修 士 論 文

キャッシュメモリの消費電力削減に適した圧縮ハードウェアに関する研究

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

川村 俊介

2008 年 3 月

修 士 論 文

キャッシュメモリの消費電力削減に適した圧縮ハードウェアに関する研究

指導教官　田中清史 准教授

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

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

0610028 川村 俊介

提出年月: 2008 年 2 月

概 要

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，電力削減の一つとしてキャッシュメモリの消費電力削減が注目されている．

キャッシュメモリの消費電力を削減する手法の一つとして，Gated-Vdd が挙げられる [1]．Gated-Vdd は，キャッシュメモリに供給する電力をオフにすることで消費電力を削減することができる．しかし電力をオフにするため，そのキャッシュ内のデータは消滅する．これによりキャッシュミスペナルティが増加する．

このキャッシュミスペナルティを抑制しながら消費電力削減を行う方法として，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 [2]．キャッシュブロック内のデータを圧縮することができれば，ブロック内の一部にデータを保持しつつ，部分的に電力をオフにすることができる．

このことに対する取り組みとして，様々な圧縮アルゴリズムによる効果を評価した研究がある [3]．しかし，浮動小数点数データに対しては未検証である．よってこの研究の浮動小数点数データへの有効性や，浮動小数点数データに適したアルゴリズムを調査する必要がある．

本研究では，データ圧縮と電圧制御を用いたキャッシュメモリの消費電力削減方式 [2, 3] を再検討する．特に未検証である浮動小数点数データ向けのデータ圧縮に注目し，最適な圧縮アルゴリズムを調査する．

目 次

第1章　はじめに

1.1　背景

1.1.1　消費電力の問題

近年，モバイルコンピュータや携帯電話などのデバイスにおいてプロセッサの高性能化と同時に低消費電力を満たす要求が高まっている．モバイル機器は，持ち運ばれることを目的とするためにバッテリーや実装が小型化されている．消費電力の増加はバッテリーの持ち時間の減少だけでなく，プロセッサ自体の発熱という問題が懸念される．発熱を抑制するために冷却装置を実装したとしても，モバイル機器という使用特性から大きな装置の実装は困難である．その一方で，今後携帯電話など組込み機器においてより性能を必要とするアプリケーションが実行されることが予想される．

またモバイル機器以外のハイパフォーマンスコンピュータでも，文献[4]では，コンピュータの消費電力に対するパフォーマンスが今日のレベルから改善しなければ，マシンの運用に必要とされる電気代がハードウェア自体のコストを大幅に上回る可能性があると指摘されている．コンピュータ機器の消費電力を抑えられなくなれば，地球環境全体への影響はもちろん，計算処理全体に関してコスト面で深刻な問題が生じる可能性も考えられる．

よって，コンピュータの中でのエネルギー消費を大きく占めるプロセッサ部は性能向上をしつつ，低消費電力化が必要となっている．

1.1.2　プロセッサの消費電力

プロセッサの性能向上はトランジスタのプロセスルールの微細化と動作周波数を上げることによって達成されてきた．しかしプロセスルールのさらなる微細化によって，トランジスタの内部で漏れ出している動作とは関係のないリーク電流が無視できなくなっている．

一般に，消費電力は大きく2つに分類できる．1つは，動的消費電力で，トランジスタのスイッチングによるものである．もう1つは，静的消費電力で，トランジスタのリーク電流によるものである．近年のプロセッサは消費電力の中で静的消費電力が占める割合が高くなっている．

一方，高性能プロセッサにおけるキャッシュメモリは，チップ面積の大部分を占めるほど増加傾向にある．そしてこの増加傾向のあるキャッシュメモリに付随したリーク電流の

増大が問題となっている．

1.2 目的

本研究では，プロセッサの消費電力削減を目的とする．プロセッサにおける消費電力問題を解決するために様々な低消費電力化の研究が行われているが，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．

具体例としては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 [2]．

文献 [3] においては 4 つの圧縮アルゴリズムについて調査している．その中で SPECint95 に対するシミュレーション結果では X-RL アルゴリズム [5] が効率がよく，非圧縮時と比較して平均 27 %電力を削減している

しかし，浮動小数点数データに対しては未検証である．よって浮動小数点数データへの有効性や，浮動小数点数データに適したアルゴリズムを調査する必要がある．

本研究では，データ圧縮と電圧制御を用いたキャッシュメモリの消費電力削減方式 [2, 3] を再検討する．特に未検証である浮動小数点数データ向けのデータ圧縮に注目し，最適な圧縮アルゴリズムを調査する．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[1]

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．

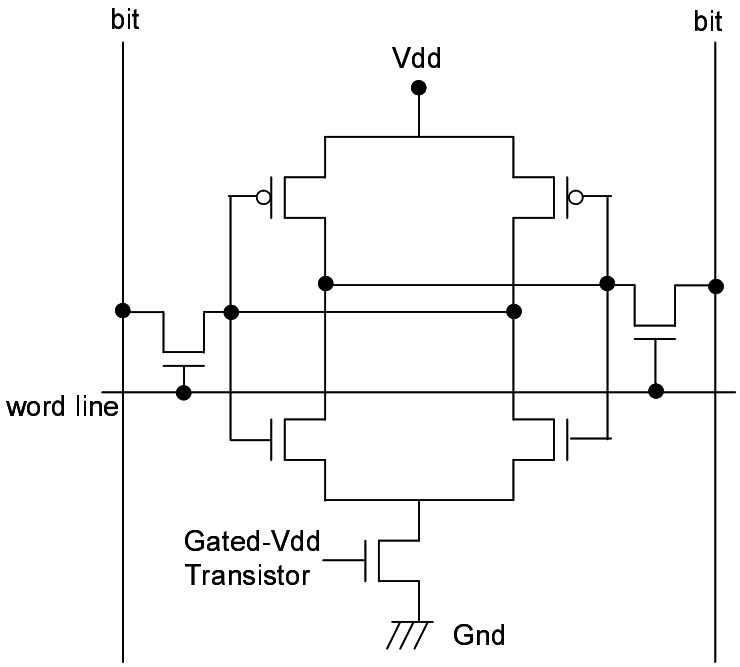

図 2.1: Gated-Vdd

2.2 Cache Decay[6]

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 がオフになった場合，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．

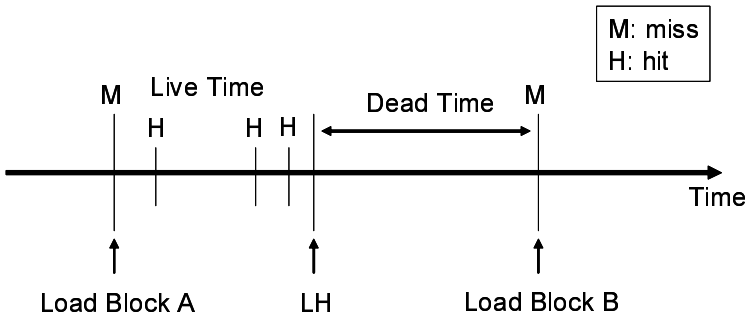

図 2.2: キャッシュ参照の流れ

2.3 データ圧縮を用いた Gated-Vdd による電力削減法 [2]

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[7]というデータ圧縮アルゴリズムが用いられている．

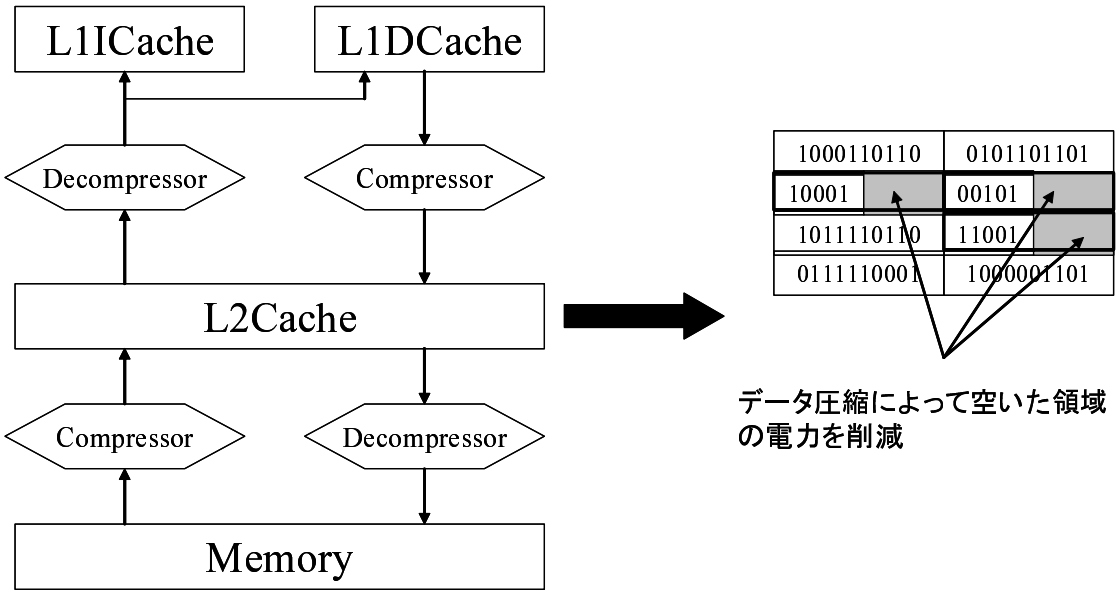

図 2.3: L2 キャッシュに対するデータ圧縮

2.4 整数データにおけるキャッシュブロック圧縮アルゴリズム [3]

データ圧縮を用いた Gated-Vdd による電力削減法 [2] では圧縮アルゴリズムは Frequent Pattern Compression[7] というアルゴリズムのみで検証されていた．文献 [3] では電力削減法の圧縮アルゴリズムの点に着目している．

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[7] 以外にも，Frequent Value Compression[8]，X-Match アルゴリズム [5]，X-RL アルゴリズム [5] を紹介し，これらのアルゴリズムを用いた評価を行っている．

その結果 X-RL アルゴリズムの性能が最も良く，非圧縮実行と比べて平均で約 27%の電力

を削減している (図 2.4)．評価対象のプログラムとしてはSPECint95[9] ベンチマークを用いている．

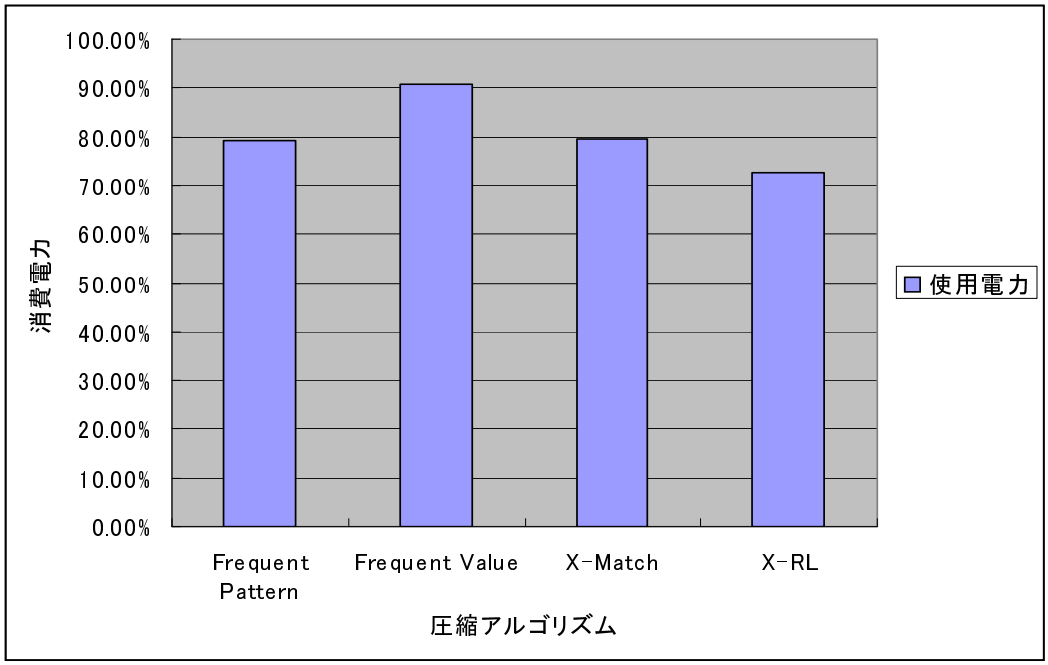

図 2.4: 平均電力量

2.5 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．

2.5.1 Frequent Pattern Compression[7]

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 2.1 に 8 つの圧縮規則を示す．

表 2.1: 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 2.5 に示す.

