycle | 10cycle | なし |
| X-RL | 10cycle | 10cycle | ・RLI counter は 3bit |

## 4.3 実験結果

実験で得られたデータを SPECint95 のプログラムごとに示し，結果を考察する．ここで挙げるデータは以下通りである．

- 圧縮アルゴリズムごとの実行サイクル数
  電源を 1/2・1/4 で制御したそれぞれについて，10 億命令実行した各アルゴリズムの実行サイクル数を示す．また，圧縮なしで実行した場合の実行サイクル数を 100%とした上で，それに対する各アルゴリズムを用いたときの実行サイクルの割合を示す．

- 圧縮サイズごとのブロック数
  1/4 制御において，10 億命令実行中に圧縮されたブロックのサイズごとの数を示す．
- プログラム実行中の L2 キャッシュ電力使用率
  10 億命令実行した各アルゴリズムの L2 キャッシュに対する電力使用率を比較して示す．また，全てのデータにおいて 1/2 制御と 1/4 制御の比較を行う．

本研究では非圧縮のデータを格納しているキャッシュブロックに使われる電力を 100%として，3/4 に圧縮されたデータを格納しているキャッシュブロックに使われる電力を 75%，1/2 に圧縮されたデータを格納しているキャッシュブロックに使われる電力を 50%，1/4 に圧縮されたデータを格納しているキャッシュブロックに使われる電力を 25%と定める．そして，これに実際にデータが L2 キャッシュに存在し続けた時間的要因，つまりクロックサイクルを用いて消費電力を表す．よって，以下の式で消費電力を表す．

$$\begin{aligned} power\ consumption = fullsize\ block\ total\ cycle\ &*\ 1.0\ + \\ 3/4\ block\ total\ cycle\ &*\ 0.75\ + \\ 1/2\ block\ total\ cycle\ &*\ 0.5\ + \\ 1/4\ block\ total\ cycle\ &*\ 0.25 \end{aligned}$$

また，本研究で用いたシミュレータの設定において，圧縮アルゴリズムの適用によってクロックサイクルが増加する要因は圧縮・復元レイテンシのみである．また，L2 キャッシュへ格納されるデータは全てに対して圧縮を試みるため，圧縮アルゴリズムの違いによる圧縮レイテンシの違いは発生しない．そのため，復元レイテンシによる違いのみがクロックサイクルの増加に影響を与えている．すなわち，最も実行速度の遅い圧縮アルゴリズムは最も多くのブロックに対して圧縮を成功させていることになる．

### 099.go

099.go のプログラム実行において消費電力を最も抑えることが出来た圧縮アルゴリズムは図 4.1 に表されているように X-RL アルゴリズムで，続いて Frequent Pattern Compression，X-Match アルゴリズム，Frequent Value Compression となっている．X-RL は通常のプログラム実行時の電力と比べて，約 35%電力を削減している．また，図 4.2 からも分かるように，Frequent Value Compression 以外のアルゴリズムは，ほぼ 100%のブロックに対してデータ圧縮が成功している．Frequent Value Compression は，約 50%のブロックに対して圧縮が成功しているが，結果的にほとんど電力を削減することができなかった．これは圧縮できたブロックが割と短い期間でリプレース対象となっていたり，圧縮できないデータパターンに更新されているものだと考えられる．最も実行速度が遅かったのも，表 4.4 より X-RL で約 3%増加している．

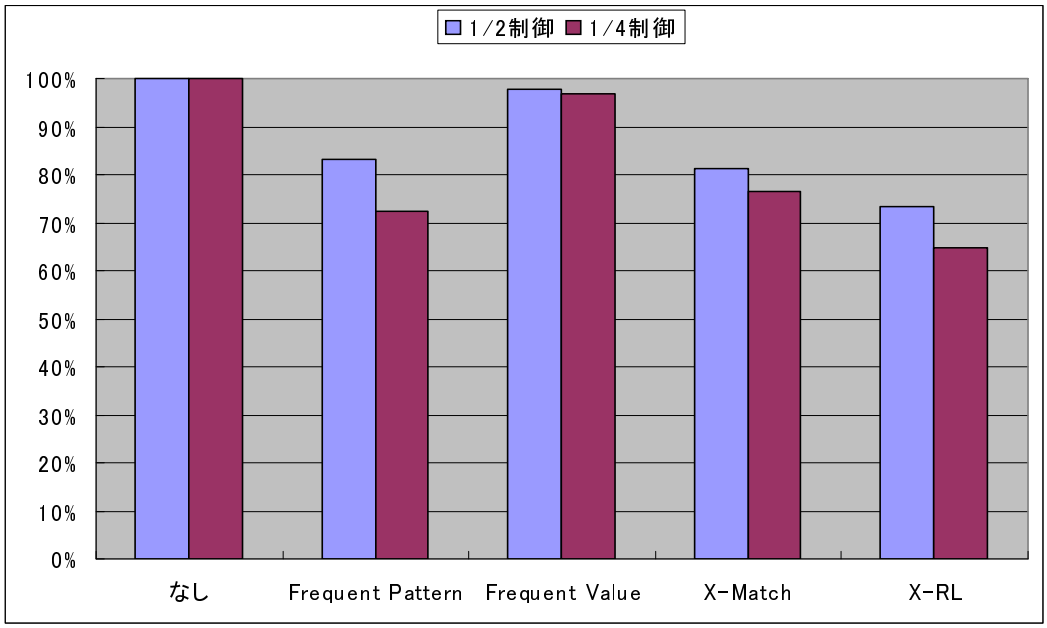


図 4.1: 099.go 圧縮アルゴリズム別消費電力

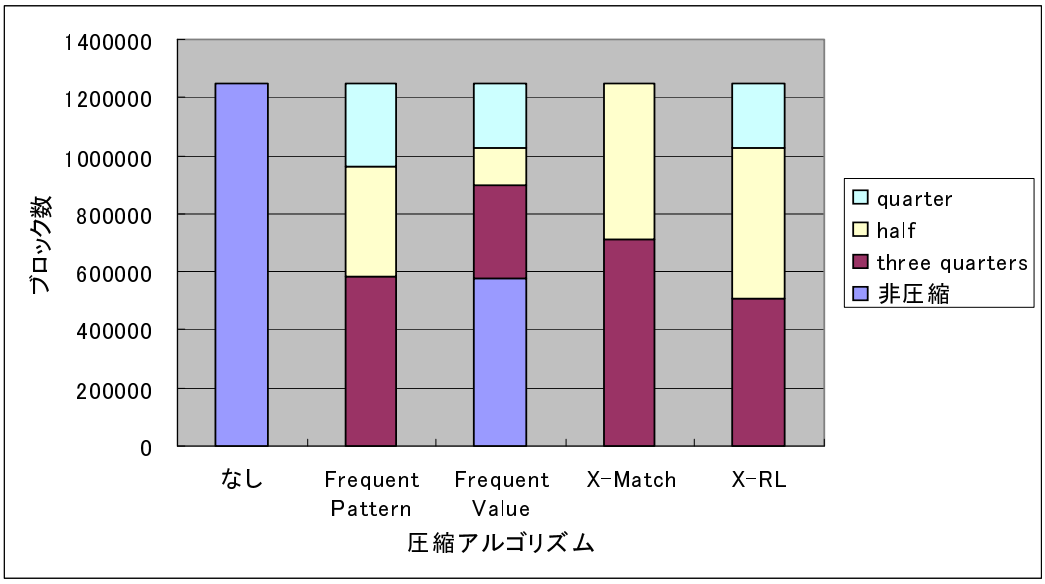


図 4.2: 099.go 圧縮サイズ別ブロック数

表 4.4: 099.go クロックサイクル

| アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 クロックサイクル | 1065860042 | 1075664209 | 1067608829 | 1076639129 | 1079954689 |
| 1/2 実行割合 | 100.00% | 101.86% | 101.17% | 101.87% | 102.26% |
| 1/4 クロックサイクル | 1065860042 | 1100307852 | 1084496292 | 1100693382 | 1100700232 |
| 1/4 実行割合 | 100.00% | 103.23% | 101.75% | 103.27% | 103.27% |

