特別研究報告書

MPEG エンコーディングにおける再利用と事前実行の比較分析

指導教官　　富田 眞治 教授

京都大学工学部情報学科

小関 悠

2004 年 2 月 2 日

MPEG エンコーディングにおける再利用と事前実行の比較分析

小関 悠

内容梗概

動画像のデジタル符号化形式の一種である MPEG は、今日広く利用されるようになっている。しかし、MPEG エンコーディングは、今なおコンピュータにとって時間を要する処理である。さて、既存プログラムを高速化する手法として投機的マルチスレッディングが提案されている。従来の投機的マルチスレッディングでは、通常実行スレッドと投機実行スレッドが１対１でデータを引き継ぐことにより高速化を図っていた。しかし、投機実行の結果を投機順に関わらず自由に効率良く利用することができれば、より高い性能が得られると考えられる。このような非対称投機的マルチスレッディングの構成方法として、並列事前実行が提案されている。並列事前実行とは、事前実行という手法を用いて、予測された入力から投機スレッドが命令区間の実行を先がけて行い、予測が正しければ通常実行スレッドが実行結果を再利用する手法である。ここで述べる再利用とは、命令区間における入出力を記録することにより、再び同じ区間を実行した際に以前記録した入力値が利用された場合は、正しい出力値を求めることの出来る機構である。

本論文では MPEG エンコーディングにおける並列事前実行の効率的な適用手法を提案し、効果を比較する。一般に再利用機構を実現するためには、入出力を特定できる区間を識別する必要があるものの、本研究ではコンパイラによる専用命令の埋め込みを必要とせず、既存ロードモジュールの実行時に関数およびループを検出することにより、再利用を可能とする機構を仮定する。

本研究では、エンコーダーに mpeg2encode を利用した。処理時間の内訳を調査した結果、総実行時間の約半分をしめる、移動予測時に前後のフレームから抜き出したブロックの差分を測定する関数 (a)、および、約 10%を占める、量子化を行う関数 (b) について高速化が見込めることが分かった。そこで、関数 (a) については、関数内の４つの動作のうち最も頻繁に呼び出される動作 (1) について関数の分割を行い、その上で前後のフレーム間の比較時に、1 ピクセル、2 ピクセル、3 ピクセルずつ調べる関数 (2)、また同様に、1 引数に 4 ピクセル分のデータを詰め込むことで 4 ピクセル、8 ピクセル、12 ピクセルずつ調べる関数 (3) を、それぞれ用意した。加えて、ポインタを用いて 1 行 16 ピクセル分の

データを一度に比較する関数 (4) も用意する一方、それぞれの手法について、ピクセル値にマスクをかけることで入力の多様性を抑える曖昧再利用 (5) を併用する場合についても評価を行った。さらに、事前予測を行わない場合 (6) についてもエンコーディング時間を計測し比較した。

その結果、関数 (a) については、4 ピクセルの比較を一度の関数呼び出しにより行うようプログラムを書き換えた上、曖昧再利用を併用することにより、オリジナルと比較して全体の約 40%、ブロック差分関数部に限って見ると約 59%のサイクル数を削減することができた。ピクセル比較の関数化 (2) は、関数呼び出しが増大するためオーバーヘッドが大きく生じるという問題があったものの、1 つの引数に 4 つのピクセル値を詰め込む (3) により、オーバーヘッドを抑える一方、曖昧再利用を用いることにより入力の多様化を防ぎ、高速化が達成できた。しかし、一度に 8 ピクセル、12 ピクセルの比較を関数化した場合は、入力が多様化するため再利用率が低下し、4 ピクセルの比較を関数化した場合よりも遅くなった。4 ピクセルの比較を関数化した場合は、事前実行なしでもブロック差分関数部の約 40%のサイクル数を削減しており、事前実行を用いてポインタにより行毎の比較を行った場合 (4) と同程度の高速化が達成できた。また、曖昧化によるファイルサイズの増大は、軽微であることも確認できた。一方、量子化については、関数の切り出しにより全体の約 3%を高速化することができた。これらの結果は互いに独立であり、ブロック差分の比較、量子化それぞれについて、以上に提案した手法を適用することにより、最大で約 43%のサイクル数を削減することができた。

A Comparison Between Reuse and Parallel Early Computation on MPEG Encoding

Yu Koseki

Abstract

Recently, MPEG which is one of the digital encoding format for video streams is widely used. Though, MPEG encoding still requires large computation power. Besides, speculative multithreading is proposed as the technique for accelerating the execution. In the case of normal speculative multithreading, a speculative thread changes into non-speculative mode and continues the thread of previous non-speculative execution. However, if a non-speculative thread can use one of many results of succeeding speculative executions, superior performance may be obtained. Parallel early computation is proposed as a method of such speculative multithreading. It is based on early computation which executes speculative threads with predicted input, and if the input is correct, the non-speculative thread reuses the output. The reuse mechanism described here records input and output of each instruction region while executing, and reuse the correct output when the same input appears again.

This paper proposes some technique for parallel early computation on MPEG encoding, and compares those effects. In order to implement the reuse mechanism, it is required to identify each instruction region with its input and output. This paper assumes that the reuse system detects functions and loops dynamically on existing load modules without special purpose reuse instructions inserted be compilers.

In this paper, mpeg2encode is used as an encoder. As a result of analyzing the original processing time, two functions are found to be candidates of parallel early computation. The function (a) which calculates the difference of the block extracted from two frames, consumes about half of the total execution time. The function (a) is called when the encoder predicts the movement between two frames. And the function (b) which quantizes data, consumes about 10% of the total execution time.

The function (a) contains four operations. I picked up a instruction region most frequently executed, and rewrote as a function. I constructed several sub-

functions (2) that compare 1, 2 or 3 pixels at each call for the comparison of two contiguous frames. Moreover, I packed 4 pixels in each argument of the function, and constructed sun-functions that compare 4, 8 or 12 pixels at each call. In addition, I designed another function (4) with 2 arguments where each argument holds the pointer to the starting address of contiguous 16 pixels. I also evaluated a tolerant reuse technique (5) which reduces the preciseness of the arguments by masking several lower bits. The base architecture is non-reuse and non early computation mechanism (6).

As the result, function (a) with 4 pixels-comparison with tolerant reuse could eliminate about 40% cycles against whole of the original program, and about 60% cycles of target function (a). Function (a) with 1, 2 or 3 pixels showed some problem that the overhead of the function call itself became very large, because of the high frequency of the function call. On the other hand, the function with packed pixels and tolerant reuse showed remarkable speed-up, because small number of function calls. However, the function with 8 or 12 pixels at each call showed less performance than 4 pixels. It is considered that the reusability becomes lower as increasing the width of arguments. I also discovered that the early computation is not so effective in the function with packed pixels, the function (4) with pointers worked well with the early computation, and these functions showed comparable performance. As for the size of compressed files, it is discovered the tolerant reuse does not increase the size too much. On the other hand, about quantize function (b), eliminate about 3% with function-ize. Because each result is independent, using above techniques in both function (a) and (b), the maximum ratio of cut down cycles reaches 43%.