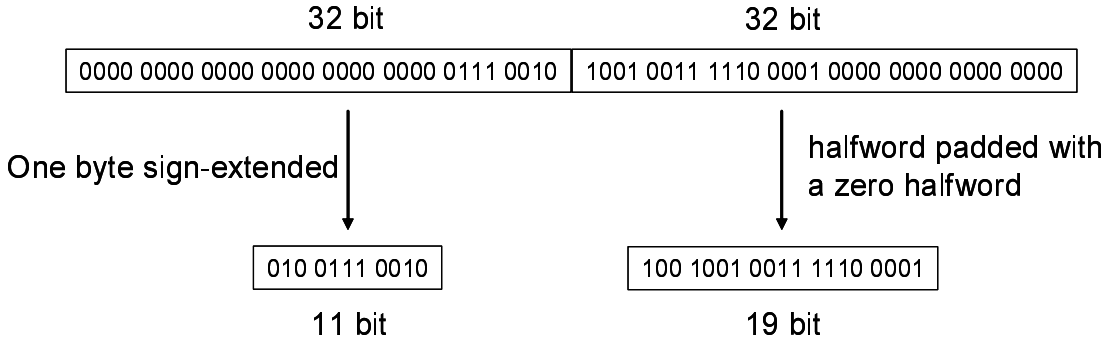

図 2.5: Frequent Pattern Compression 圧縮例

図 2.5 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.

2.5.2 Frequent Value Compression[8]

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 [8] 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 2.6 に示す．

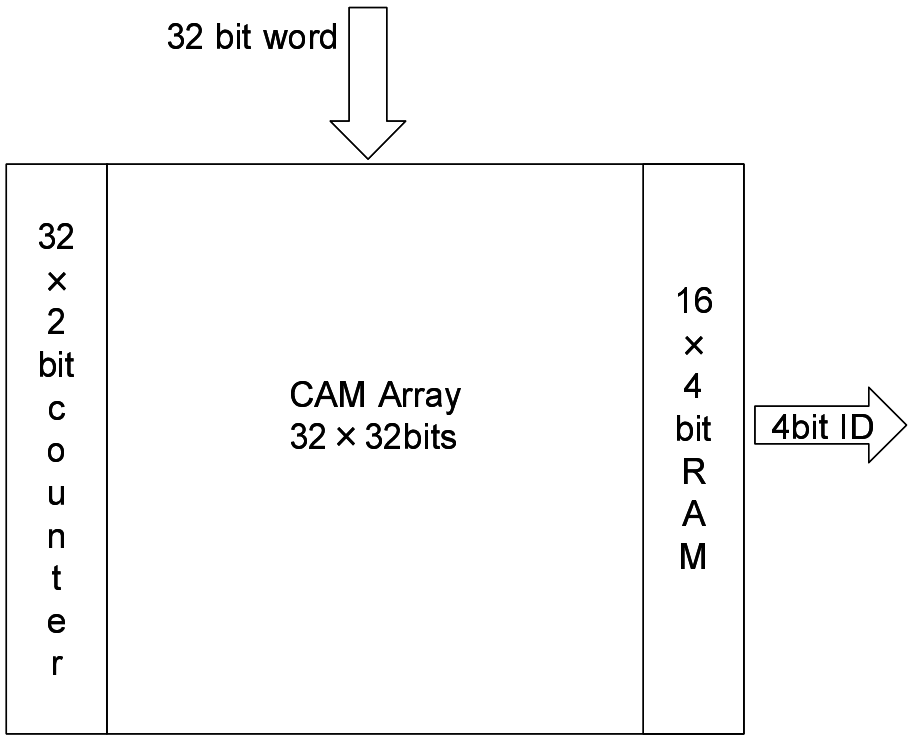

図 2.6: Finder と Encoder の複合機構

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 2.6 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 2.6 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 2.6 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 2.6 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 2.7 に FV の圧縮例を示す．

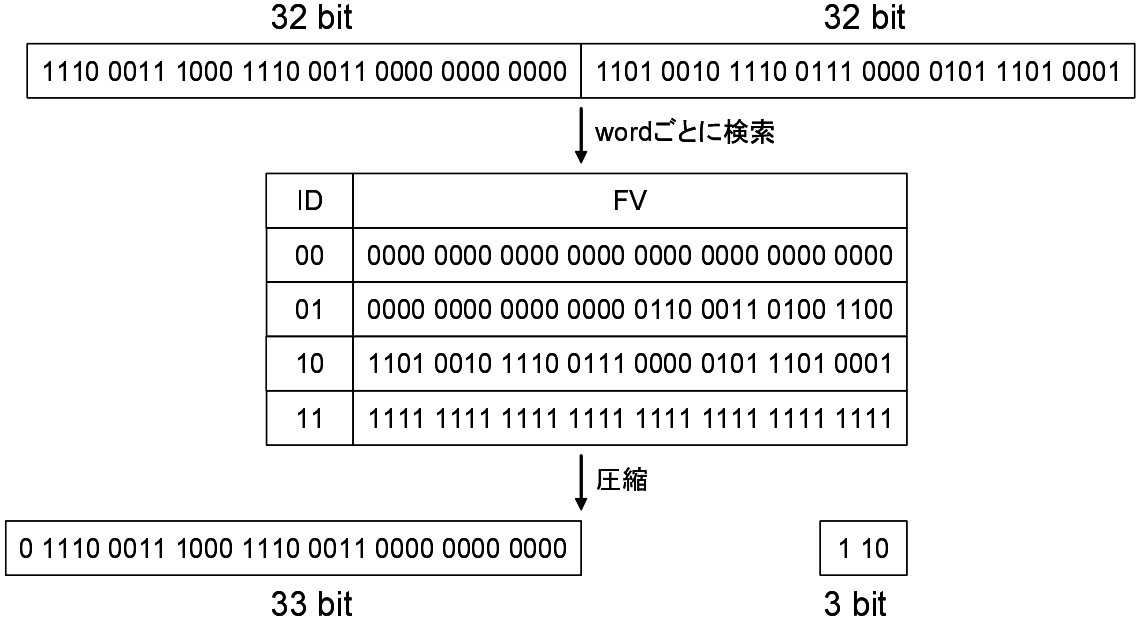

図 2.7: Frequent Value Compression 圧縮例

図 2.7 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．

2.5.3 X-Match[5]

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．

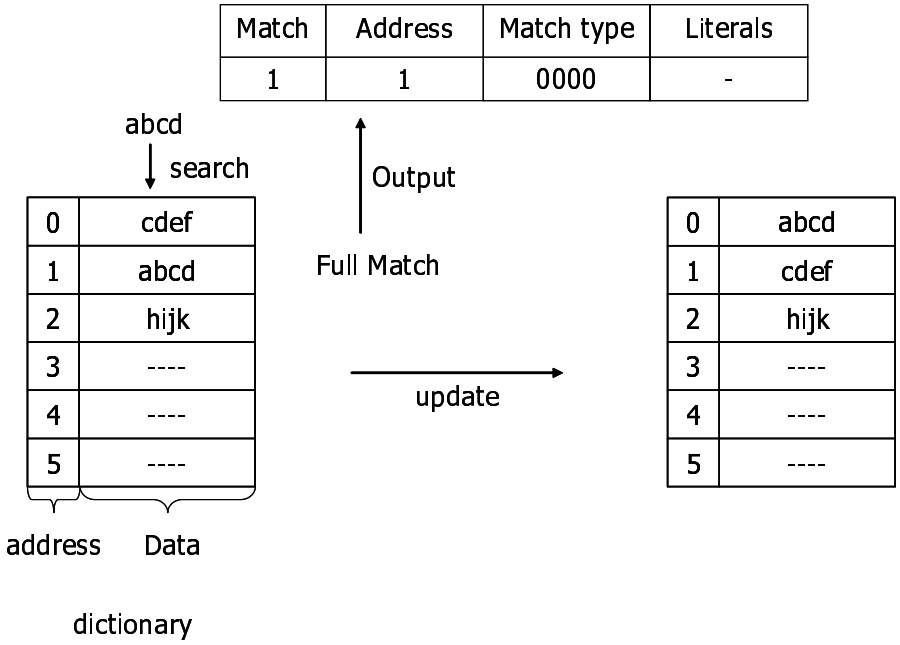

図 2.8: X-Match：full match 圧縮例

full match

図 2.8 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．

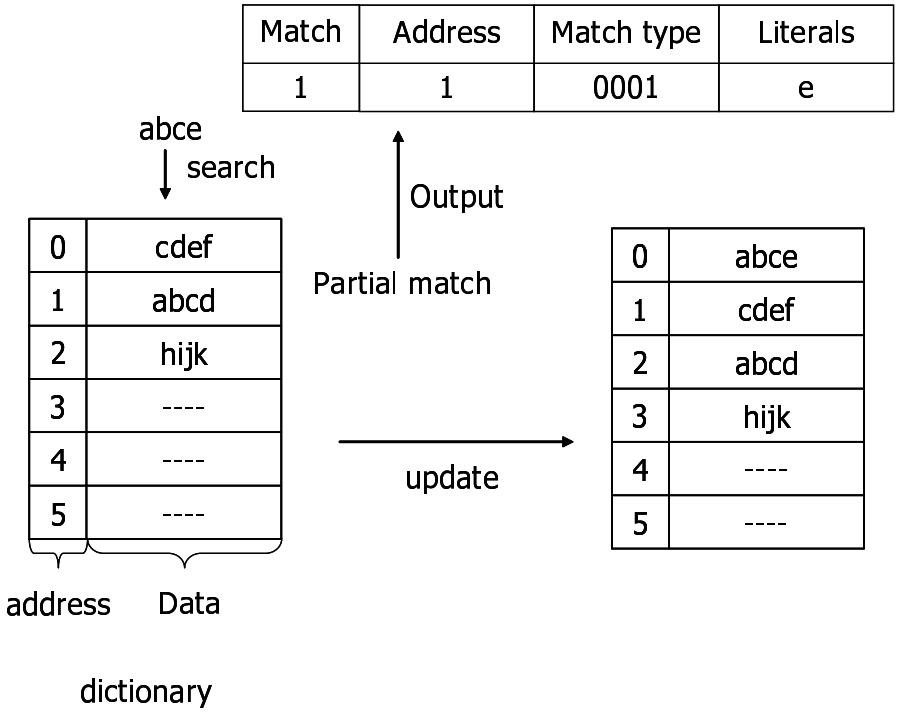

図 2.9: X-Match：partial match 圧縮例

partial match

図 2.9 は 3byte partial match の例である．図 2.8 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，ρ はそのときの有効エントリ数となる．