## 124.m88ksim

124.m88ksim のプログラム実行において消費電力を最も抑えることが出来た圧縮アルゴリズムは図 4.3 に表されているように X-RL アルゴリズムで，続いて X-Match アルゴリズム，Frequent Pattern Compression，Frequent Value Compression となっている．X-RL は通常のプログラム実行時の電力と比べて，約 40%電力を削減しており，X-Match もほぼ同程度の電力削減率となっている．圧縮アルゴリズム全般に渡って，124.m88ksim では 1/4 圧縮が成功していない (図 4.4)．実行速度はほぼ通常実行と変わらない速度で，最も遅い X-RL でも約 0.1%の増加となっている (表 4.5)．L2 キャッシュに格納されるブロックが少ないため，リプレースがあまり行われず，圧縮された状態で格納されたブロックが長期間滞在し続けていると思われる．

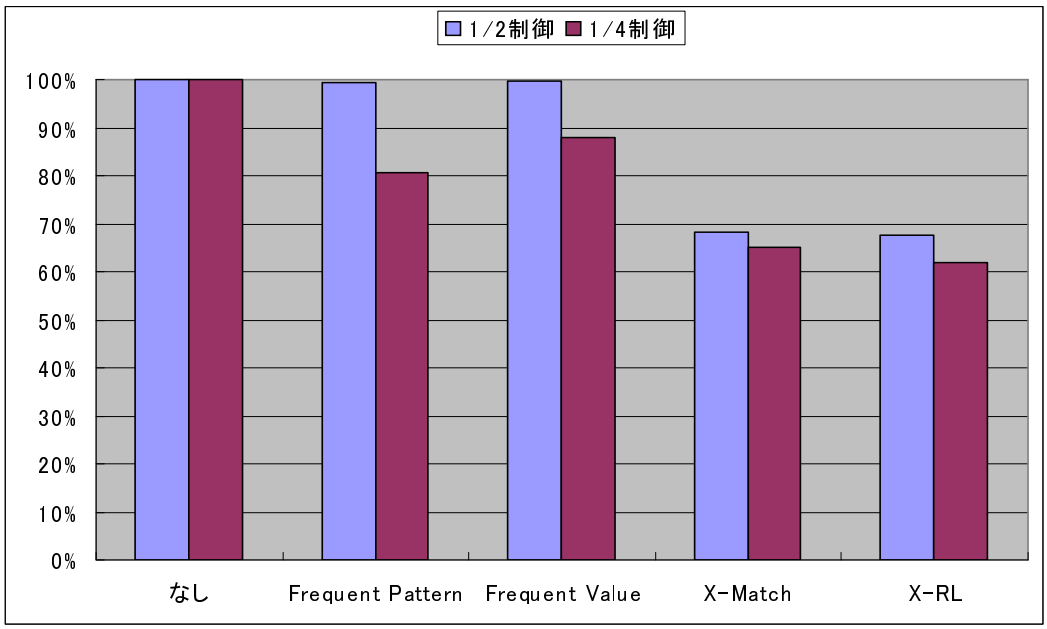


図 4.3: 124.m88ksim 圧縮アルゴリズム別の消費電力

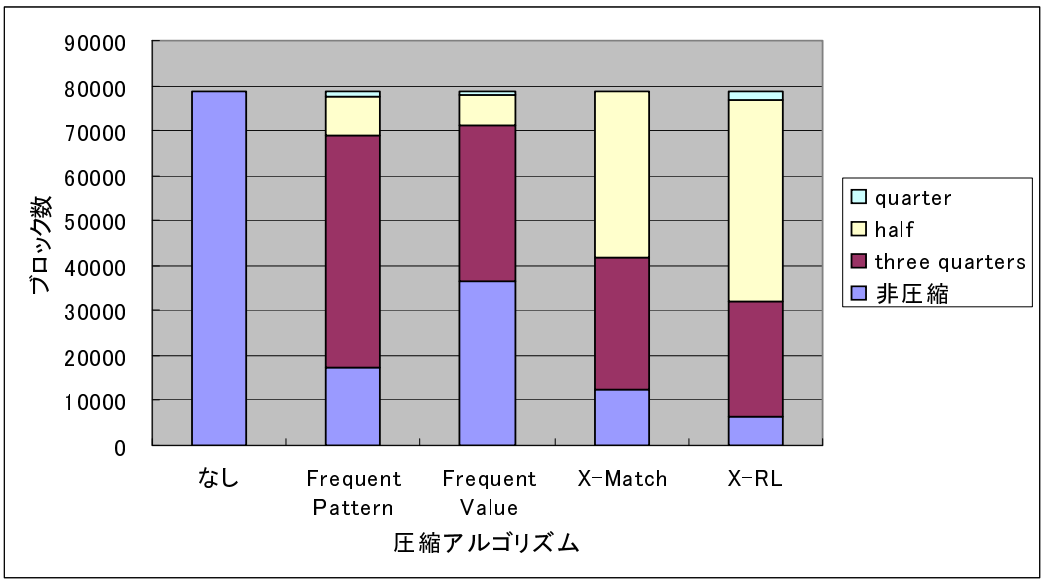


図 4.4: 124.m88ksim 圧縮サイズ別のブロック数

表 4.5: 124.m88ksim クロックサイクル

| アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 クロックサイクル | 1006222610 | 1006825590 | 1006798990 | 1007119640 | 1007220770 |
| 1/2 実行割合 | 100.00% | 100.06% | 100.06% | 100.09% | 100.10% |
| 1/4 クロックサイクル | 1006222610 | 1007324580 | 1007282830 | 1007586960 | 1007605640 |
| 1/4 実行割合 | 100.00% | 100.11% | 100.11% | 100.14% | 100.14% |

### 126.gcc

126.gcc のプログラム実行において消費電力を最も抑えることが出来た圧縮アルゴリズムは図 4.5 に表されているように X-RL アルゴリズムで，続いて Frequent Pattern Compression，X-Match アルゴリズム，Frequent Value Compression となっている．最も電力削減できた X-RL で約 20%の電力削減となっている．最も電力削減できなかった Frequent Value Compression との差が約 10%以内と，大きな違いは見られなかった．実行速度は X-RL が最も遅く，約 3%の増加となっている (表 4.6).