MPEG エンコーディングにおける再利用と事前実行の比較分析

目次

第 1 章　はじめに　1

第 2 章　再利用と並列事前実行　2

2.1　並列事前実行 2

2.2　再利用 5

2.2.1　関数再利用 5

2.2.2　ループ再利用 7

第 3 章　MPEG の特徴　7

3.1　MPEG のエンコーディング行程 8

3.1.1　YUV 変換 8

3.1.2　移動予測 9

3.1.3　フレーム差分取得 10

3.1.4　DCT 11

3.1.5　量子化 11

3.2　実行時間の分析 11

第 4 章　再利用の適用手法　12

4.1　ブロック差分の測定の高速化 12

4.1.1　関数の分割 15

4.1.2　ピクセルずつの比較 16

4.1.3　ピクセルデータの詰め込み 17

4.1.4　ポインタを用いた行毎の比較 18

4.1.5　曖昧再利用 19

4.2　量子化の高速化 20

第 5 章　性能評価　21

5.1　ブロック差分関数の評価 21

5.2　量子化関数の評価 25

第 6 章　まとめ　25

謝辞　26

第1章　はじめに

計算機の高速化が進む中で、かつては専用機でなければ扱いが困難であった動画像データは、今日汎用のパソコンでも頻繁に利用されるようになった。例えば動画像のデジタル符号化形式として事実上標準となっている**MPEG**(Motion Picture Expert Group) は、ビデオ CD や DVD といった当初に想定されていた利用目的のみならず、放送、ハードディスクレコーディング等のデータ蓄積、携帯電話によるビデオメールなどにまで広く利用されるようになっている。しかし動画像をカメラなどから取得し、軽量で一般的に扱いやすいフォーマットに変換するエンコーディング作業は、今なおコンピュータにとって時間を要する作業であり、特に放送のようにリアルタイム・エンコーディングが必用な場面において、より高速な動作が求められている。

一方、既存プログラムを高速化する手法として**投機的マルチスレッディング** (Speculative Multi-threading:以下**SpMT**) がある。従来の SpMT では通常実行スレッドと投機実行スレッドが 1 対 1 でデータを引き継いでいた。しかし通常スレッドが複数の投機スレッドによる結果を、投機した順に関わらず自由に効率良く利用可能であれば、より高い性能が得られると考えられる。しかしこの実現のためには通常実行スレッドと投機実行スレッドの間で、1 対多のデータ引き継ぎが可能となる必要がある。現在、こうした非対称な SpMT として、**並列事前実行** [1] が提案されている。

並列事前実行は、**事前実行**という手法を用いて予測された入力から投機スレッドが命令区間の実行を先がけて行い、予測が正しければ通常実行スレッドがその結果を**再利用**するという手法である。

ここで用いられる再利用とは、プログラム中の区間において実行される度に入力と出力の組を再利用表に記録しておき、次回以降同じ区間が以前に与えられた入力で実行された場合、実際の演算を行うことなく前回に記録しておいた出力を用いるという手法である。切り出す区間における演算の複雑さと再利用機構には関係性が無いので、一般には時間のかかる演算であっても再利用可能であれば高速に正確な値を得ることが出来、また対象の命令区間数が増えても再利用機構の複雑さは増大しないという利点がある。この再利用機構が存在することで、並列事前実行では投機スレッドによって得た結果を通常スレッドで効率的に用いることが可能となっている。

また、単純な再利用では過去に同じ入力で当該区間の実行が行われている場合に限り再利用可能であるのに比べ、事前実行を用いた場合では、当該区間の入力を予測して投機実行を行うため、入力が単調に変化する場合などにも再利用が可能となり、高速化が図れる。

これまで再利用を用いた画像や音楽データのエンコーディング高速化については、JPEG エンコーダを用いた報告 [2] がある。この報告ではコンパイル時に専用命令を埋め込むことにより区間再利用を実現し、通常の適用では 8%、加えて入力の一致判別に寛容さを持たせる曖昧再利用を用いることで、25〜40%の命令数削減に成功している。曖昧再利用により画質は劣化するものの、劣化の度合は軽微であることが報告されている。また MP3 のエンコーディングにおいては、曖昧再利用を用いた報告 [3] によると、10〜30%の命令数削減に成功している。加えてステレオ画像処理の視差測定部において、曖昧再利用を用いて最大 90%のサイクル数削減に成功したという報告もある [4]。

これらを踏まえて本論文では MPEG のエンコーディングにおける並列事前実行の効率的な適用手法を提案し、それらを比較・検討する。以下では並列事前実行と再利用の仕組みについて述べたあと、MPEG の圧縮技術について説明し、エンコーディング過程の分析を行う。そして特に時間を必要とする移動予測のブロック差分関数と量子化関数について、それぞれより再利用が可能となる手法を提案する。最後にそれらの手法について評価を行う。

第 2 章　再利用と並列事前実行

本章では本研究の背景となる、並列事前実行と再利用の機構とその特徴について述べる。

2.1　並列事前実行

並列事前実行は Main Stream Processor(以下 **MSP**)1 つに対して Shadow Stream Processor(以下 **SSP**) が複数存在する、非対称な SpMT の一種である。概要を図 1 に示す。MSP は通常実行を行うプロセッサであり、可能な場合は命令区間の再利用を行う。一方、SSP は MSP と並行して投機的実行を行う。演算器、レジスタ、キャッシュは各プロセッサごとに独立しており、再利用表、主記憶は全プロセッサで共有する。