表 2.2 は，0 から 8 を表す phasing in binary codes 例である

表 2.2: phasing in binary codes 例

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

2.5.4 X-RL[5]

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 2.10 にその例を示す．

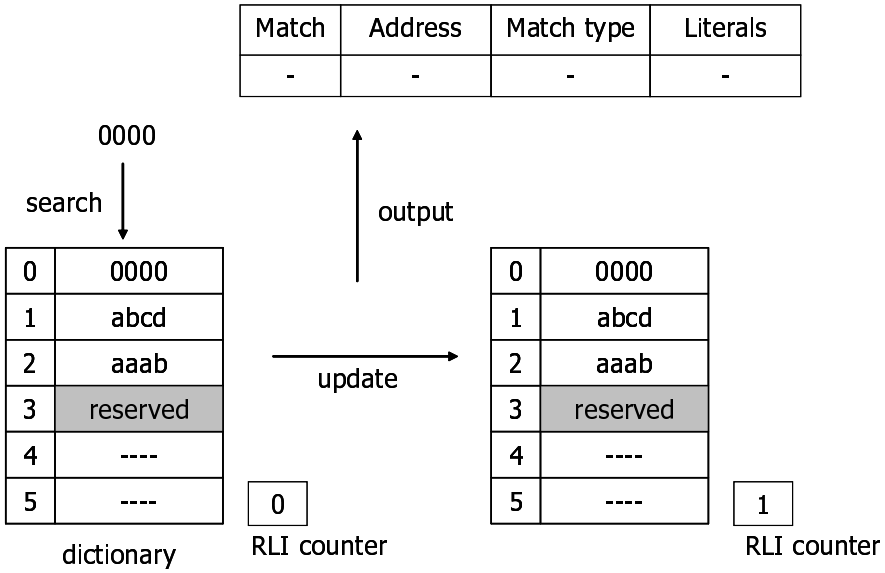

図 2.10: RLI counter 使用例

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．

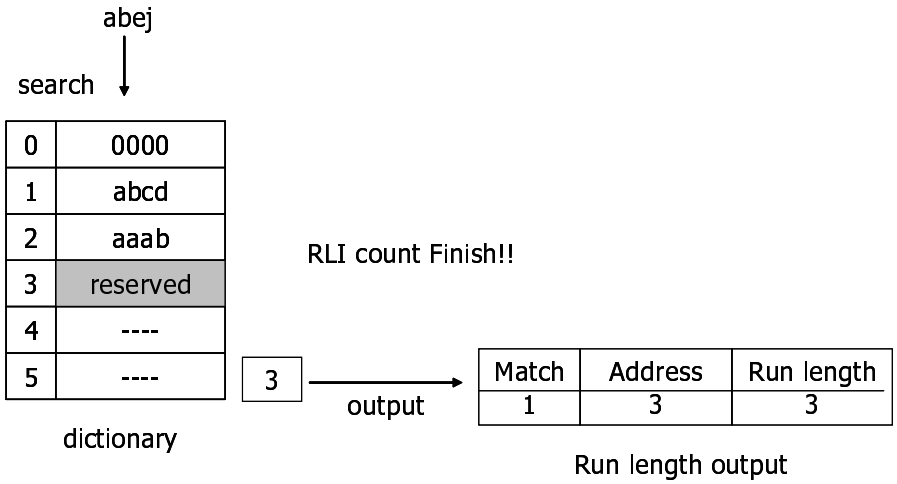

図 2.11: RLI counter 使用時の出力例

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 2.11 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．

第3章　浮動小数点数データ向けのデータ圧縮

本章では，浮動小数点数向けデータを圧縮するに際して新たに導入するアルゴリズムについて述べる．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　浮動小数点数データ向けのデータ圧縮アルゴリズム

本研究では浮動小数点数データ向けのデータ圧縮において FPC(Floating-Point Compression) アルゴリズム [10] に注目した．

FPC アルゴリズムは科学技術計算など高い性能が要求される計算向けに設計されていて，64bit の浮動小数点数データに対して可逆圧縮するアルゴリズムである．

FPC アルゴリズムは次のデータ値を予測する 2 つのコンテキスト予測機構 (FCM[11], DFCM[12]) を使います．予測値と実際のデータ値は排他的論理和を行い，そしてその結果の連続する 0 部分を圧縮します．図 3.1 にその例を示す．

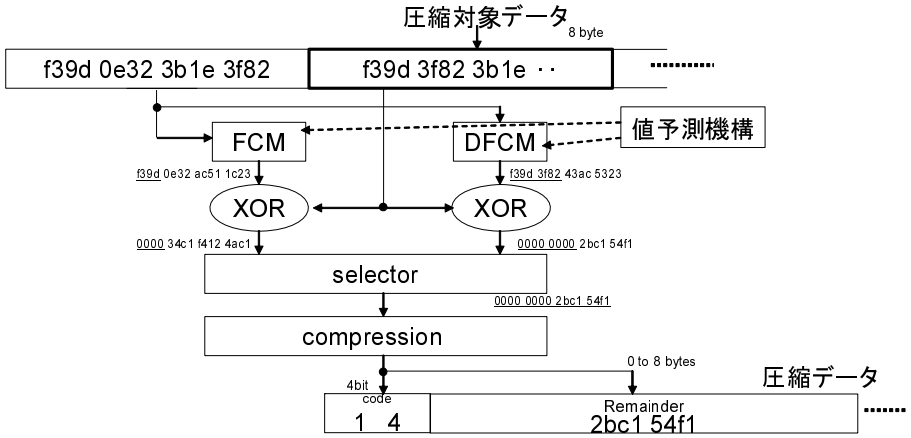

図 3.1: FPC 圧縮アルゴリズム

FCM は過去の連続した有限個の値の結果をハッシュ・テーブルを用いて保存しておき，データ値予測を行う機構である．これに対し DFCM は値自身の代わりに差分をハッシュ・テーブルに保存しておき，データ値予測を行う機構である．2 つのコンテキスト予測機構で予測した値のうち，圧縮対象データにより近いものと圧縮対象データとの排他的論理和を計算し，結果の下位側の連続する 0 部分を省略することにより圧縮データを得る．FCM，DFCM のどちらを使用したか，何ビット省略したかを記したコードを圧縮後のデータの先頭に付ける．

図 3.1 の値を用いて説明する．圧縮対象データの 1 つ前の対象データは値予測機構である FCM，DFCM に入る．この値予測機構で対象データ以前のデータ値群を元に圧縮対象データ値を予測した値を出力する．

- FCM による予測値 … f39d 0e32 ac51 1c23
- DFCM による予測値 … f39d 3f82 43ac 5323

この 2 つの値と圧縮対象データを排他的論理和を行う．

- FCM による予測値との排他的論理和 … 0000 34c1 f412 4ac1
- DFCM による予測値との排他的論理和 … 0000 0000 2bc1 54f1

DFCM による予測値との排他的論理和を行った値の方が先頭からの 0 部分が多い．そしてセレクタでより 0 部分が多い値を選択する．その後，圧縮機構で 0 部分の圧縮を行う．

- 圧縮前　0000 0000 2bc1 54f1
- 圧縮後　2bc1 54f1

この例では 4byte 分の値が圧縮された．この時，圧縮後のデータ値に 4bit 圧縮コードを付加する．圧縮コードには FCM，DFCM どちらによる予測値を用いたかと，何 byte 分 0 部分が圧縮されたかの情報が入っている．

復元はまず最初の 4bit コードを読み込む．その中の 3bit により何ビット分圧縮されたかがわかる．残りの 1bit により FCM か DFCM のどちらで排他的論理和を行ったのかが選択できる．以上の工程で FPC アルゴリズムにおいて圧縮が行われる．

FCM と DFCM はそれぞれハッシュ・テーブルである．本研究ではこのテーブルサイズを 3 つ用意して検証した (表 3.1)．テーブルサイズが大きい程，たくさんの過去の bit パターンを取り込むことができる．つまり FCM，DFCM のテーブルサイズは FCM，DFCM の値予測の精度のパラメータである．しかしテーブルサイズは処理速度とトレードオフの関係にある．

表 3.1: FCM,DFCM のテーブルサイズ

	FCM,DFCM のテーブルサイズ
Lv0	8B
Lv3	64B
Lv5	256B

3.2 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 3.2 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．

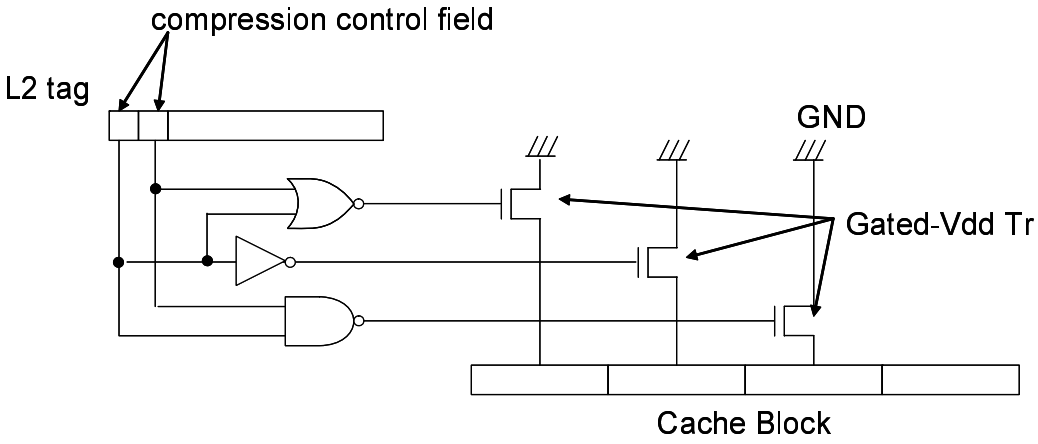

図 3.2: Gated-Vdd による電源制御

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 SPECfp95 ベンチマーク [13] を用いる．表 4.1 に評価に用いた SPECfp95 のプログラムをまとめる．

表 4.1: SPECfp95 Benchmarks

Benchmark	Detail
101.tomcatv	Vectorized mesh generation
102.swim	Shallow water equations.
103.su2cor	Monte-Carlo method.
104.hydro2d	Navier Stokes equations.
107.mgrid	3d potential field.
110.applu	Partial differential equations.
125.turb3d	Turbulence modeling.
141.apsi	Weather prediction.
145.fpppp	From Gaussian series of quantum chemistry benchmarks.
146.wave5	Maxwell's equations.

プログラム実行の開始後 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．

4.2　評価環境

評価に用いる CPU シミュレータは SPARC V9[14] 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，FORTRAN 言語で記述された SPECfp95 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を 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後 1 億命令 ・FV 数は 16 個
X-Match	10cycle	10cycle	なし
X-RL	10cycle	10cycle	・RLI counter は 3bit
FPC	10cycle	10cycle	なし

4.3 実験結果

実験で得られたデータを SPECfp95 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
  1/4 制御において，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．

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．

101.tomcatv

図 4.1 に表されているように 101.tomcatv のプログラム実行において非圧縮時を 100%とした消費電力量は全ての圧縮アルゴリズム方式で超えていた．消費電力の増加分が最も少ないアルゴリズムは X-RL アルゴリズムで 0.4%の増加であった．また図 4.2 から分かるように圧縮できたブロックは X-RL アルゴリズムで全ブロック数の約 9%であった．