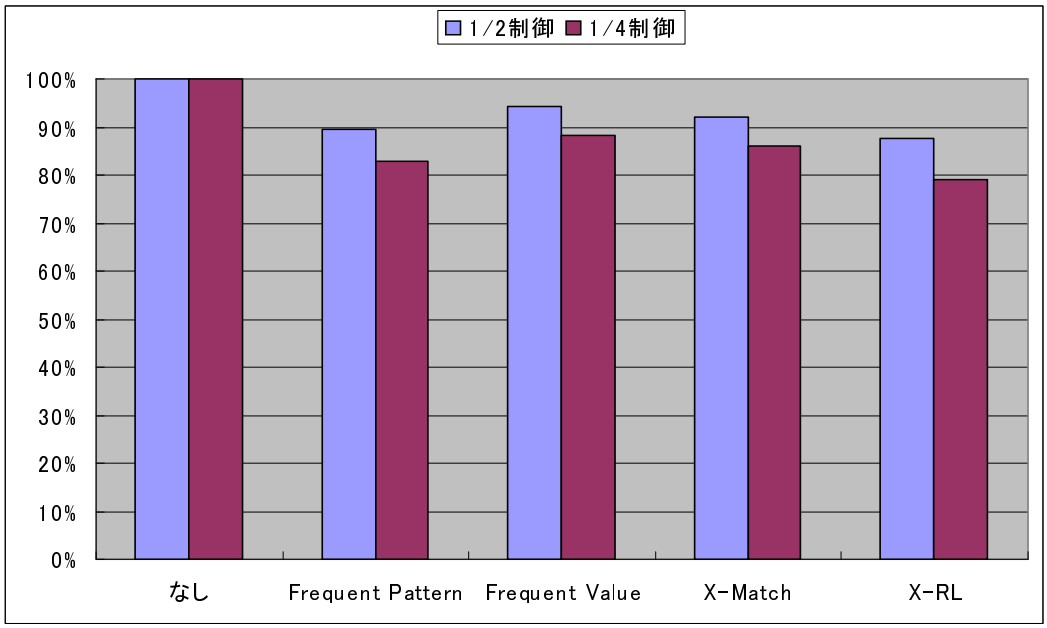


図 4.5: 126.gcc 圧縮アルゴリズム別の消費電力

表 4.6: 126.gcc クロックサイクル

| アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 クロックサイクル | 891566490 | 901145340 | 899248790 | 900356480 | 902570780 |
| 1/2 実行割合 | 100.00% | 101.07% | 100.86% | 100.99% | 101.23% |
| 1/4 クロックサイクル | 891566490 | 908322860 | 905861210 | 911535920 | 917274020 |
| 1/4 実行割合 | 100.00% | 101.88% | 101.60% | 102.24% | 102.88% |

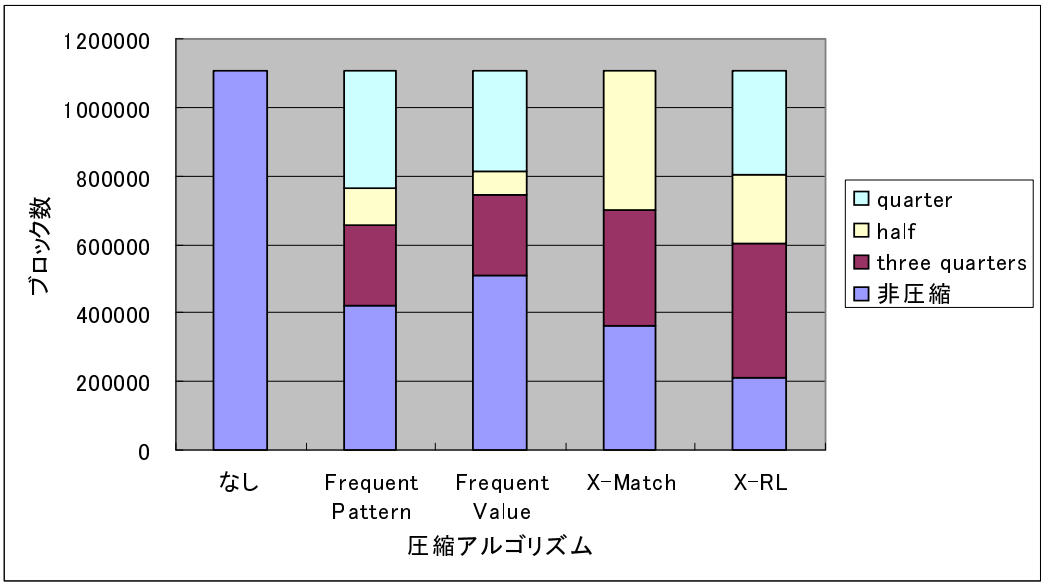


図 4.6: 126.gcc 圧縮サイズ別のブロック数

## 129.compress

129.compress は今回の実験において最も特徴的な結果となったプログラムである．図 4.8 が示すように，全ての圧縮アルゴリズムで約 50%のブロックが 1/2 以下に圧縮が成功している．特に X-Match 以外のアルゴリズムは，そのほとんどが 1/4 以下の圧縮である．しかし図 4.7 が表す通り，全てのアルゴリズムにおいてほぼ電力削減できていない．X-Match・X-RL で約 2%ほどである．129.compress はそのプログラム内容が，ほとんどの値がゼロで構成されている配列データを対象として動くプログラムとなっている．プログラム中で用いられる命令は浮動小数点命令が多く，生成されるデータも浮動小数点データに偏ってる．そのため，主記憶から読み込んだ圧縮対象のデータに対しては圧縮が成功し，プログラム中で生成されたデータに対しては圧縮が失敗するという傾向が現れたと考えられる．表 4.7 より，X-RL で約 0.7%実行サイクル数が増加している．

表 4.7: 129.compress クロックサイクル

| アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 クロックサイクル | 1770709306 | 1776229036 | 1774623936 | 1779563666 | 1779607426 |
| 1/2 実行割合 | 100.00% | 100.31% | 100.22% | 100.50% | 100.50% |
| 1/4 クロックサイクル | 1770709306 | 1777271056 | 1774625826 | 1782952536 | 1782886486 |
| 1/4 実行割合 | 100.00% | 100.37% | 100.22% | 100.69% | 100.69% |