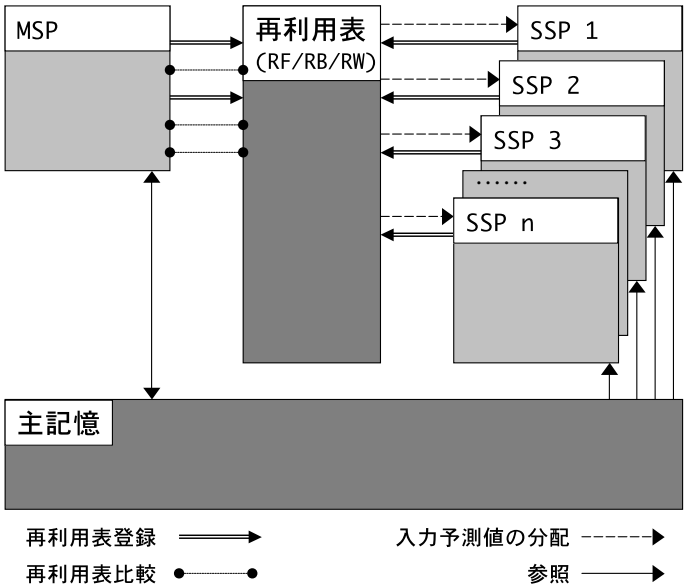

図 1: 並列事前実行の構造

並列事前実行では、再利用表が MSP と SSP による 1 対多のデータ引き継ぎを可能としている。再利用表は関数管理表 (RF)、入出力記録表 (RB) などから構成されている。RF は再利用対象となる命令区間のアドレス、および読み出しと書き込みで参照されるアドレスを保持し、1 エントリが 1 命令区間に対応している。RB は命令区間の実行が開始されたときのスタックポインタの値や命令区間の入出力を記録するための表である。命令区間の実行時には参照されたレジスタおよび主記憶のアドレスを RF に、参照された値を命令区間の入力として RB に、それぞれ登録する。書き込みが発生したレジスタおよび主記憶についても同様に出力として登録を行う。

一般的な SpMT とは異なり MSP は通常実行のみを行い、自身あるいは SSP が RB に登録したエントリで再利用を行う。この機構では SSP は常に投機実行だけを行うので、一般の SpMT のように失敗した投機実行をキャンセルする必要が無いという利点がある。一方この機構には、いかにして投機実行の対象となる命令区間を選択するか、いかにして入力を予測し SSP に割り当てるか、いかにして主記憶一貫性を保つか、というような課題が存在する。

一般に同一パラメタが出現する間隔の長い命令区間や、パラメタが単調に変化し続ける命令区間に対しては、投機実行による効果が高いと予想出来る。しかしこうした各々の命令区間の性質や投機実行による削減ステップ数は、事前には分からない。このため、RF に新規に登録された命令区間については、直

ちに SSP による数回分の事前実行を試みる。その結果、対象の RF が MSP による高い登録頻度を持ち、かつ SSP が登録したエントリの再利用頻度も高い場合、事前実行による効果が高いと見て継続して SSP による投機対象とする。

この動的に変化する登録頻度および再利用頻度を把握するため、一定期間における登録や再利用の状況を示すシフトレジスタを RF 毎に付加する。また同じく RF 毎に小さなハードウェアを付加し、各エントリについて、その有効性 $E =$ (過去の省略ステップ数) $\times$ (登録回数) $\times$ (再利用回数) を計算する。各 SSP は E が最大となる RF エントリを選択し、1 セットの予測引数を受け取り、その区間の投機実行を行う。実行が終わった SSP は再び RF の参照を行い、次の引数セットを受け取るという動作を繰り返す。これにより、常に再利用頻度が高いか削減ステップ数の多い命令区間について投機実行を行うことが出来る。

また、効率的な投機実行のためには RB の使用履歴に基づいて将来の入力を予測し、SSP に渡す必要がある。このため RF の各エントリごとに小さなハードウェアを設け、MSP や SSP とは独立に入力予測値を求める。具体的には最後に出現した引数 a、および最近出現した 2 組の引数の差分 d に基づき、ストライド予測 [5] を行う。$a + d$ に基づく命令区間の実行は MSP が既に開始していると考え、n 台の SSP が存在する場合は i 番目の SSP に対して入力予測値 $a + d \times (i+1)$ を渡す。全ての SSP に割り当てた次の引数、即ち $a + d \times (n + 2)$ は MSP に割り当てることになる。

一方、SSP は予測された入力により命令区間を実行するため、SSP により書き込まれた主記憶値は MSP により将来的に書き込まれる正しい値とは一般に異なるという課題もある。このため、キャッシュおよび主記憶に値を書き戻すことが出来るのは MSP のみとしている。各 SSP は局所変数以外を記録するため、キャッシュや主記憶のかわりに専用の RB エントリを用いる。また、スタックエリア内の局所変数を記録するため、局所メモリを使用する。MSP が主記憶に対して書き込みを行った場合は、対応する SSP のキャッシュラインは無効化される。RB への登録対象のうち、読み出しが先行するアドレスについては主記憶を参照し、MSP 同様アドレスおよび値を RB へと登録する。これにより以後、主記憶ではなく RB を参照することで他のプロセッサからの上書きによる矛盾の発生を避けることが出来る。一方で局所メモリに対する、書き込みに先行する読み出しアクセスは変数を初期化せずに使うことに相当し、値は不定で構わないと見なし主記憶を参照しない。関数フレームの大きさが局所メモリ容

量を超えた場合はSSPの投機実行を打ち切る。

更に命令区間の入力である主記憶参照アドレスが書き換えられた場合、主記憶一貫性をいかに保つかという課題もある。これについてはstore時に対象アドレスを参照しているRFには入力エントリにフラグを立て、再利用時にRBにおけるそのアドレスに対応する値のみを比較するという手法を採用する[6]。これにより、store命令が生じていないような場合には主記憶比較を省略することでオーバーヘッドを抑える一方、主記憶参照アドレスが変更となったエントリを削除させないことで再利用率を損わないということが可能となっている。これは、RBの参照するアドレスが頻繁に読み出される一方、書き換えられることが少ないような場合、特に有効である。

2.2 再利用

再利用とは、命令区間における入出力を記録することで、再び同じ区間を実行した際に以前記録した入力値が利用された場合は、正しい出力値を求めることの出来る機構である。そのため一般に再利用を実現するためには、入出力を特定出来る区間を識別する必要がある。こうした手法には専用命令をコンパイラに埋め込ませてハードウェアにより識別を実現するもの、ソフトウェアを用いるもの、その双方を用いたものと、様々なものが提案されている。本研究ではSPARC ABI(Application Binary Interface)の規定に基づき区間を自動的に判別する機構[1]を仮定する。この機構には、プログラムにおいて再利用のための専用命令が不要であり、既存のロードモジュールをそのまま利用出来るという利点がある。

一般的なプログラムにおける命令区間としては、関数とループが挙げられる。以下ではそれぞれについて、再利用実現のための手法を述べる。

2.2.1 関数再利用

関数再利用を実現するためには、関数管理表(RF)および入出力記録表(RB)が必要となる。一つの関数を再利用するためのハードウェア構成を図2に示す。複数の関数を再利用する場合には、本構成が複数必要となる。

各表のVは有効なエントリであるか否かのフラグである。またLRUはそのエントリが参照された頻度を示すカウンタ値のためのフィールドで、エントリが上限を超えて入れ替える必要が生じた場合に参照する。読み取りアドレスはRFが一括管理し、マスク値および値はRBが管理する。これにより読み出しア