結果的に全てのアルゴリズムにおいて圧縮をかけることによって，消費電力を増やす結果となった．これは実行速度の増加分が消費電力の増加につながったと考えられる．

最も実行速度が遅かったのも，表 4.4 より X-RL で約 3.6%増加している．

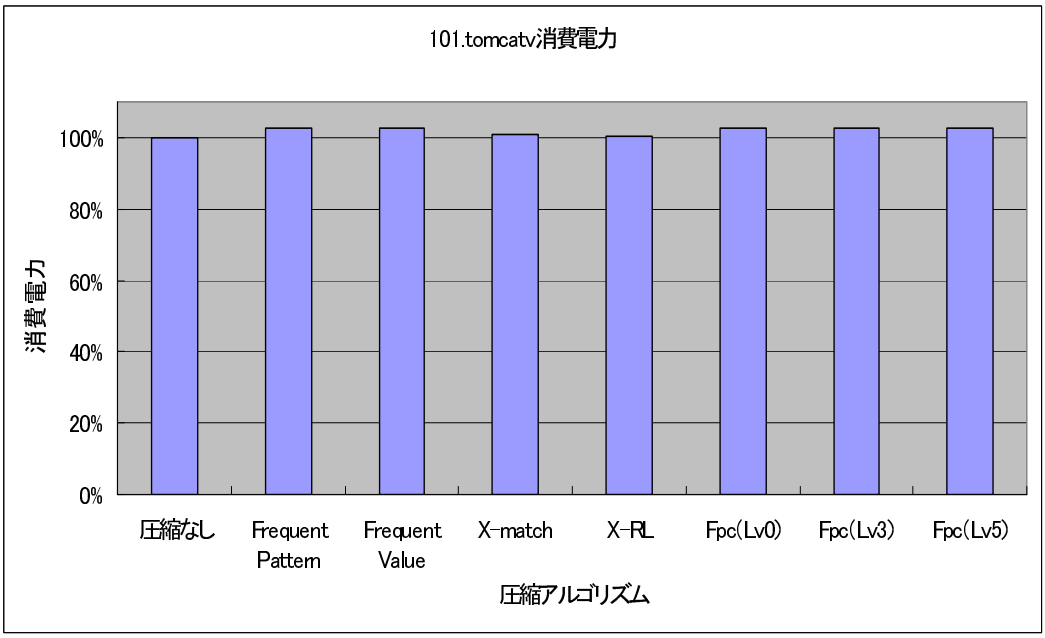

図 4.1: 101.tomcatv 圧縮アルゴリズム別消費電力

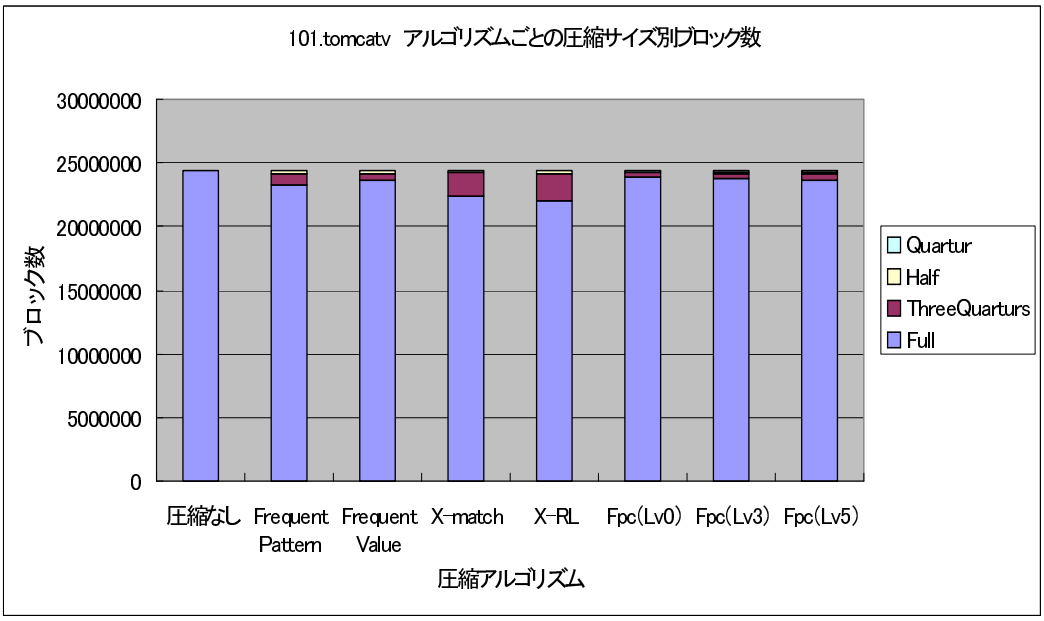

図 4.2: 101.tomcatv 圧縮サイズ別ブロック数

表 4.4: 101.tomcatv クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	4830055058	4996039578	4993500498	5002556698
実行速度	100.00%	103.44%	103.38%	103.57%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	5004584748	4991281198	4992629488	4993220888
実行速度	103.61%	103.34%	103.37%	103.38%

102.swim

図 4.3 に表されているように 102.swim のプログラム実行において消費電力も全てのアルゴリズムで 100%を超えていた．消費電力の増加分が最も少ないアルゴリズムは X-Match アルゴリズムで約 3%の増加であった．

また図 4.4 から分かるように圧縮できたブロックは X-Match アルゴリズムで約 8%であった．結果的に全てのアルゴリズムにおいて圧縮をかけることによって，消費電力を増やす結果となった．これは実行速度の増加分が消費電力の増加につながったと考えられる．

最も実行速度が遅かったのも，表 4.5 より X-Match で約 4.6%増加している．

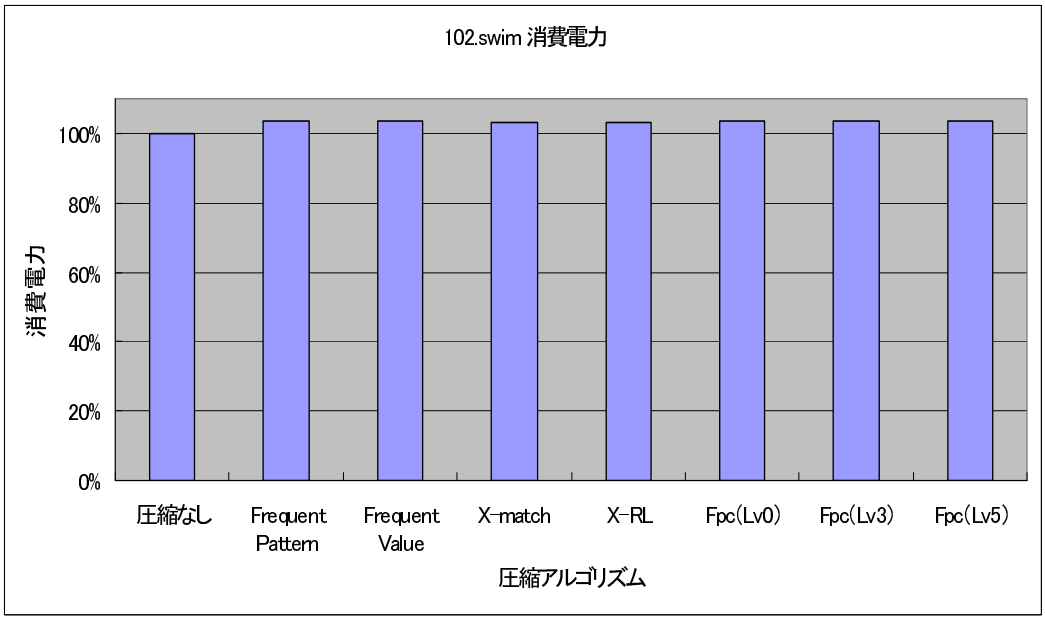

図 4.3: 102.swim 圧縮アルゴリズム別消費電力

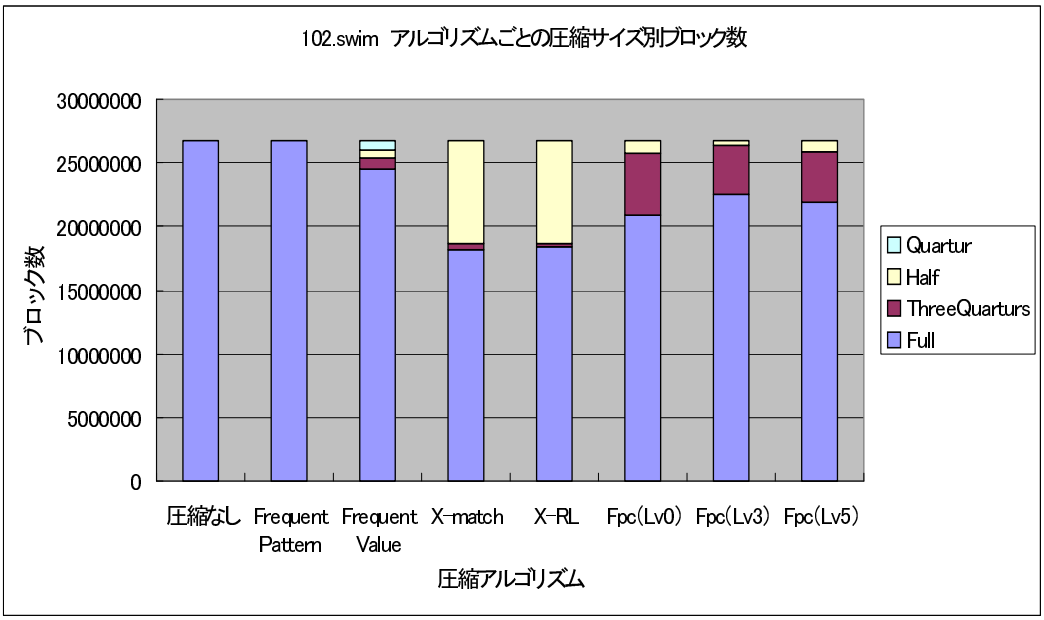

図 4.4: 102.swim 圧縮サイズ別ブロック数

表 4.5: 102.swim クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	5554385459	5747216359	5762487239	5809252699
実行速度	100.00%	103.47%	103.75%	104.59%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	5807782649	5789756289	5777746289	5782892259
実行速度	104.56%	104.24%	104.02%	104.11%

103.su2cor

図 4.5 に表されているように 103.su2cor のプログラム実行において消費電力は全てのアルゴリズムで 100%を超えていた．消費電力の増加分が最も少ないアルゴリズムは X-RL アルゴリズムで約 0.5%の増加であった．

また図 4.6 から分かるように圧縮できたブロックは X-RL アルゴリズムで約 5%であった．結果的に全てのアルゴリズムにおいて圧縮をかけることによって，消費電力を増やす結果となった．これは実行速度の増加分が消費電力の増加につながったと考えられる．

最も実行速度が遅かったのも，表 4.6 より X-RL で約 3.8%増加している．

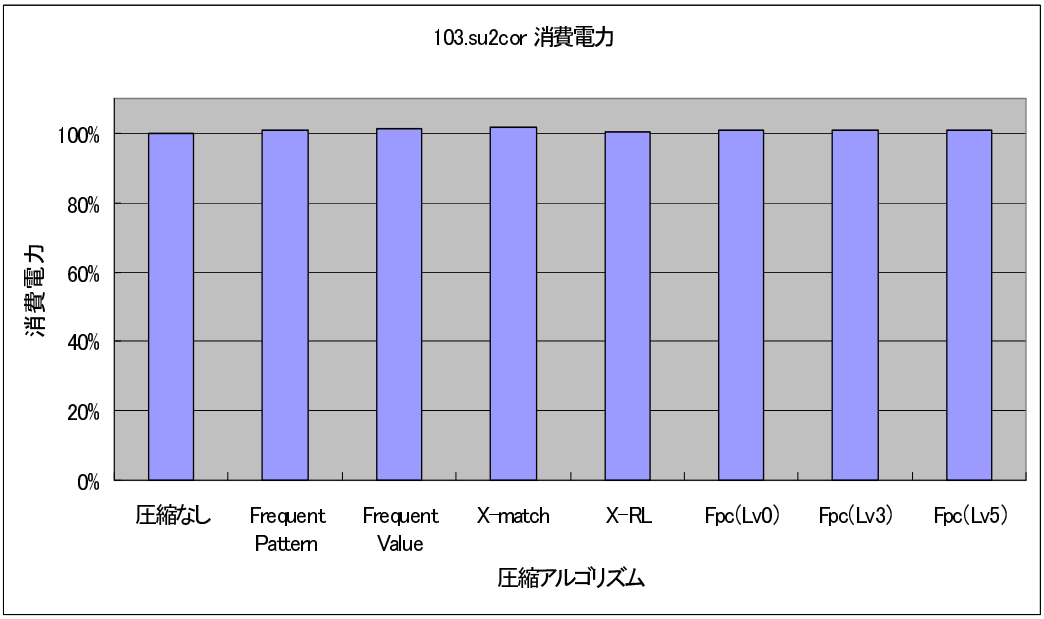

図 4.5: 103.su2cor 圧縮アルゴリズム別消費電力

表 4.6: 103.su2cor クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	4553517622	4723429572	4719578092	4723369612
実行速度	100.00%	103.73%	103.65%	103.73%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	4724164822	4723022402	4723170082	4723210802
実行速度	103.75%	103.72%	103.73%	103.73%