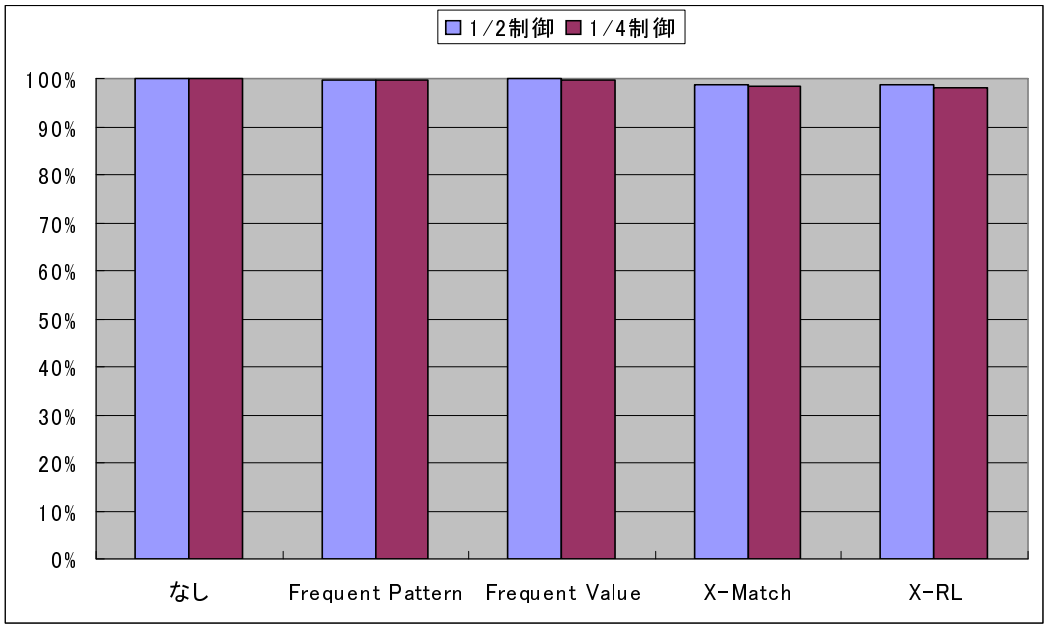


図 4.7: 129.compress 圧縮アルゴリズム別の消費電力

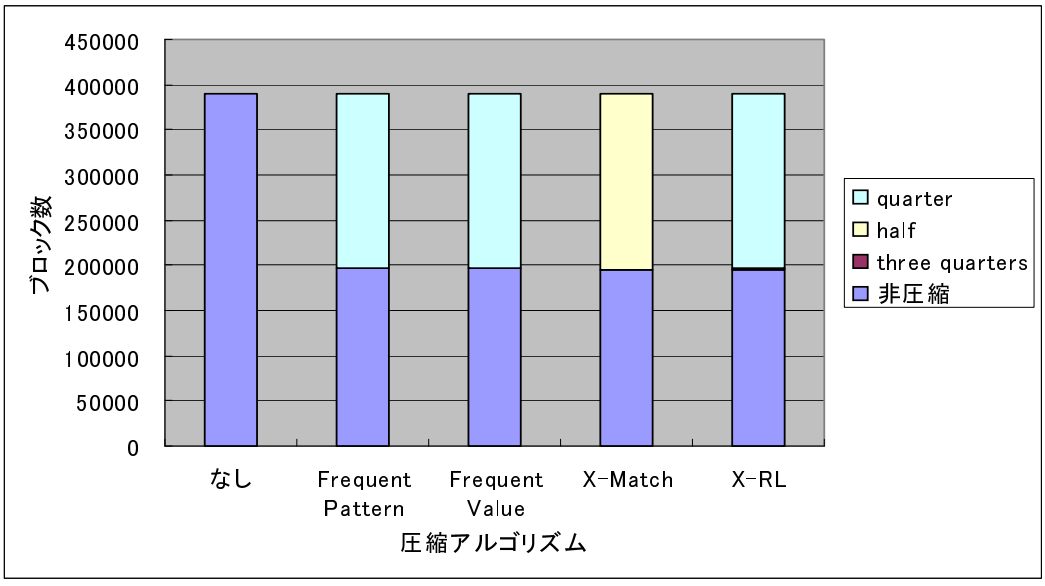


図 4.8: 129.compress 圧縮サイズ別のブロック数

## 130.li

130.li のプログラム実行において消費電力を最も抑えることが出来た圧縮アルゴリズムは図 4.9 で表されているように X-RL アルゴリズムで，続いて X-Match アルゴリズム，Frequent Pattern Compression，Frequent Value Compression となっている．X-RL アルゴリズムにおける電力削減率は約 30%である．また，全てのアルゴリズムで 20%以上の電力削減を達成している．表 4.8 より，X-RL アルゴリズムで約 3.7%実行サイクル数が増加している．

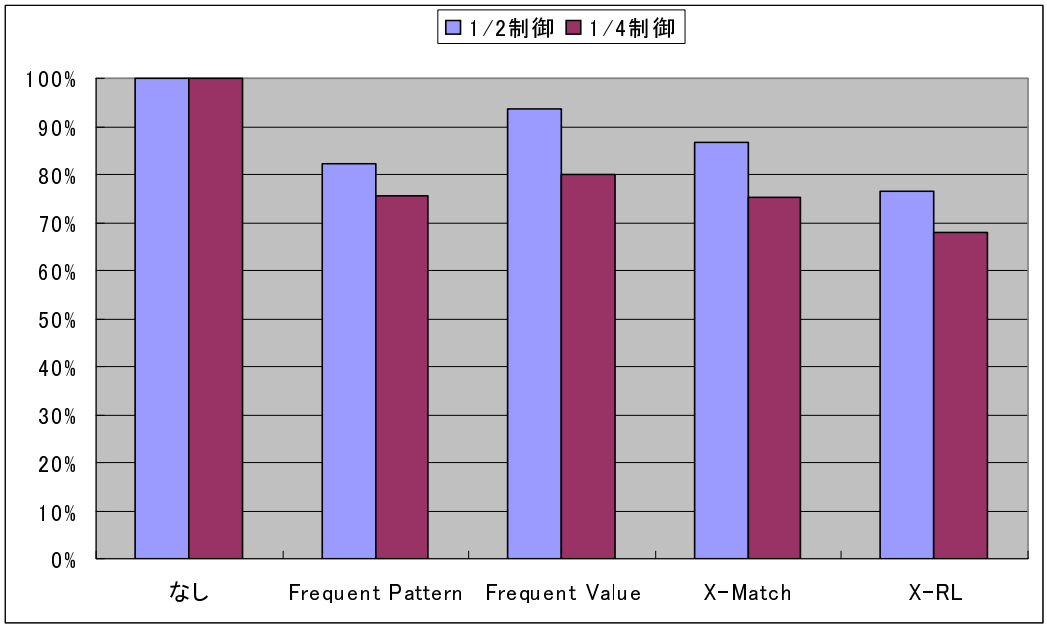


図 4.9: 130.li 圧縮アルゴリズム別の消費電力

表 4.8: 130.li クロックサイクル

| アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 クロックサイクル | 1038658082 | 1046816682 | 1044203322 | 1046222252 | 1052564842 |
| 1/2 実行割合 | 100.00% | 100.79% | 100.53% | 100.73% | 101.34% |
| 1/4 クロックサイクル | 1038658082 | 1061317322 | 1067987712 | 1074940452 | 1077124732 |
| 1/4 実行割合 | 100.00% | 102.18% | 102.82% | 103.49% | 103.70% |

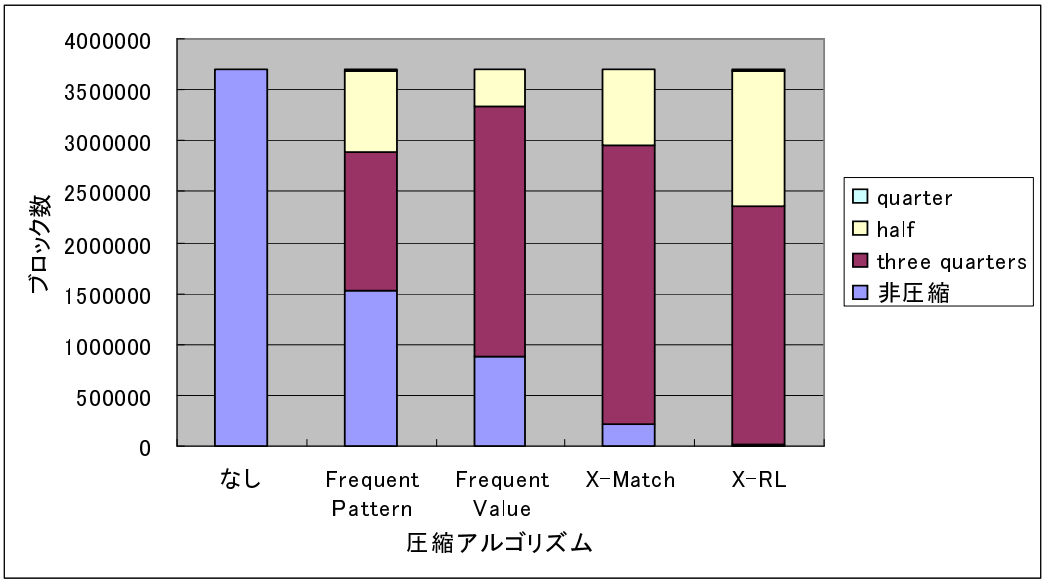


図 4.10: 130.li 圧縮サイズ別のブロック数

## 132.ijpeg

132.ijpeg のプログラム実行において消費電力を最も抑えることが出来た圧縮アルゴリズムは図 4.11 で表されているように X-RL アルゴリズムで，続いて Frequent Pattern Compression，Frequent Value Compression，X-Match アルゴリズムとなっている．最も電力が削減された X-RL アルゴリズムで約 15%の削減となっている．図 4.12 からも分かるように，全てのアルゴリズムで圧縮できたブロック数がほぼ同じとなっている．また，圧縮できたブロック数の割合としては全プログラム中最も少ない．表 4.9 より，最も遅ラムで約 1.2%実行サイクル数が増加している．