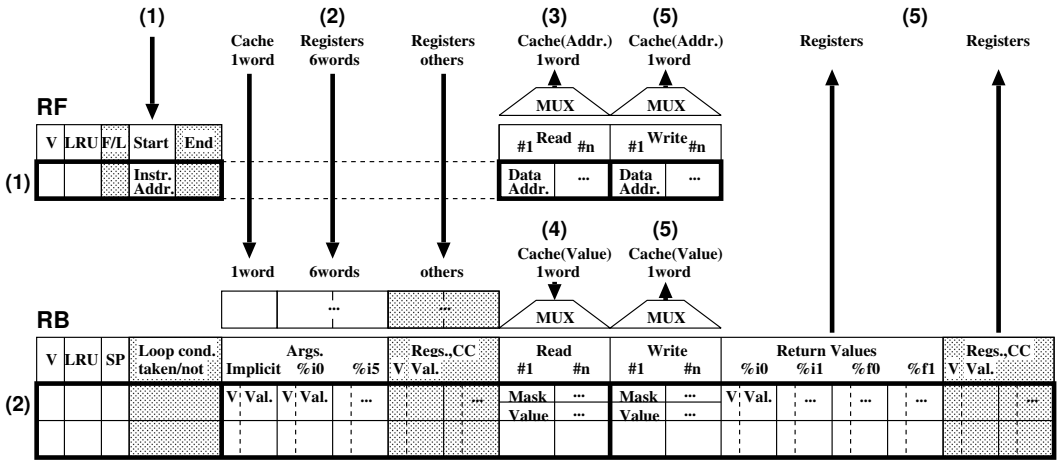

図 2: 再利用表の構成

ドレスの内容と RB の複数エントリを CAM(Contents Addressable Memory) により一度で比較することが可能になっている。関数の先頭アドレス、および読み出しと書き込みで参照される主記憶アドレスは RF によって保持されている一方、関数が呼び出されたときのスタックポインタの値 (SP)、その関数に渡された引数、主記憶の読み出しと書き込みデータ、および関数の返り値は RB によって保持される。返り値は%i0〜1(リーフ関数の場合は%o0〜1) または%f0〜1 に格納され、%f2〜3 を使用する返り値 (拡張倍精度浮動小数点数) は対象プログラムには存在しないものと仮定している。なお、各エントリ先頭の V はそのエントリの有効性を示すフラグであり、Mask はそのエントリの有効バイトを示すマスク値である。

実際に関数を再利用しようとする場合は、実行時に局所変数を除外しながら、引数、返り値、大域変数および上位関数の局所変数に関する入出力情報を登録する必要がある。読み出しが専攻した引数レジスタは関数の入力として、返り値レジスタへの書き込みは関数の出力として登録する。主記憶参照も同様に読み出しが先行したアドレスについては入力、書き込みが先行した場合は出力として登録する。その後、復帰命令を実行した時点で登録中のエントリを有効とする。ただし復帰までの間に他の関数の呼出、入出力数の容量オーバー、引数の第 7word の検出、システムコールや割り込みの発生などが生じた場合は、その時点で登録を打ち切る。

以後は関数を呼び出す前に (1) 関数先頭アドレスを検索し RF に登録済かを調べ、(2) 引数が完全に一致する RB エントリを探し、(3) 主記憶読み出しデータ

を参照し、(4) 一致比較を行い全ての入力が一致しいば場合、(5) 登録されている出力、すなわち返り値、大域変数、局所変数を書き戻すことで、関数の実行を省略出来る。

2.2.2 ループ再利用

関数に含まれる命令は call/jump 命令の分岐先から return 命令までである。同様にループは、後方分岐命令の分岐先から、同じ後方分岐命令までである。従ってこの間の入出力を登録しておくことで、関数同様にループを再利用することが出来る。ただし関数の return 命令とは異なり、分岐先が同じであるような各分岐命令のアドレスは同じであるとは限らない。このため、ループの再利用では分岐先アドレスも RB に格納しておく必要がある。また、ループ内の局所変数は動的に識別不可能であることも関数の場合とは異なり、参照されたレジスタおよび主記憶アドレスの全てを記録しておく必要がある。

再利用をループに適用する際に必要となる拡張は、RF では関数とループの区別 (F/L) とループの終了アドレス (End)、RB ではループ終了時の分岐方向 (taken/not)、引数や返り値以外のレジスタおよび条件コード (Regs,CC) である。

ループが完了するより前に関数から復帰したり、前節で示したような入出力の容量オーバーなどによりループの入出力登録が中止されなかった場合、登録中のループに対応する後方分岐命令を検出した時点で登録中の入出力表エントリを有効にし、ループの登録を完了する。さらに後方分岐命令が成立する場合には、次ループが再利用可能であるかどうかを関数と同様の手順で判断する。再利用した場合は RB に登録されている分岐方向に基づいて、さらに次のループに関して同様の処理を繰り返す。一方、次のループが再利用不可能である場合は、さらに次のループの実行と RB への登録を開始する。

第 3 章　MPEG の特徴

本章では今日、マルチメディア符号化の分野で事実上の標準となりつつある MPEG の特徴について述べる。MPEG には主に動画像と音楽についての規定があり、またその用途や必要とされる画質に合わせて MPEG-1/MPEG-2/MPEG-4 などの種類が存在する。

MPEG のような動画像を扱うデジタルデータは、通常一枚一枚の画像を次々と表示することで動いているように見せている。この動画像中の一枚の画像の

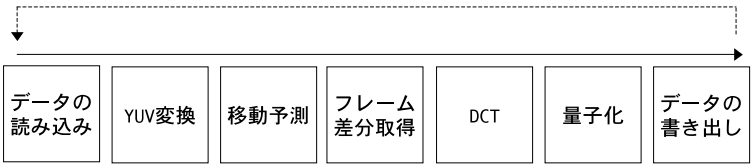

図 3: MPEG のエンコーディング行程

ことを**フレーム**と呼ぶ。

効率の良い符号化によるデータ圧縮のため、MPEG では様々なアイデアが用いられている。MPEG における 1 フレーム分のエンコーディングの処理を図 3 に示す。エンコーダーはファイルやカメラから得た 1 フレーム分の画像データを**YUV 変換**にかけ、その後**移動予測**を行う。更に**フレーム差分取得**を行い、得た値を**離散コサイン変換** (Discrete Cosine Transform:以下 **DCT**) にかける。この結果 YUV 値から波の成分へと変換された画像データは、**量子化**によって圧縮され、最後にファイルへ書き出される。エンコードする画像データがまだ存在する場合は、この処理を繰り返す。