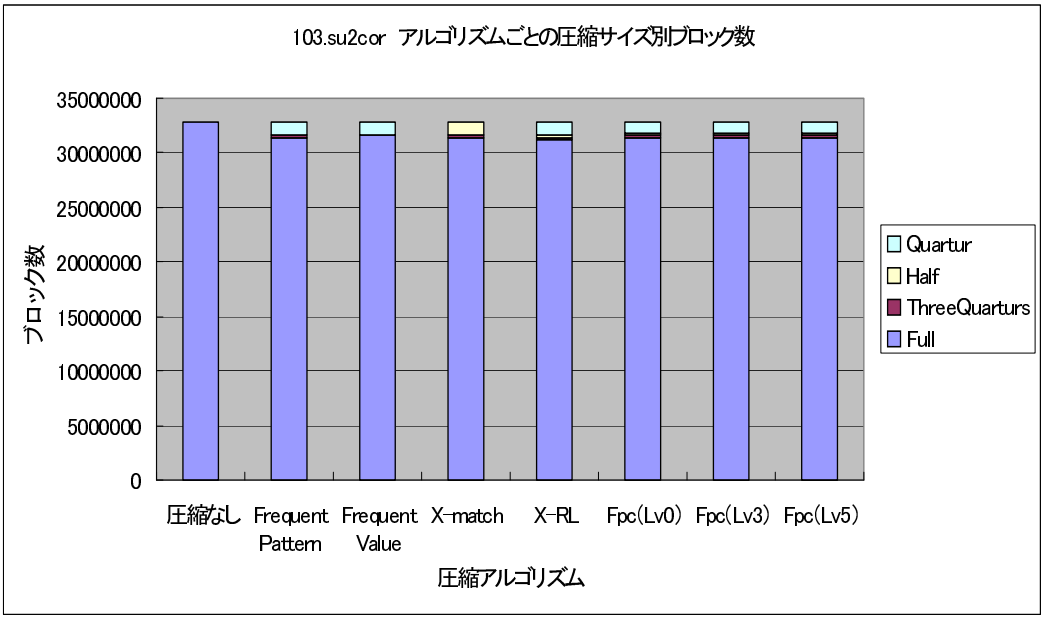

図 4.6: 102.swim 圧縮サイズ別ブロック数

104.hydro2d

104.hydro2d のプログラム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 4.7 に表されているように Frequent Value Compression アルゴリズムで，約 72%削減している．その他の圧縮アルゴリズムも平均すると約 50%削減しており，今回の浮動小数点数アプリケーションの中で最も電力削減できたプログラムである．

また図 4.8 から分かるように，全ての圧縮アルゴリズムでほぼ 100%のブロックに対してデータ圧縮が成功している．最も消費電力が削減出来ていた Frequent Value Compression アルゴリズムは 1/4 圧縮が約 100%のブロックに対して行われた．他の圧縮アルゴリズムでは約 90%以上のブロックが 1/2 圧縮されていた．

実行速度はどのアルゴリズムも約 10%の増加となっている．最も電力削減が出来た Frequent Value Compression もまた約 10%の増加であった (表 4.7)．

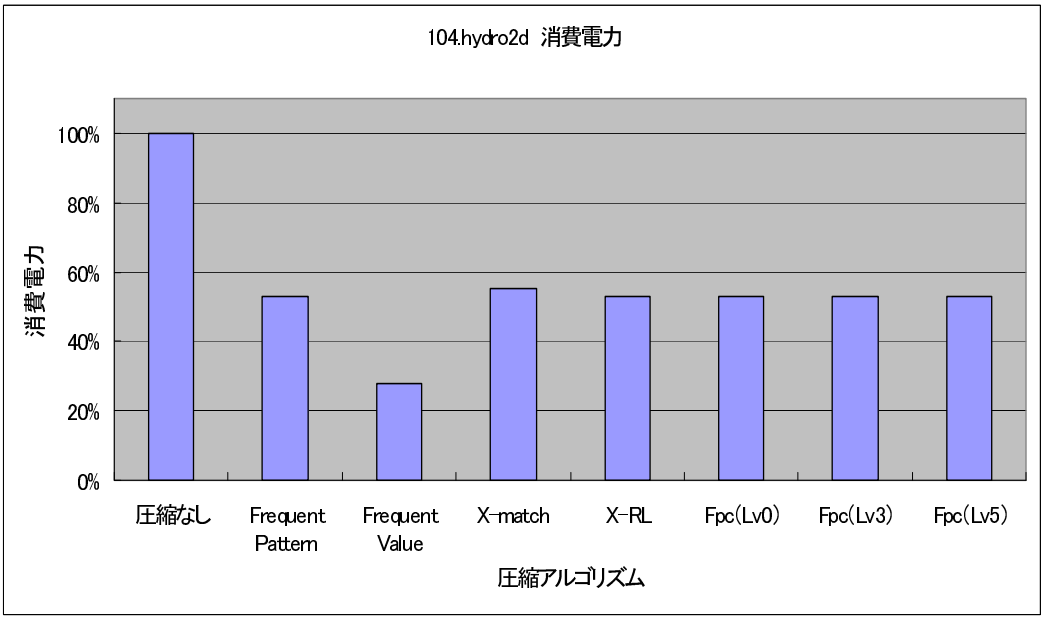

図 4.7: 104.hydro2d 圧縮アルゴリズム別消費電力

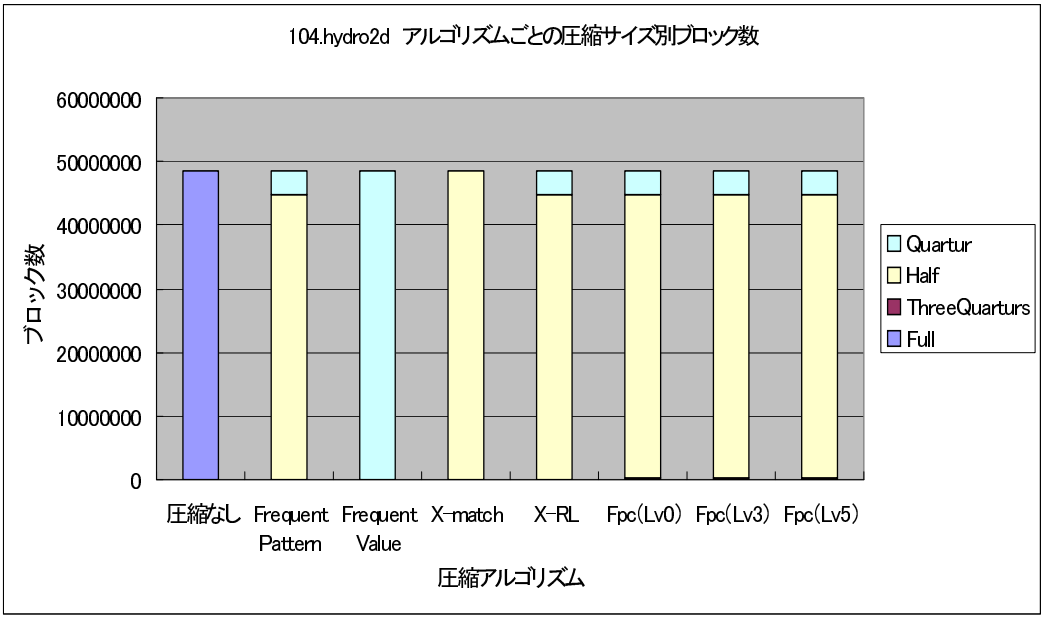

図 4.8: 104.hydro2d 圧縮サイズ別ブロック数

表 4.7: 104.hydro2d クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	7326013184	8079076954	8080506604	8081835394
実行速度	100.00%	110.28%	110.30%	110.32%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	8081847904	8080090224	8080137494	8080156244
実行速度	110.32%	110.29%	110.29%	110.29%

107.mgrid

107.mgrid 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 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 アルゴリズムとなっている．

最も電力削減出来た X-RL で約 15%の電力削減となっている．また，図 4.10 をみると，最も電力削減できた X-RL で約 39%のブロックに対して圧縮が適用されていた．

実行速度は最も電力削減出来た X-RL が最も遅く，約 3.3%の増加となっている (表 4.8)．

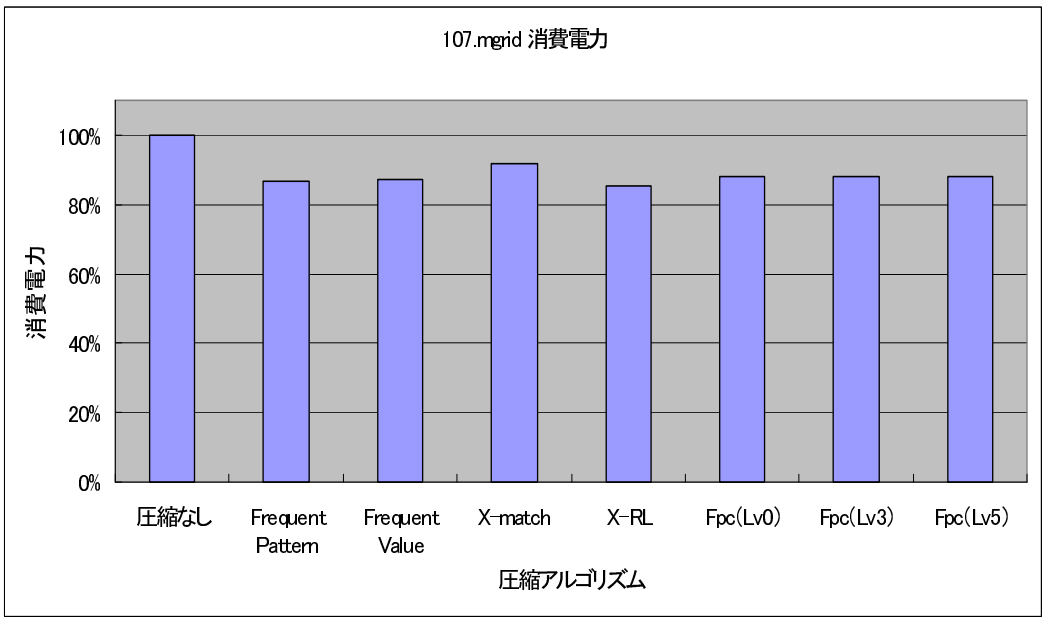

図 4.9: 107.mgrid 圧縮アルゴリズム別消費電力

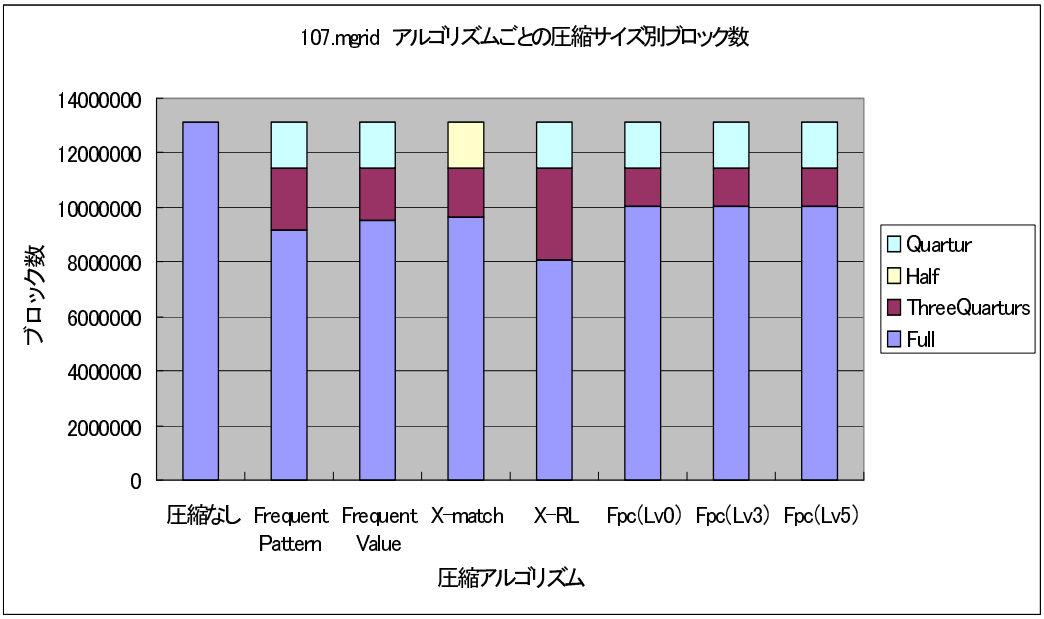

図 4.10: 107.mgrid 圧縮サイズ別ブロック数

表 4.8: 107.mgrid クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	4231630612	4361186942	4357220362	4355592942
実行速度	100.00%	103.06%	102.97%	102.93%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	4372712802	4351082102	4351162042	4351179342
実行速度	103.33%	102.82%	102.82%	102.83%

110.applu

110.applu 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 に表されているようにFPC(Lv5) アルゴリズムで，続いて FPC(Lv3) アルゴリズムとなっている．