表 4.9: 132.ijpeg クロックサイクル

| アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 クロックサイクル | 1125775918 | 1138427228 | 1137749278 | 1138082288 | 1138493848 |
| 1/2 実行割合 | 100.00% | 101.12% | 101.06% | 101.09% | 101.13% |
| 1/4 クロックサイクル | 1125775918 | 1138750538 | 1138158748 | 1138838348 | 1138879048 |
| 1/4 実行割合 | 100.00% | 101.15% | 101.10% | 101.16% | 101.16% |

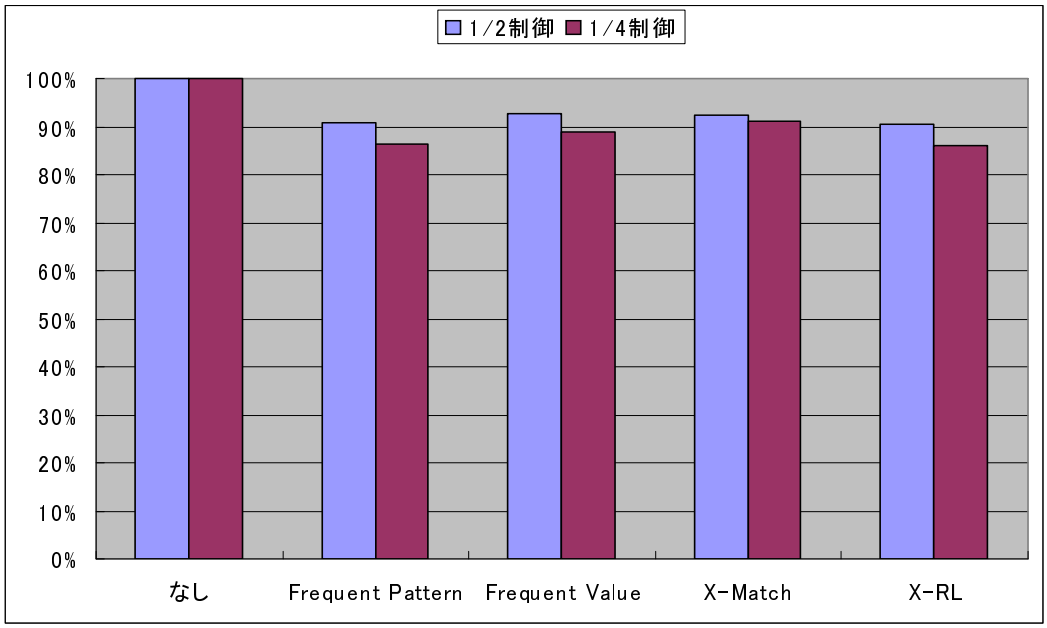


図 4.11: 132.ijpeg 圧縮アルゴリズム別の消費電力

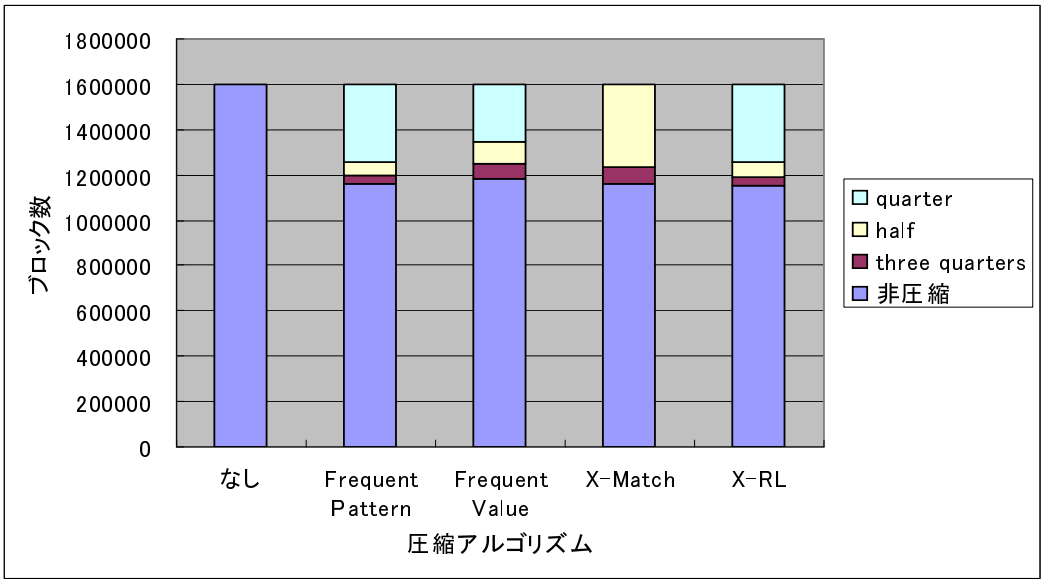


図 4.12: 132.ijpeg 圧縮サイズ別のブロック数

### 134.perl

134.perl のプログラム実行において消費電力を最も抑えることが出来た圧縮アルゴリズムは図 4.13 で表されているように X-RL アルゴリズムで，続いて Frequent Pattern Compression，X-Match アルゴリズム，Frequent Value Compression となっている．最も電力が削減された X-RL アルゴリズムで約 40%の削減となっている．図 4.14 より，全体として圧縮が成功したブロック数が非常に多いが，Frequent Value アルゴリズムはほとんど電力を削減できなかった．図 4.14 の y 軸を見ると L2 キャッシュに格納されたブロック数自体が非常に少ない．下位の記憶領域からの読み込みが少なく，キャッシュが有効に機能しているプログラムであるといえる．表 4.10 では有効桁数の問題で割合としての速度差は出ていない．全てのアルゴリズムで約 0.002%の実行サイクル数増加となっている．