以下ではデータ読み込みと書き出しを除いた 5 つの行程について、行程順に従ってそれぞれの処理内容と特徴について述べる。加えて章の最後では実際にエンコーディングを行い、各処理ごとの必要時間を調べ、再利用可能性を探る。なお MPEG-2 は DVD などの高画質用途のため、MPEG-4 は特許の問題が存在するため、本論文では MPEG-1 を用いてエンコーディングを行う。しかし以下に述べる特徴は MPEG 一般に共通のものである。

3.1 MPEG のエンコーディング行程

3.1.1 YUV 変換

人間の目は色の変化よりも明るさの変化に対してより敏感であることが知られている。MPEG はこの特徴を用い、画像データを一般的な色成分表現である RGB 信号ではなく、輝度信号 Y、輝度信号と赤色成分の差 U、輝度信号と青色成分の差 V による YUV 信号を用いて表現している。その上で輝度信号に十分なビット数を与え、見る人にとって違和感を失わせない一方、それ以外の情報を間引くことで効率的に圧縮をしている。

カメラからの入力や一般の画像ファイルなど、MPEG エンコーディングのソースとなるデータでは通常 RGB が用いられているため、読み出されたデータはエ

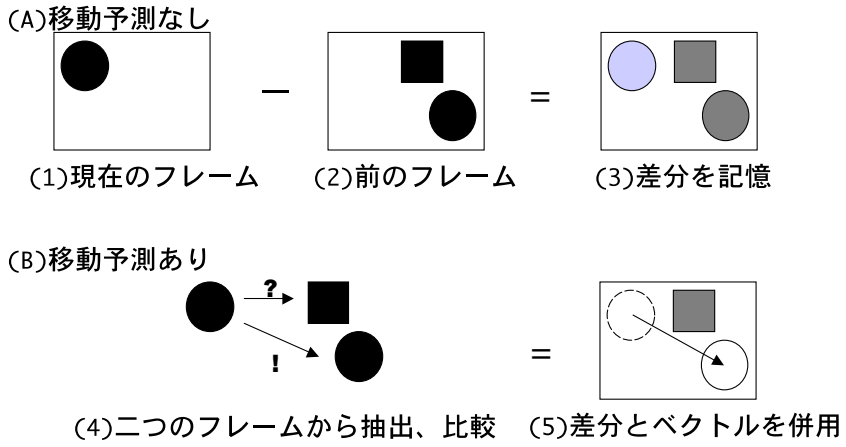

図 4: 移動予測

ンコーディング過程においてまず最初に RGB から YUV への変換が行われる。

3.1.2 移動予測

移動予測とは、あるフレームにおいて一部が前のフレームから変化した時、その変化が前のフレームのどの部分から移動して来たものなのかを調べるというものである。

例えば図 4 のようにボールが (1) 左上から (2) 右下へ移動して来たとする。この時 (2) の情報を全て保持するか、あるいは後述するフレーム差分取得を行って (3) のようなデータを記録することは出来る。しかし (4) このボールが同じ形状であることを理解出来れば、(5) 左上から右下への動きベクトルを求めて (2) や (3) のデータのかわりにこれを記録すれば、データを圧縮することが出来る。なお移動予測では、YUV の Y 値がパラメータとして用いられる。つまり輝度を基準に、フレーム間の移動検出が行われる。

一般に移動予測は、(4) に示すようにフレームから一定サイズのピクセルブロックを取り出し、それに似たブロックを前フレームから検出するという仕組みで実装されている。フレーム中のピクセルブロックはおおまかに言って、前のフレームから移動せずにそのままの場所にあるか、どこか別の場所から移動してきたか、前のフレームに存在したものが大きく変化して現れたか、フレーム外などから突然現れたかのどれかであると言える。よって取り出されたピクセルブロックは、前フレームから次々にピクセルブロックを取り出して一つ一つ類似性を比較することで、最も似かよった、移動先と思われるピクセルブロックを発見するか、あるいはどこかへ消えてしまったと認識することが出来る。

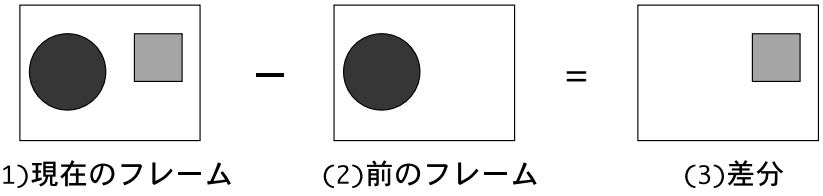

図 5: フレーム差分取得

MPEG で生成する動画像は 1 秒間に 10 フレーム以上を含む場合がほとんどなので、データがリアルタイムに何かを映したものであれば現在のフレームに存在するデータは一つ前のフレームにも存在する場合が多い。そのため、動きベクトルを求めることでデータを大きく圧縮することが出来る。

なお、フレームによっては前のフレームからの移動予測だけではなく、後のフレームからの移動予測も同時に行う。この場合も基本的な原理は同様である。

移動予測はこのようにピクセルブロックの取り出しと比較を大量に行うため、MPEG のエンコーディング作業において一番時間を要する部分である。そのため MPEG の圧縮率とエンコーディング速度は、動きベクトル検出のバランスによって変化するとも言える。例えば、動きがほとんどないような画像セットから動画を作成する場合は、動きベクトルを全く求めなくとも十分な圧縮率を得ることが出来る。しかし一方で動きの激しい画像セットから作成する場合、動きベクトルを求めなければ圧縮率は著しく低下する。ただし先にも述べた通り、動きベクトルの検出には非常に時間がかかる。よって、最低限の圧縮率を保持しつつ、十分な速度を得ることの出来る仕組みが求められる。

3.1.3 フレーム差分取得

動きベクトルによりデータの表現が可能であった部分を除くと、MPEG では一枚一枚のフレームについて、図 5 に示すように直前のフレームとの差分を記録している。例えばカメラが静止している状態で 15fps(frames per second) のエンコーディングを行う場合は、1/15 秒のような短い時間での変化は一般的にあまり大きくない。よってフレーム全体を保持するよりも、差分画像の方が少ないデータ量で済む場合が多い。この仕組みは、似たようなフレームが続く動画像で特に有効である。

この仕組みは言い変えればフレーム間の静止部分は記録せずに、変化のある部分だけを記録しているということである。そしてその変化のある部分については前項の移動検出により、動きベクトルでデータが表現出来ている可能性が

高い。このようにMPEGでは動きのある部分については移動予測、無い部分についてはフレーム差分の取得を行い、それぞれ効率的にデータを圧縮している。

3.1.4 DCT

移動予測とフレーム差分の取得によって圧縮されたデータは、予めパラメータで定められたピクセルブロックに分けられ、それぞれDCTにかけられる。DCTとはデータをコサイン波の重ね合わせで表現するもので、これによりデータを表現するYUV値は定められた波の振幅へと変換される。DCTによって得られる値の数は、与えたデータの数と同じである。例えば入力となるピクセルブロックのサイズが8*8であったとすると、DCTによる出力は64個の波のそれぞれの振幅となる。一般的にDCTによって得られたそれぞれの波は、波長の長いものほど振幅も大きく、波長が短くなるに従って振幅も小さくなるという性質がある。