最も電力削減出来た FPC(Lv5) で約 9%の電力削減となっている．また，図 4.12 をみると，最も電力削減できた FPC(Lv5) で約 33%のブロックに対して圧縮が適用されていた．

実行速度は Frequent Pattern Compression アルゴリズムが最も遅く，約 4.8%の増加となっている (表 4.9)．最も電力削減出来た FPC(Lv5) は約 4.7%の増加となっている．

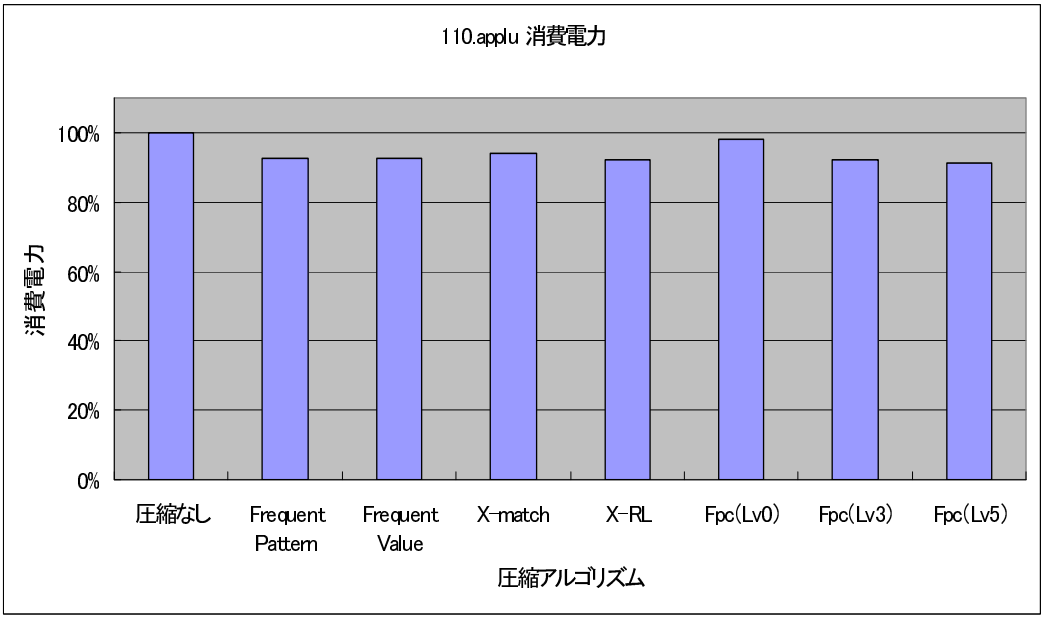

図 4.11: 110.applu 圧縮アルゴリズム別消費電力

表 4.9: 110.applu クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	4831647093	5062998813	5062993603	5062892663
実行速度	100.00%	104.79%	104.79%	104.79%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	5063146513	5029244823	5051948113	5056212673
実行速度	104.79%	104.09%	104.56%	104.65%

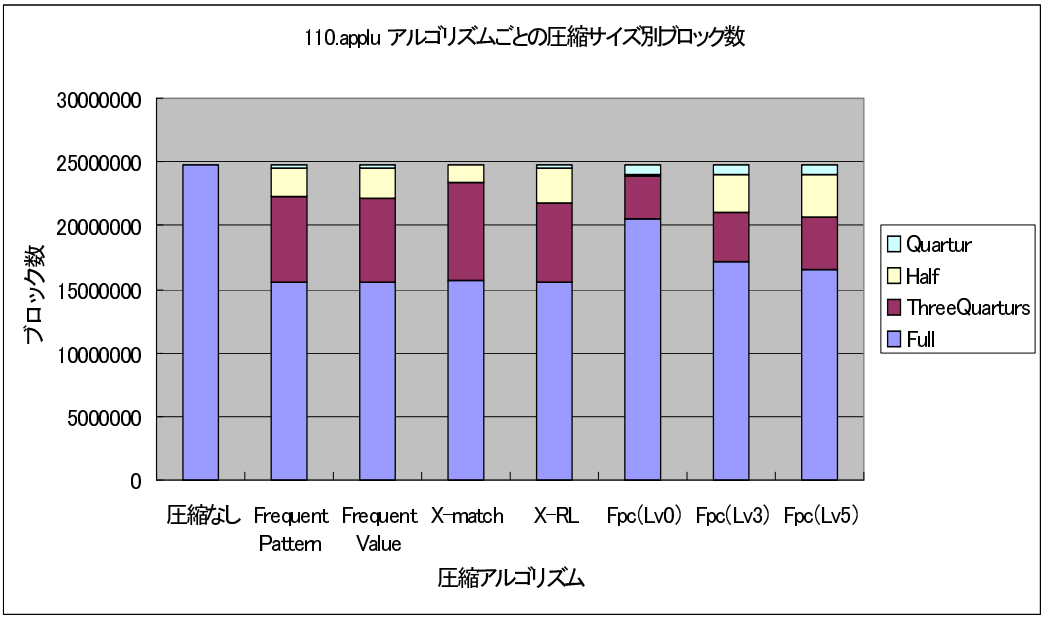

図 4.12: 110.applu 圧縮サイズ別ブロック数

125.turb3d

125.turb3d 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 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 アルゴリズムとなっている．

最も電力削減出来た X-RL で約 24%の電力削減となっている．また，図 4.14 をみると，最も電力削減できた X-RL で約 73%のブロックに対して圧縮が適用されていた．

実行速度は最も電力削減出来た X-RL が最も遅く，約 3.2%の増加となっている (表 4.10)．

表 4.10: 125.turb3d クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	2737387326	2823750986	2823680296	2823432016
実行速度	100.00%	103.15%	103.15%	103.14%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	2824069396	2797938286	2798069476	2798079836
実行速度	103.17%	102.21%	102.22%	102.22%

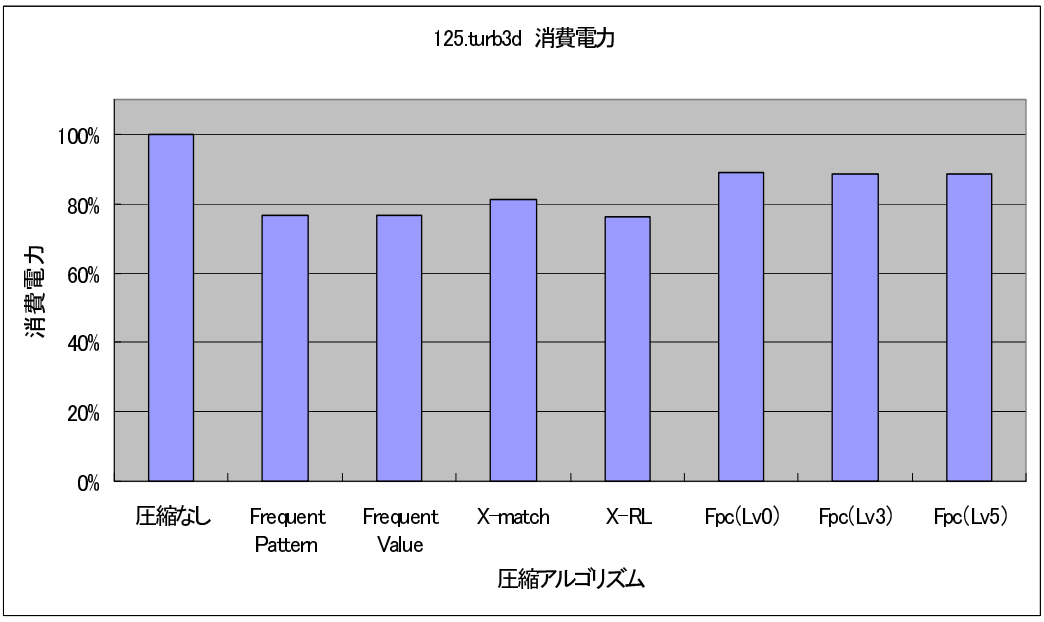

図 4.13: 125.turb3d 圧縮アルゴリズム別消費電力

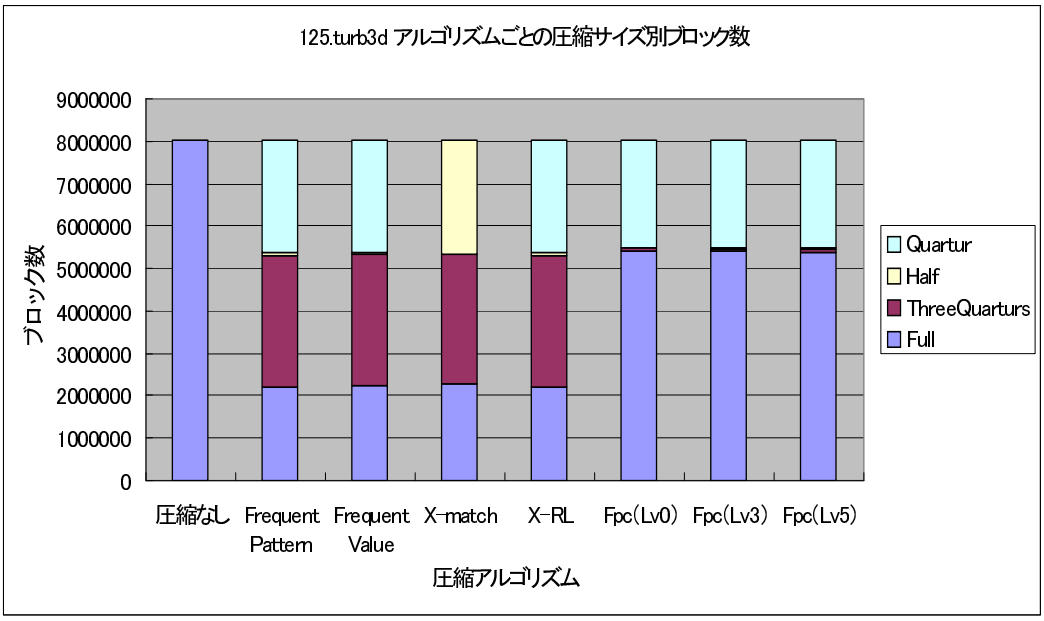

図 4.14: 125.turb3d 圧縮サイズ別ブロック数

141.apsi

141.apsi 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 4.15 に表されているように X-RL アルゴリズムで，続いて X-match アルゴリズムとなっている．

最も電力削減出来た X-RL で約 10%の電力削減となっている．また，図 4.16 をみると，最も電力削減できた X-RL で約 22%のブロックに対して圧縮が適用されていた．

実行速度は最も電力削減出来た X-RL が最も遅く，約 2.8%の増加となっている (表 4.11)．

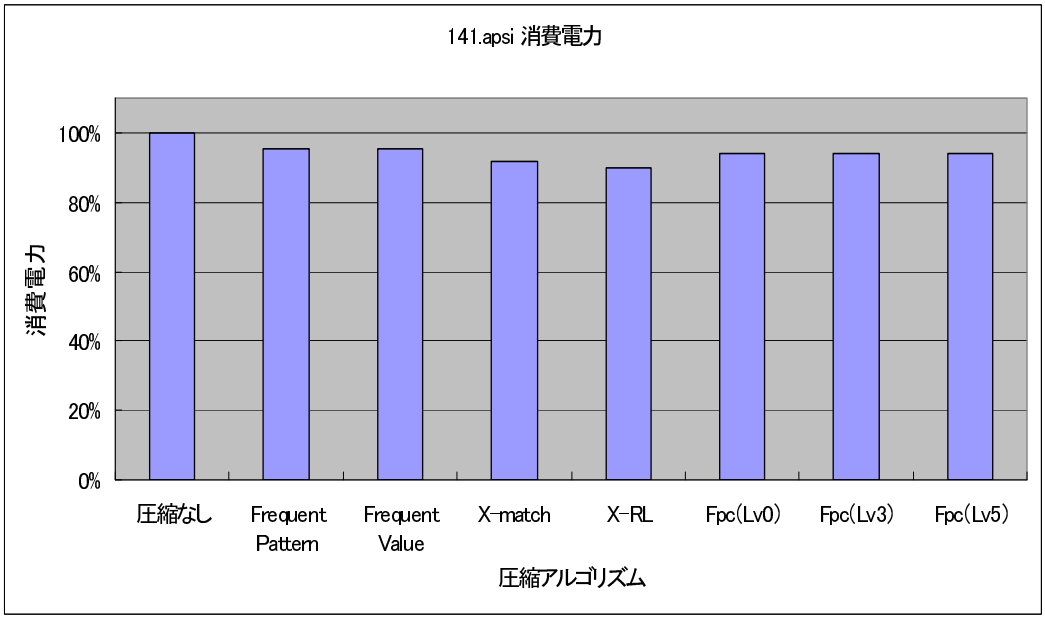

図 4.15: 141.apsi 圧縮アルゴリズム別消費電力

表 4.11: 141.apsi クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	3448020915	3495363665	3496277345	3543789295
実行速度	100.00%	101.37%	101.40%	102.78%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	3543854275	3532675675	3532703655	3532715865
実行速度	102.78%	102.46%	102.46%	102.46%