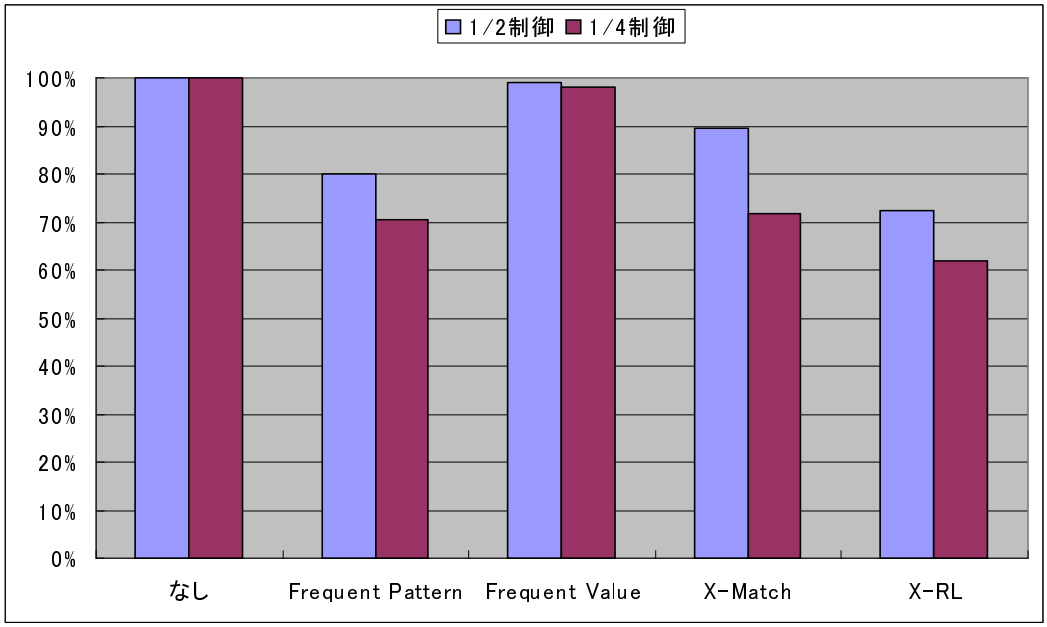


図 4.13: 134.perl 圧縮アルゴリズム別の消費電力

表 4.10: 134.perl クロックサイクル

| アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 クロックサイクル | 1128872405 | 1128896745 | 1128896585 | 1128896625 | 1128896785 |
| 1/2 実行割合 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| 1/4 クロックサイクル | 1128872405 | 1128896965 | 1128896775 | 1128896975 | 1128897065 |
| 1/4 実行割合 | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |

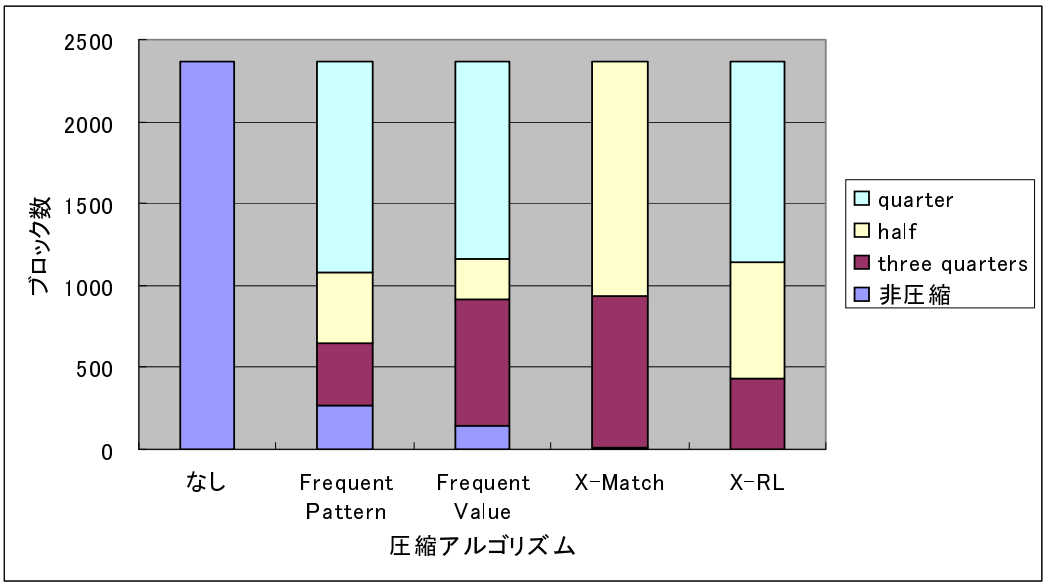


図 4.14: 134.perl 圧縮サイズ別のブロック数

## 147.vortex

147.vortex は全ての圧縮アルゴリズムで大きく電力を削減することができている (図 4.15)．X-RL，Frequent Pattern Compression 共に約 45%の電力を削減している．最も電力削減できなかった Frequent Value Compression でも，約 30%の電力が削減できている．最も遅いアルゴリズムで，約 2.7%実行サイクルが増加している (表 4.11)．

表 4.11: 147.vortex クロックサイクル

| アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 クロックサイクル | 1158357706 | 1181948346 | 1176374226 | 1178720606 | 1183483956 |
| 1/2 実行割合 | 100.00% | 102.04% | 101.56% | 101.76% | 102.17% |
| 1/4 クロックサイクル | 1158357706 | 1185201466 | 1181066846 | 1189155366 | 1189563746 |
| 1/4 実行割合 | 100.00% | 102.32% | 101.96% | 102.66% | 102.69% |