3.1.5 量子化

DCTによった得られた波の成分は実際にはそのままファイルに書き出されるのではなく、近似値が用いられる。この近似を行うことを量子化と呼ぶ。前述の通り通常、DCTによって得られた波の振幅は波長が短くなるほど小さくなる。そのため近似を行うと一般的には、ある程度より波長の短い波は振幅が全てゼロになる。よってこうした波については記録せず、残った波だけを記録することで圧縮を行う。これにより失われるデータは、波長も振幅も小さな波で表されるものであり、全体から見て軽微であると考えられる。

3.2 実行時間の分析

MPEGエンコーディングの動作について、その特徴を見るため実際にエンコーディングし、関数毎の占有時間を見た。なお、エンコーダにはこの研究を通してmpeg2encode (Copyright (C) 1996, MPEG Software Simulation Group.)を用いている。プロセッサはIntel Xeon 2.8GHzを使用し、入力となる画像は240*180ピクセルのppm形式ファイル100枚とした。

各行程の所要時間を図6に示す。なお、全体の実行時間は7.45秒であった。

一番の割合を占めるのは**ブロック差分の測定**である。この行程は移動予測時、正しい動きベクトルを求めるために用いられる。全体から見るとこの行程が実行時間の半分以上を占めており、この行程の高速化無しにエンコーディング全体の高速化は期待出来ないと言える。

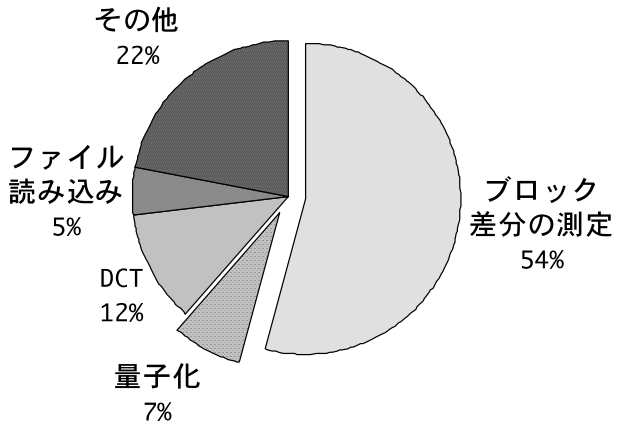

図 6: 実行時間を多く占める関数

続いて割合を占めるのはDCTであり、その次に量子化、ファイル読み込みと続く。DCTは一度の入力が多く多様なため、再利用による高速化が期待出来ないと考えられる。一方で量子化については一つ一つの値について近似を行うだけなので、再利用による高速化が期待出来る。ファイル読み込みはデバイスとのアクセスに時間を要しており、これは再利用での高速化は望めない。

よって、本研究ではブロック差分の測定と量子化のそれぞれについて再利用可能性を探ることとした。

第 4 章　再利用の適用手法

本章では再利用可能性とそれによるエンコーディングの高速化の可能性が高いと予想される、ブロック差分の測定および量子化についてまとめる。そして各々について処理を高速化するための手法を提案する。

4.1　ブロック差分の測定の高速化

本研究で用いたmpeg2encodeでは、ブロック差分の測定は関数`dist1`によって実装されている。関数`dist1`は前後2つのフレームから切り出したピクセルブロックについて、その2つがどれだけ類似しているかを判別し、その類似度を返す。この、移動予測のために切り出されたピクセルブロックのことをマクロブロックと呼ぶ。フレーム間で移動予測を行う場合、マクロブロックは16*16ピクセルである。

与えられた2つのピクセルブロックがどれだけ類似しているかを示すため、

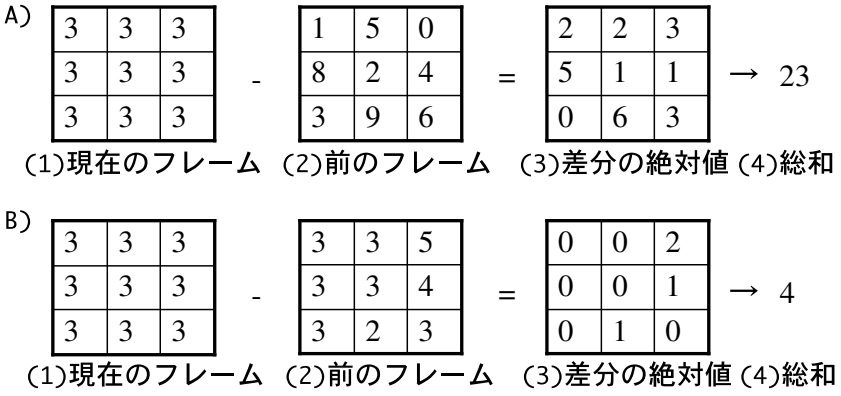

図 7: ピクセルブロックの差分

関数 dist1 はこの両ピクセルブロック中の対応するピクセル 1 つずつについて、その差の絶対値を取り、更にその総和を取ることでこれを実装している。この値が小さければ小さいほど、前後のフレームから取り出したピクセルブロックは、同じ対象を示している可能性が高いと言える。値が小さかった場合、取り出したピクセルブロックの位置が前後フレームで同じであればフレーム間で移動が無かったということであり、前後フレームで取り出した位置が異なるのであれば、フレーム間でそのように移動があったということを示している。

図 7 は関数 dist1 の実行例である。説明を簡単にするため、ここではピクセルブロックを 3*3 としている。あるフレームから抜き出されたピクセルブロック (1) は、その前のフレームから抜き出されたピクセルブロック (2) と共に関数 dist1 に引数として与えられる。関数 dist1 は与えられたピクセルブロックについて、対応するピクセル毎に比較を行い、その差分の絶対値を得る (3)。例えばピクセルブロックの左上のピクセルを比較する場合、A) のようなピクセルブロックが与えられたとすると (1) の左上のピクセルの値である 3 と、(2) のやはり左上のピクセルの値である 1 から、(3) の左上にその差分の絶対値は 2 が返される。全てのピクセルについてこの比較が終わると、(3) の全ての値について和を取り、これを返り値として返す。この例の場合、(4) に示すように 23 が返り値になる。B) のようなピクセルブロックが与えられた場合も同様の演算を行い、実行結果として 4 が得られる。A) と B) において (1) のピクセルブロックは同一で、(4) の値は B) の方が小さい。よってこの場合、A) の (2) を切り出した部分よりも B) の (2) を切り出した部分の方が、(1) の移動元の候補と言える。