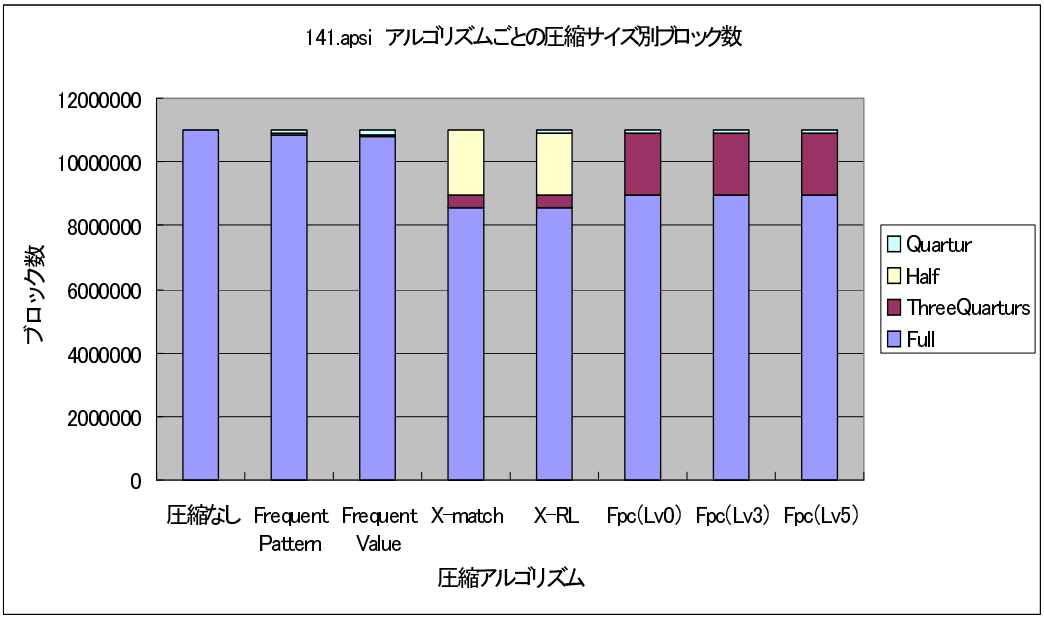

図 4.16: 141.apsi 圧縮サイズ別ブロック数

145.fpppp

145.fpppp 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 に表されているように X-RL アルゴリズムで，続いて X-Match アルゴリズムとなっている．

最も電力削減出来た X-RL で約 35%の電力削減となっている．また，図 4.14 をみると，最も電力削減できた X-RL で約 68%のブロックに対して圧縮が適用されていた．

実行速度は最も電力削減出来た X-RL が最も遅く，約 0.03%の増加となっている (表 4.12)．

表 4.12: 145.fpppp クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	2756796246	2757183996	2757002246	2757664786
実行速度	100.00%	100.01%	100.01%	100.03%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	2757678486	2757513086	2757570876	2757582376
実行速度	100.03%	100.03%	100.03%	100.03%

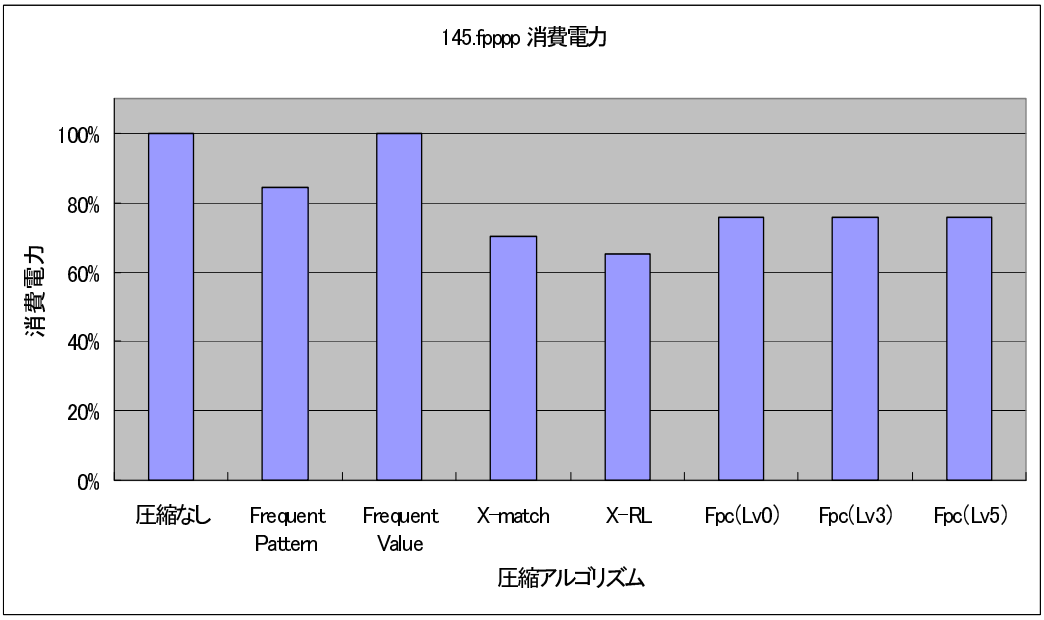

図 4.17: 145.fpppp 圧縮アルゴリズム別消費電力

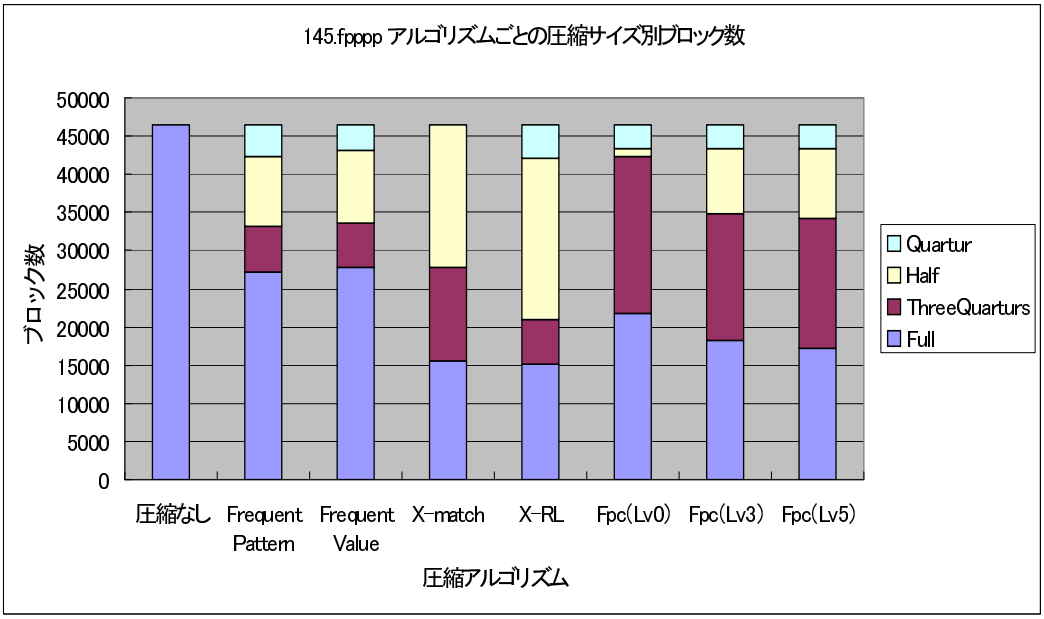

図 4.18: 145.fpppp 圧縮サイズ別ブロック数

146.wave5

146.wave5 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 4.19 に表されているように X-RL アルゴリズムで，続いて FPC(Lv0) アルゴリズムとなっている．

最も電力削減出来た X-RL で約 7%の電力削減となっている．また，図 4.20 をみると，最も電力削減できた X-RL で約 24%のブロックに対して圧縮が適用されていた．

実行速度は最も電力削減出来た X-RL が最も遅く，約 2.8%の増加となっている (表 4.13)．

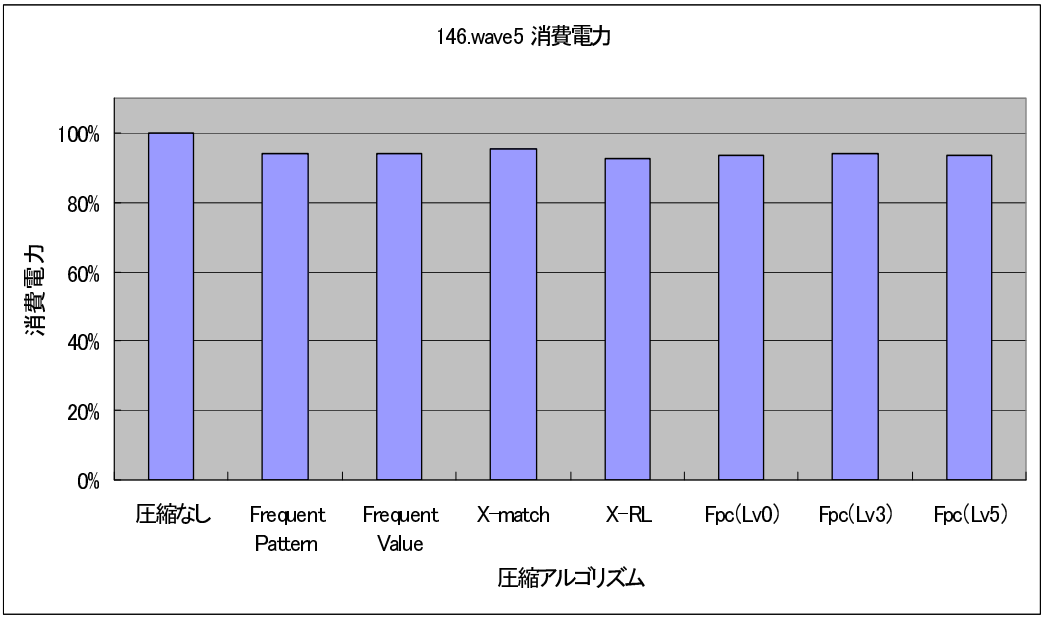

図 4.19: 146.wave5 圧縮アルゴリズム別消費電力

表 4.13: 146.wave5 クロックサイクル

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
クロックサイクル	3457648868	3533595158	3530490338	3554975028
実行速度	100.00%	102.20%	102.11%	102.81%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
クロックサイクル	3555374818	3548993298	3547158128	3548390918
実行速度	102.83%	102.64%	102.59%	102.62%