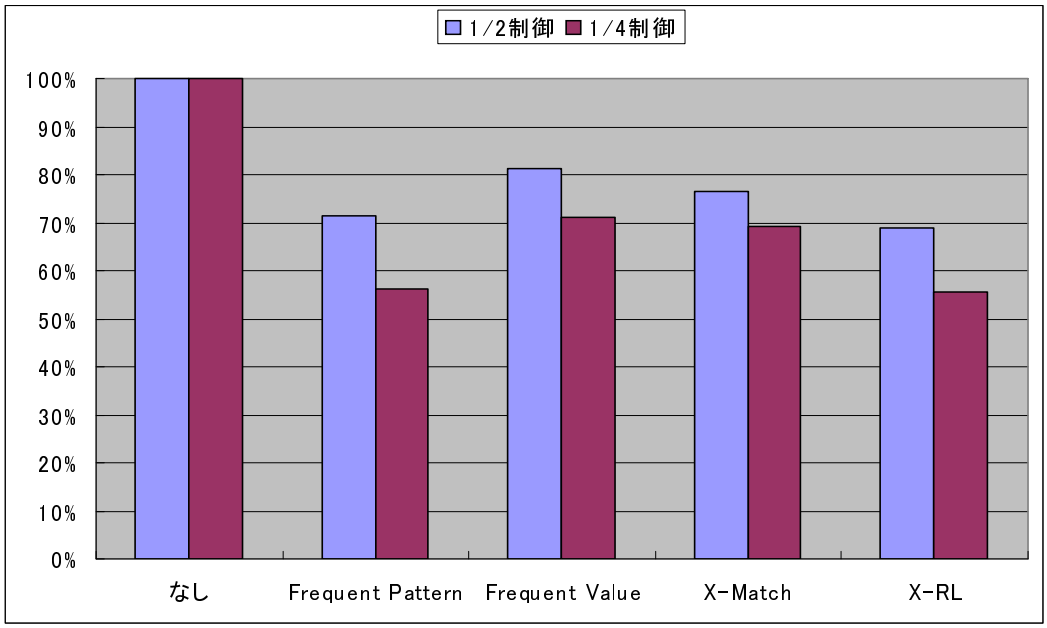


図 4.15: 147.vortex 圧縮アルゴリズム別の消費電力

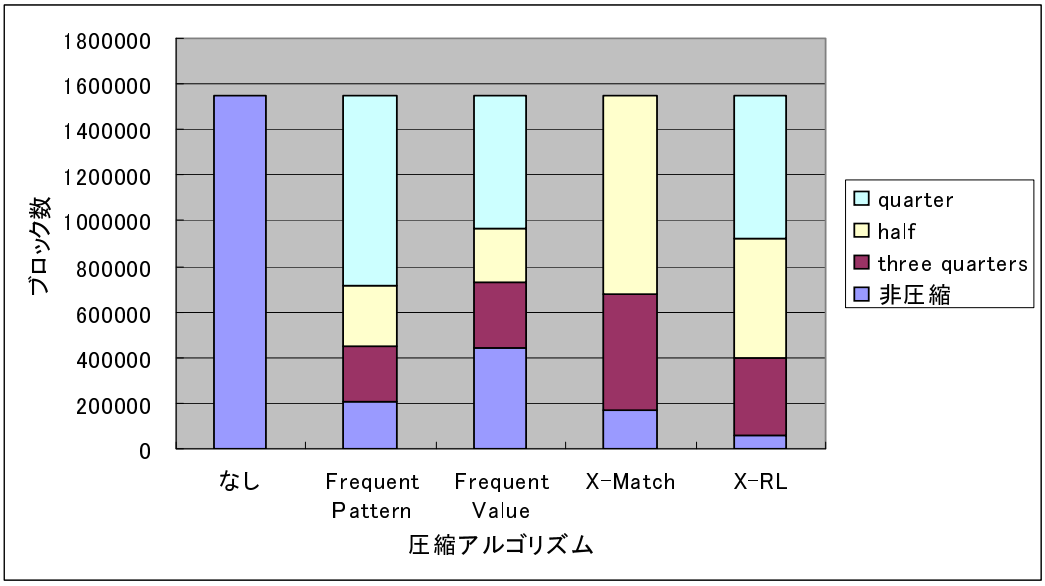


図 4.16: 147.vortex 圧縮サイズ別のブロック数

## 4.4 考察

1/4 制御において，今回の実験で用いた SPECint95 における 8 つのプログラム実行全てで，X-RL アルゴリズムを用いた方式の電力削減率が最も有効という結果となった．X-RL アルゴリズムにおいて最も電力削減率が高かったのは 147.vortex での約 45%削減，最も削減率が少なかったのが 129.compress での約 2%の削減である．

1/2・1/4 制御それぞれの，SPECint95 の 8 つのプログラムに対するアルゴリズムごとの電力使用量の平均を図 4.17 に示す．

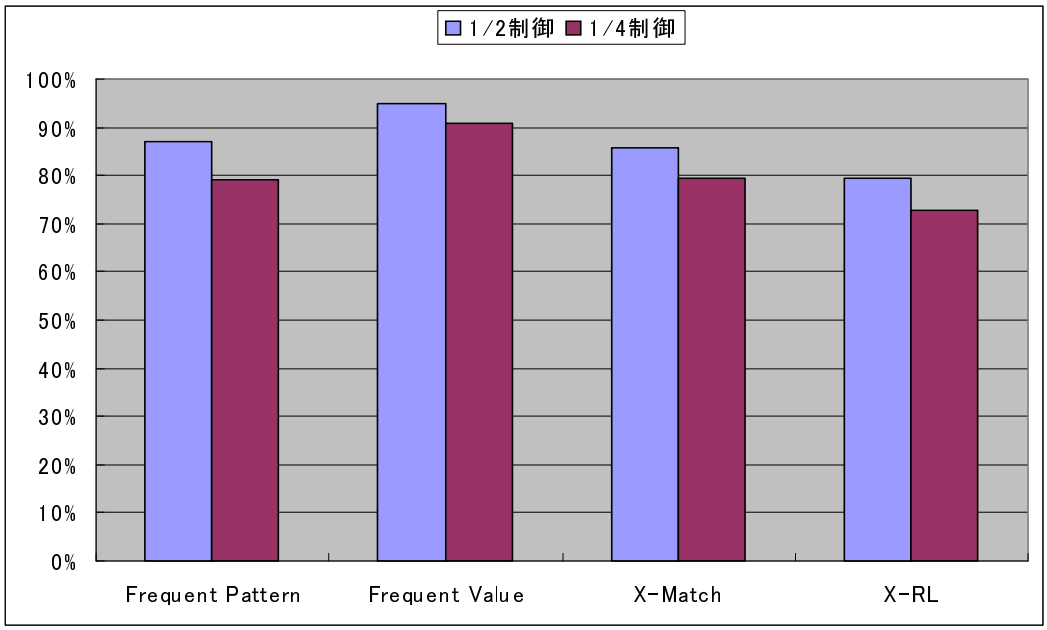


図 4.17: 平均電力量

図 4.17 より，全ての圧縮アルゴリズムにおいて，1/2 制御よりも 1/4 制御が消費電力を削減できた．平均で約 5%，1/4 制御の方が 1/2 制御よりも消費電力が少ない．

1/4 制御において，X-RL アルゴリズムの平均電力削減率は約 30%，二番目に電力削減率の良かった Frequent Pattern Compression との差は平均で約 6%である．最も電力削減量の悪かったのは Frequent Value アルゴリズムで，平均電力削減量は約 10%である．

表 4.12 に 1/2・1/4 制御それぞれの，アルゴリズムごとの平均実行速度を示す．

最も実行クロックサイクル数が増加したのは，1/4 制御の場合の X-RL アルゴリズムで 1.82%である．全体としてもクロックサイクルの増加は 1%〜2%以内に抑えられており，本研究で設定したレイテンシ以内で圧縮・復元を行うことができれば，プログラム実行自体に大きな影響は与えない．

図 4.18 は 1/4 制御の場合の，SPECint95 全体の圧縮サイズの割合である．全圧縮プログラムに渡って約 60%以上，X-RL アルゴリズムでは約 80%ものブロックがいずれかの圧

表 4.12: 平均実効サイクル割合

| 圧縮アルゴリズム | なし | FPC | FVC | X-Match | X-RL |
|---|---|---|---|---|---|
| 1/2 平均実効速度 | 100.00% | 100.91% | 100.68% | 100.88% | 101.09% |
| 1/4 平均実行速度 | 100.00% | 101.41% | 101.20% | 101.71% | 101.82% |

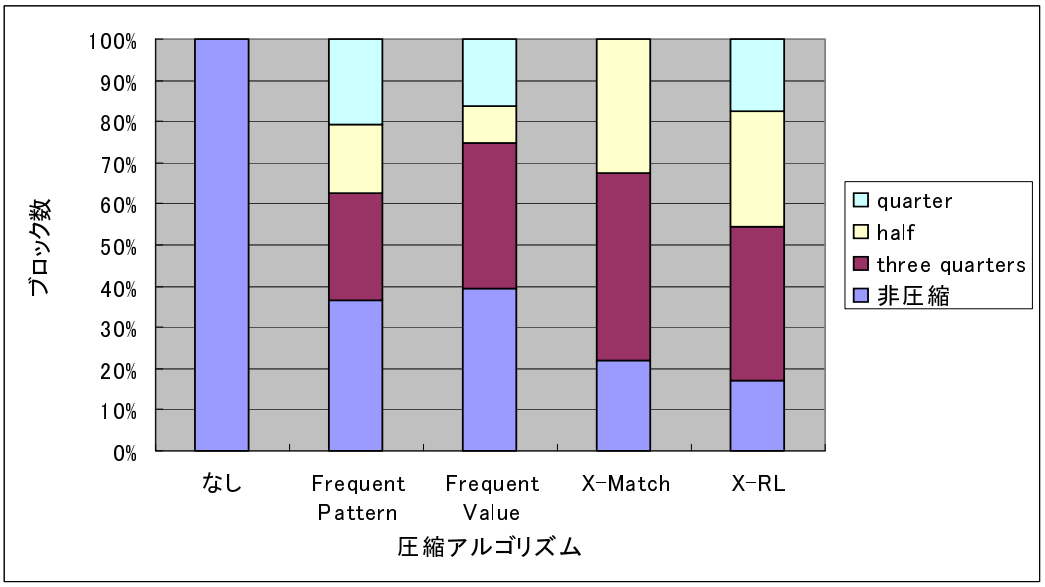


図 4.18: 圧縮サイズ別ブロック数の割合

縮状態となってL2キャッシュに格納されたことが分かる．ここで本研究で用いた1/4単位の電力削減法の対象となる部分について着目する．

全アルゴリズムにおいて，3/4サイズに圧縮することが出来たブロック数は全体の約20%以上存在する．また，X-Matchアルゴリズム以外の圧縮アルゴリズムでは全体の約15%以上のブロックが1/4サイズに圧縮することができており，圧縮ブロック数のみを表すこの結果からは，1/4単位での圧縮は効果的であると言える．X-Matchアルゴリズムに1/4圧縮が存在しないのは，X-Matchアルゴリズムを用いた圧縮で最も圧縮できた場合でもブロックサイズの1/4以下に圧縮することが不可能なためである．