移動予測ではあるピクセルブロックが前フレームでどの位置に存在するかを

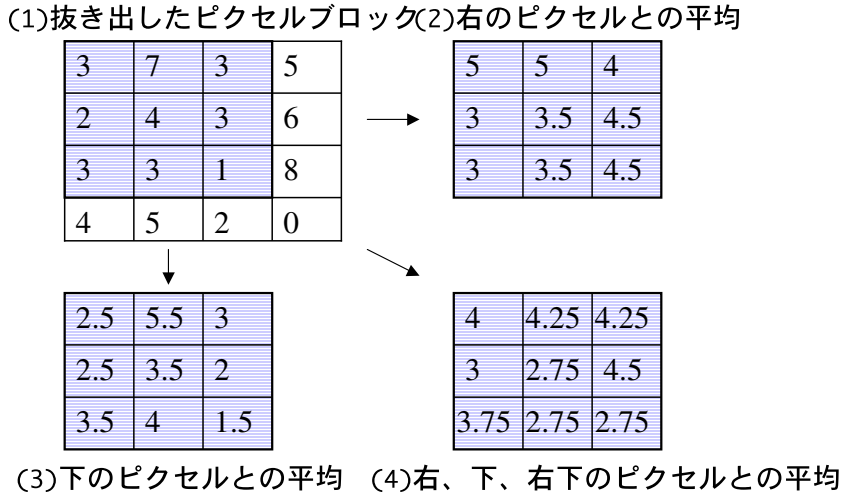

図 8: 半ピクセルの移動検出

調べるため、前フレームから幾つものピクセルブロックを切り出し、その中から関数 dist1 で最も小さい値を返すものを探し出す必要がある。一方、あるピクセルブロックについてその移動予測先となり得るのは次フレーム内のピクセルブロックの中で高々一つである。ということは、これまでに比較した中で最も関数 dist1 が小さな値を返した次フレームのピクセルブロックより、関数 dist1 で大きい値を返すようなピクセルブロックは移動元の候補となり得ない。そこで関数 dist1 は移動予測を調査しているピクセルブロックについて、これまでの関数 dist1 での最良の結果を記録して、関数 dist1 実行中にそれより大きな値となった場合は実行を終了する仕組みを備えている。

ここまでの検出は全てピクセル単位の演算である。しかしもしピクセル単位未満の移動がフレーム間で存在した場合は見逃してしまう可能性がある。そのため、MPEG では精度を上げられるよう、ここまでに得られた最も良い移動元の候補の周辺で半ピクセル単位の移動検出を行い、更に良い結果が得られないかを調べる。この場合、関数 dist1 は与えられた引数により、移動予測に用いる現在のピクセルブロックの値を近傍のピクセルとの平均値に置き換える。

図 8 はピクセルブロックを 3*3 とした時の、この動作の具体例である。これまでに述べた通り、関数 dist1 は通常与えられたピクセルブロックの値そのものを比較対象に用いる。ここでは図中 (1) の網かけ部分がこれに当たる。しかし半ピクセル横にずれたような移動検出を行う場合は、(2) のようにそれぞれのピクセルが自身のピクセルの値と右のピクセルとの平均値を取り、これを比較

対象に用いる。同様に半ピクセル縦にずれたような移動検出を行う場合は (3) のように下のピクセルとの平均値を、半ピクセル斜めにずれたような移動検出を行う場合は (4) のように右、下、右下のピクセルとの平均値を用いる。即ち dist1 は、それぞれ縦と横に半ピクセル移動の検出を行うか否かという、4 つの動作パターンを持っている。これらの動作パターンは関数 dist1 に与えられる引数によって決まり、一度の呼び出しに対して複数の動作パターンを行うことは無い。

以上のような特徴を持つ関数 dist1 について、再利用による高速化を計るため、プログラムに下に示すような変更を加えた。

4.1.1 関数の分割

前述の通り関数 dist1 は四つの動作パターンを持ち、引数によってどれか一つを実行している。そしてこれらの動作パターンは、それぞれによって参照するピクセルの数が異なる。例えば 3*3 のピクセルブロックを用いるとすると、図 8 にある通り、ピクセルブロックの値そのものを用いる場合は 9 個のピクセルを参照するだけになる。一方、右や下との平均を取る場合は 12 個、右・下・右下との平均を取る場合は 16 個のピクセルを参照する必要がある。

このため、ある動作パターンで再利用表に記録した入力と出力の組を他の動作パターンで再利用することはあまり効率的では無いと考えられる。例えば 2 値の差の絶対値を取る部分について再利用を行う場合では、1 ピクセル毎の比較ではピクセルの値そのまま、つまり移動予測に用いるピクセル Y 値、0 から 255 の整数が入力になると考えられる。しかし下のピクセルとの平均値を取る場合、入力に小数点以下の数字が入る場合が現れる。そしてこうした入力を再利用のために記録したとしても、1 ピクセル毎の比較では用いられないため、結果的に再利用可能性の低い RB のエントリを増やすことになる。

また四つの動作パターンのうち対応する 1 ピクセル毎を比較する動作パターンは、関数 dist1 呼び出しの 9 割を占めることが、エンコーディングを実際に行った結果分かった。先の結果と合わせると、この動作パターンがエンコーディング時間全体の 4 割以上を占めているということになり、この部分の高速化がエンコーディングの高速化の前提となることが分かる。

以上のような理由から、関数 dist1 から 1 ピクセル毎を比較する動作パターン部分だけを抜き出し、別の関数 dist1a として分割した。これにより、四つの動作パターンが混在して再利用表のエントリを占めるという無駄を防ぎ、最も

```
int \texttt{dist1a}(*p1,*p2,distlim)
  *p1 : 現在のフレームから切り出したピクセルブロック;
  *p2 : 一つ前のフレームから切り出したピクセルブロック;
  d : これまでで最良の移動予測の値;
{
  s=0;
  for (j=0;j<ピクセルブロックの縦の長さ;j++){
    v=p1 と p2 の j 行 0 列目の値の差;    v の絶対値を s に加算。
    v=p1 と p2 の j 行 1 列目の値の差;    v の絶対値を s に加算。
       …
    v=p1 と p2 の j 行 15 列目の値の差;   v の絶対値を s に加算。
    v=p1 と p2 の j 行 16 列目の値の差;   v の絶対値を s に加算。
    s が d よりも大きくなったら中断。
  }
}
```

図 9: 関数 dist1a

頻繁に呼び出される動作パターンに限って独自のエントリ表を確保することが可能になる。これにより再利用率が向上することも予想される。分割した関数 dist1a の擬似コードを図 9 に示す。