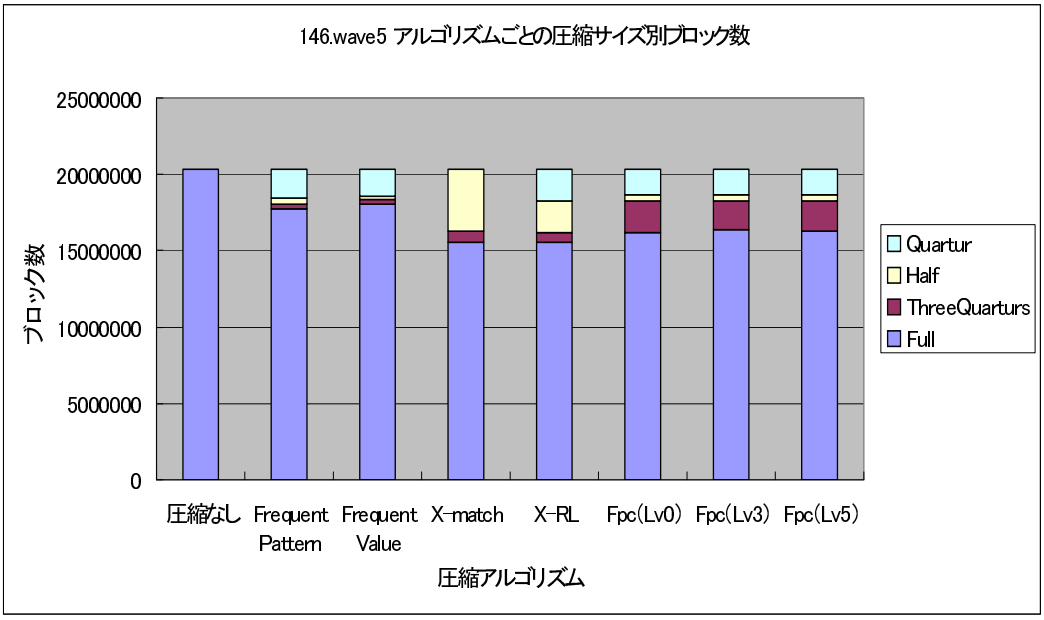

図 4.20: 146.wave5 圧縮サイズ別ブロック数

4.4 考察

SPECfp95の10個の実行プログラムにおいて，電力消費量を平均してまとめると図4.21の結果になった．最も平均して電力消費を削減できたのがX-RLアルゴリズム，続いてFrequent Value Compressionであった．X-RLアルゴリズムは平均で約14%の電力削減である．平均で最も電力削減が出来きなかったFPC(Lv0)は約10%の削減で，X-RLアルゴリズムとの差は約4%であった．

101.tomcatv，102.swim，103.su2corの３つのプログラムでは，全ての圧縮アルゴリズムが非圧縮時に比べ消費電力が増加している．その一方で104.hydro2dではFrequent Value Compressionで約72%電力削減とプログラムによって大きく電力削減量に差がみられた．電力増加となった３つのプログラムは実行速度の増加分が消費電力の増加につながったと考えられる．しかし最も電力削減の効果が見られた104.hydro2dではL1キャッシュサイズ以下の特定のデータパターンが頻繁に使用され，1度L2キャッシュに取り込まれたブロックはほとんどリプレース対象にならなかったと考えられる．

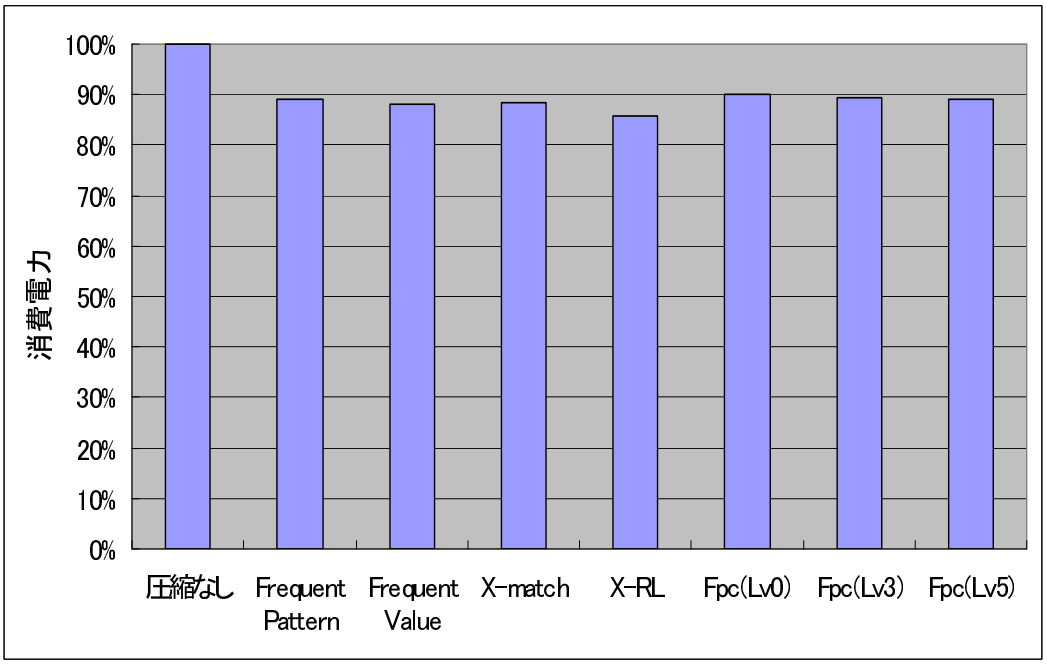

図 4.21: 平均電力量

図4.22はSPECfp95全体の圧縮サイズの割合である．X-RLアルゴリズムでは約41%のブロックが圧縮対象となってL2キャッシュに格納されたことが分かる．

また，表4.14にアルゴリズムごとの平均実行速度を示す．最も実行クロック数が増加したのはX-RLアルゴリズムで，約4.5%増加であった．最も実行クロック数が増加していないFPC(Lv0)との差は0.39%であった．全体としてクロックサイクルの増加は4%〜

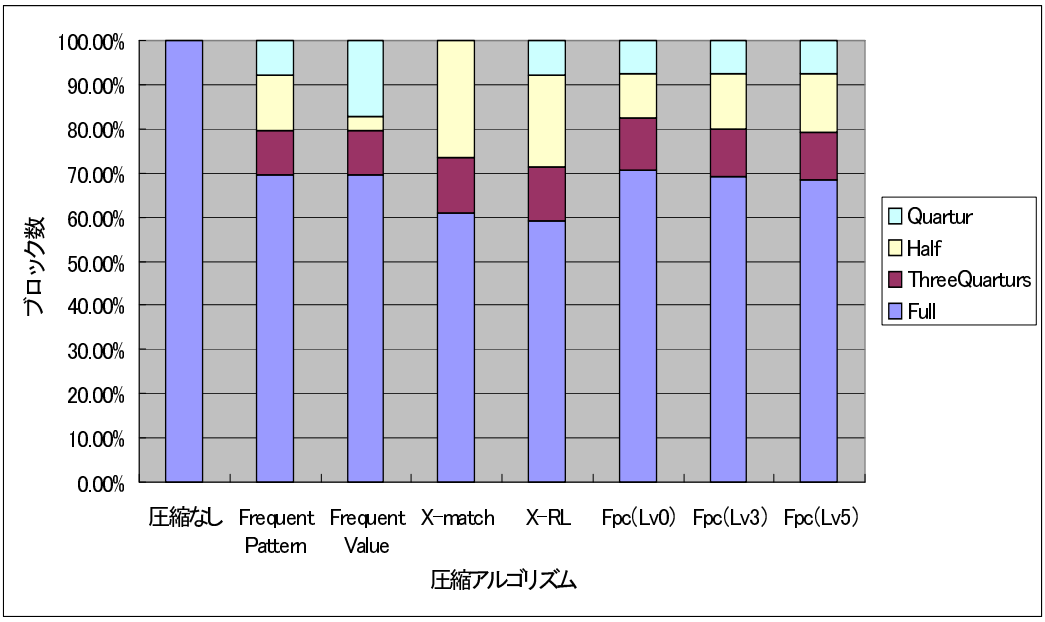

図 4.22: アルゴリズムごとの圧縮サイズ別ブロック数の割合

5%以内に抑えられている．

表 4.14: 平均実行クロックサイクル割合

アルゴリズム	圧縮なし	Frequent Pattern	Frequent Value	X-match
平均実行速度	100.00%	104.16%	104.17%	104.48%
アルゴリズム	X-RL	Fpc(Lv0)	Fpc(Lv3)	Fpc(Lv5)
平均実行速度	104.54%	104.15%	104.18%	104.21%

本研究で浮動小数点数データ向けに新たに追加したFPC アルゴリズムは，電力消費量を平均すると Lv0 で 10.1%，Lv3 で 10.7%，Lv5 で 10.8%削減した．他の 4 つのアルゴリズムに比べると削減量が約 1%〜4%少なかった．

また，FCM，DFCM はテーブルサイズによるデータ圧縮と処理速度の間にトレードオフの関係がある．図 4.22 より圧縮対象となったブロック数の割合は Lv0 で約 29%，Lv3 で約 31%，Lv5 で約 32%とテーブルサイズと比例して増えている．一方表 4.14 より各々圧縮の実行速度は Lv0 で 4.15%，Lv3 で 4.18%，Lv5 で 4.21%の増加とテーブルサイズと反比例して遅くなっている．これによりテーブルサイズによるデータ圧縮と処理速度のトレードオフの関係が確認できた．

第5章　まとめ

5.1　まとめ

本研究では，データ圧縮と電圧制御を用いたキャッシュメモリの消費電力削減方式を再検討した．特に未検証である浮動小数点数データ向けのデータ圧縮に注目し，最適な圧縮アルゴリズムを調査することで，L2 キャッシュの低消費電力化を目指した．また，従来の 4 つのアルゴリズムに加えて Floating-Point Compression アルゴリズムを追加して検証した．

実験の結果，評価対象プログラムの SPECfp95 の 10 個のプログラム実行において，非圧縮時と比較して X-RL は平均 14%削減することができた．平均消費電力が最も削減されたのは X-RL アルゴリズムであった．続いて Frequent Value Compression，X-match アルゴリズムであった．電力削減量は X-RL アルゴリズムで約 14%，Frequent Value Compression で約 12%，X-match アルゴリズムで 11%であった．

この時の平均実行速度は最も電力削減できた X-RL アルゴリズムで 4.5%の増加であった．

また浮動小数点数データ向けに新たに提案した FPC アルゴリズムは，非圧縮時と比較して平均消費電力量を Lv0 で約 10%，Lv3 で約 11%，Lv5 で約 11%削減出来た．平均実行速度は Lv0 で約 4%，Lv3 で約 4%，Lv5 で約 4%の増加であった．

5.2　今後の課題

本研究では，シミュレーションシステムにおいて L2 キャッシュメモリの静的消費電力についてのみ検証した．そのために圧縮ハードウェア自体の規模，消費電力，圧縮に伴うレイテンシ等が考慮されていなかった．実際のプロセッサに本機構を適用するにはこれらの評価が必要である．

謝辞

本研究を行うにあたり，ご指導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准教授に心から深く感謝致します.

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

そして，本学における研究，生活において常に御意見，御助言を頂いた計算機アーキテクチャ講座の皆様に厚く御礼申し上げます.

最後に，これまで支援してくれた家族に深く感謝致します.

参考文献

[1] M.Powell, S.Yang, B.Falsafi, K.Roy, T.N.Vijaykumar. “Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories”, Proc. of ISLPED, pp.90–95, 2000.

[2] 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.

[3] 川原貴裕. “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修士論文 2007.

[4] Luiz Andre Barroso. ‘The Price of Performance: An Economic Case for Chip Multiprocessing’, ACM Queue 2005.

[5] M.Kjelso, M.Gooch, S.Jones. “Design and Performance of a Main Memory Hardware Data Compressor”, Proc.of EuroMicro, pp.423–430, 1996.

[6] S.Kaxiras, Z.Hu, M.Martonosi. “Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power”, Proc. of ISCA, pp.240–251, 2001.

[7] 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.

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

[9] Standard Performance Evaluation Corporation. ‘SPEC CINT95 Benchmarks’, http://www.spec.org/cpu95/CINT95/.

[10] M.Burtscher, P.Ratanaworabhan. “High Throughput Compression of Double-Precision Floating-Point Data”, Data Compression Conference, pp.293–302. 2007.

[11] Y. Sazeides and J. E. Smith. “The Predictability of Data Values”, Proc. of 30th International Symposium on Microarchitecture, pp.248–258. 1997.

[12] B. Goeman, H. Vandierendonck and K. Bosschere. "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency", Proc. of HPCA, pp.207–216. 2001.

[13] Standard Performance Evaluation Corporation. 'SPEC CFP95 Benchmarks', http://www.spec.org/cpu95/CFP95/.

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