最後に1/4制御における圧縮サイズ別に分けた場合の消費電力内わけを図4.19に示す．

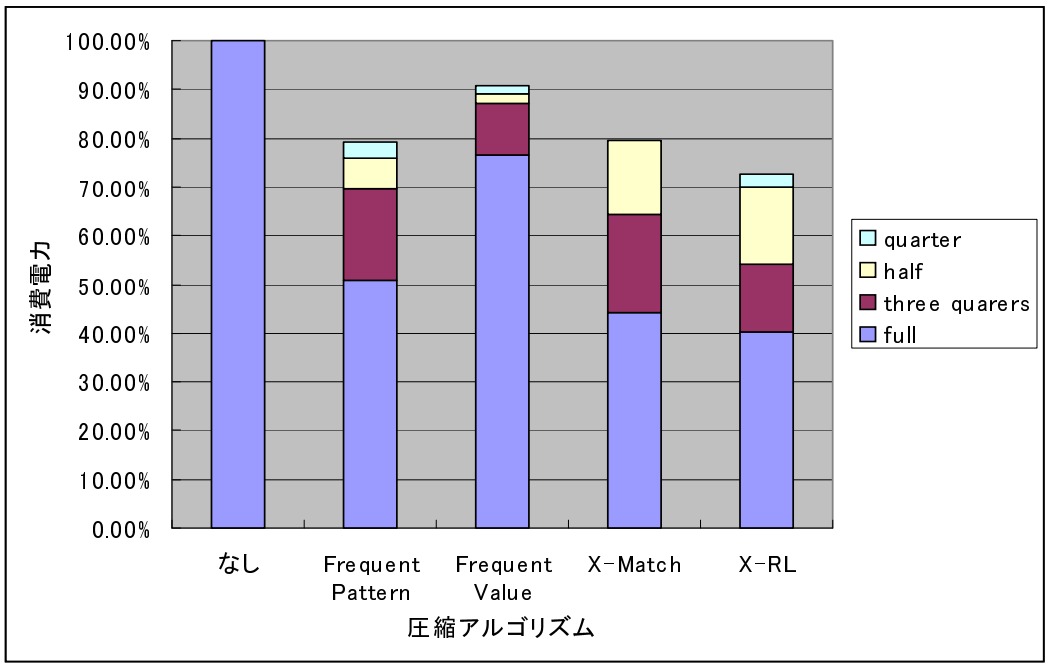


図 4.19: 圧縮サイズ別消費電力の内わけ

X-RL アルゴリズム着目すると，圧縮を用いなかった場合に，元々は 100%であった非圧縮ブロックに対する消費電力が 40%となっている．X-RL アルゴリズム自体の消費電力は非圧縮実行の約 70%となっているので，非圧縮実行の消費電力 60%分を電源制御により 30%，つまり約半分に削減しているということになる．

# 第5章　まとめ

最後に本論文のまとめと，課題を記す.

## 5.1　まとめ

本研究では，データ圧縮を用いたGated-Vddによる電力削減法の性能を向上させるため，キャッシュブロックの圧縮の効率を上げるべく，複数のデータ圧縮アルゴリズムを用いて電力削減率の評価を行った．また，Frequent Pattern Compressionを用いた場合のL2キャッシュに格納されるデータの圧縮サイズの分布に着目し，圧縮データの制御を1/2から1/4とする制御方法を提案した.

実験の結果，評価対象プログラム全てにおいてX-RLアルゴリズムが従来のFrequent Pattern Compressionを用いた場合よりも高い圧縮率をあげることが出来た．このことが電力削減率の向上へと繋がり，X-RLアルゴリズムを用いた場合，非圧縮実行時と比べて平均で約30%電力を削減した．また，1/4単位での圧縮を用いたことで評価対象とした圧縮アルゴリズム全てにおいて，SPECint95のプログラムを用いた測定期間中にL2キャッシュへ格納された総ブロック数の60%以上をいずれかの形で圧縮することができた．そして，1/2制御から1/4制御にすることで，全てのアルゴリズムで約5%の消費電力削減を実現した.

## 5.2　今後の課題

本研究で示した消費電力は，L2キャッシュにおける静的消費電力のみに着目した結果となっており，実際に圧縮に用いるハードウェア，また電源制御に用いるGated-Vdd機構に対する消費電力を考慮していない．実際のハードウェアに本機構を適用するためには，この部分の消費電力を調査することが不可欠となる．それら動的な消費電力とL2キャッシュで削減できる静的な電力とのトレードオフを考慮した上で，圧縮アルゴリズムを選択することが必要となる.

また，今回のシミュレーションでの評価対象はSPECint95という整数演算を主としたプログラムを用いている．129.compressの結果で触れたように，浮動小数点演算を多用するプログラムでは電力削減の傾向が違う可能性が高いため，SPECfpを評価対象として実験を行うことも必要である．浮動小数点データに対する圧縮が現段階のアルゴリズムでは

不十分であると判断された場合，浮動小数点データの圧縮に適した圧縮アルゴリズムの調査も必要となる．

# 謝辞

本研究を行うにあたり，御指導，御鞭撻を頂いた北陸先端科学技術大学院大学情報科学研究科　田中清史　助教授に深く感謝するとともに，ここに御礼申し上げます.

貴重な御意見，御助言を頂きました本学の日比野　靖教授，井口　寧助教授に深く感謝致します.

その他，貴重な御意見を頂きました日比野研究室，田中研究室の皆様をはじめとする多くの方々の御助言に対しまして深く感謝致します.

最後に，これまで温かく見守って下さった両親，兄弟に深く感謝致します.

# 参考文献


[1] 松田愛子. 'データ圧縮を用いたキャッシュメモリの消費電力削減に関する研究', 北陸先端科学技術大学院大学修士論文 2006.

[2] M.Powell, S.Yang, B.Falsafi, K.Roy, T.N.Vijaykumar. 'Gated-Vdd:A Circuit Technique to Reduce Liakage in Deep-Submicron Cache Memories', Proc. of ISLPED, pp. 90–95, 2000.

[3] S.Kaxiras, Z.Hu, M.Martonosi. 'Cache Decay:Exploiting Generational Behavior to Reduce Cache Leakage Power', ISCA, 2001.

[4] A.T.Alameldeen and D.A.Wood. 'Frequent Pattern Compression:A Significance-Based Compression Scheme for L2 Caches', Technical Report 1500, Computer Sciences Dept., UW-Madison, April 2004.

[5] J.Yang, R.Gupta. 'Energy Efficient Frequent Value Data Cache Design', Proc. of MICRO-35, pp. 197–207, 2002.

[6] M.Kjelso, M.Gooch, S.Jones. 'Design and Performance of a Main Memory Hardware Data Compressor', Rroc. of EuroMicro, PP.423–430, 1996.

[7] Standard Performance Evaluation Corporation. 'SPEC CINT95 Benchmarks', http://www.spec.org/cpu95/CINT95/.

[8] SPARC International, Inc. 'The SPARC Architecture Manual Version 9'.