以下に続く手法は、ここに述べた関数 dist1 の分割を行った上で、その分割した関数 dist1a に限って更なる高速化を行うための提案である。これらの手法を関数 dist1a に限って適用するのは、その他の動作パターンについては呼び出し回数が少なく、再利用によって高速化するよりも適用による各種オーバーヘッドの方が大きくなるか、あるいは高速化しても僅かであると予想されるからである。

4.1.2 ピクセルずつの比較

関数 dist1 の再利用率を高めるためには、再利用が期待出来る区間を関数に切り出すという手法が考えられる。図 9 の示す通り、関数 dist1a は大まかに言って一重ループの構造であり、ループ内で二つのピクセルブロックの j 行 n 列目の値の差を取り、その絶対値を加算するというのを 16 回繰り返していた。そこでループ内のピクセル比較を関数として切り出せば、再利用率の上昇が期待出来る。ただし再利用機構が再利用可能な区間と認識する関数は、引数が 6 つまででである。よって、一連のピクセル比較を関数化する場合、一度に 3 つのピクセルずつを比較する関数までが可能であると考えられる。関数として比較するピクセルの数を増やすと入力が多様化するので再利用率は低下する。その一

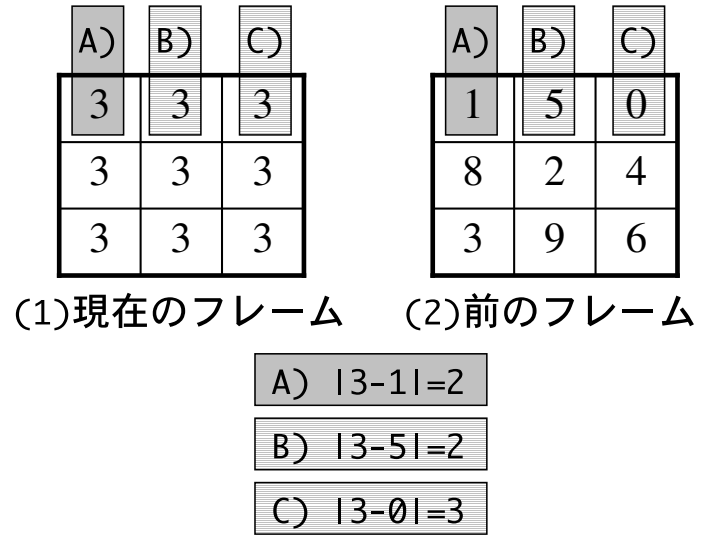

図 10: 関数 pix1

方、再利用が可能であった場合は一度で多くの演算が省略出来るので効果が大きい。

以上のように、一度の呼び出しで比較するピクセル数の大小は一概にはどれが良いかを予想出来ないので、ここでは 1 ピクセルずつを比較する関数 pix1、2 ピクセルずつを比較する関数 pix2、3 ピクセルずつを比較する関数 pix3 の三種類を用意し、それぞれの評価を行うことにした。

関数 pix1 を用いた実行例を図 10 に示す。説明を簡単にするため、ピクセルブロックを 3*3 としている。(1) と (2) のようなデータが関数 dist1a に渡されたとすると、MSP は A)1 ピクセル目の演算を行いつつ、B)2 ピクセル目のデータや C)3 ピクセル目のデータを SSP へ投入する。この結果、MSP が A) の演算を終える頃、SSP は B) や C) の演算を終えてその結果を再利用表に登録しておくことが出来る。MSP はその結果を用い、自身は実際に演算を行うことなく B) や C) の演算結果を得られる。また、ここで比較するアドレスは常に同じなので、以後も A) で演算を行った 3 と 1、B で演算を行った 3 と 5 というような入力があった場合は、実際に演算を行うことなく結果を得ることが出来る。

4.1.3 ピクセルデータの詰め込み

前項では 3 ピクセルずつの比較までを考えた。もし、一度の関数呼び出しでより多くのピクセルの比較が可能となれば、再利用率は低下するものの、一度の再利用で削減されるステップ数が大きくなるため、高速化する可能性がある。

図 11: 関数 pointer

そこでより多くのピクセルを一度の関数呼び出しで比較する手法を考える。

現在、関数 pix1・pix2・pix3 の引数となっているピクセル値は YUV 変換で得た Y 値であり、与えられる値は 0〜255 の範囲の整数値である。一方、再利用機構は 32 ビットまでの値を引数として認識する。そこで各 2 ビットのピクセル値 4 つを結合して 8 ビットとし、これを関数に与え、関数内部で結合した値を分割した後、それぞれを比較するという手法を考えた。関数 pix1・pix2・pix3 にこの手法を適用したものを関数 pix1-4・pix2-4・pix3-4 とする。これにより、一度の関数呼び出しで比較出来るピクセル数はそれぞれ 4、8、12 となる。

4.1.4 ポインタを用いた行毎の比較

ピクセル毎の比較では関数呼び出しの回数が多いため、オーバーヘッドが大きくなると予想される。また、これまでに読み出したピクセル値から次のピクセル値を予想するのは困難であるため、事前実行による高速化が期待出来ない。そこでこの 16 回の一行分の比較をそのまま切り出し、関数 pointer とする。画像データはポインタを用いた配列の形で与えられているので、切り出した関数 pointer には参照する一行分のデータの先頭のアドレスを引数として渡す。関数 pointer は与えられた一行分のデータについてそれぞれのピクセルの差分の和を返すので、関数 dist1a はポインタを操作し次の一行のデータを再び関数 pointer に与える、というのを繰り返す。ある行のデータの先頭アドレスが関数 pointer に渡された時、次に引数として渡される一つ下の行のデータの先頭アドレスは簡単に予測されるので、事前実行によって高速化が期待される。

関数 pointer を用いた実行例を図 11 に示す。説明を簡単にするため、ピクセルブロックを 3*3 としている。これまでは 1 ピクセル毎に比較を行っていたと

ころを、行全てのデータをポインタで渡している。(1) と (2) のようなデータが関数 dist1a に渡されたとすると、MSP は A)1 行目の演算を行いつつ、B)2 行目のデータや C)3 行目のデータを SSP へ投入する。この結果、MSP が A) の演算を終える頃、SSP は B) や C) の演算を終えてその結果を再利用表に登録しておくことが出来る。MSP はその結果を用い、自身は実際に演算を行うことなく B) や C) の演算結果を得られる。

ただし再利用機構では参照するアドレスが異なる場合、その値が同じであっても再利用を行うことは出来ない。よってポインタ操作により参照するアドレスが頻繁に変更されるこの仕組みでは、事前実行で再利用表に登録した以上の再利用を行うことは困難であると予想される。またポインタ操作が存在しなかったとしても、2 ブロックの各 16 ピクセルの値が関数によって参照されるため、これらの値が完全に一致しない限り再利用は行われず、その確率はとても低いと考えられる。ただし一度の再利用で多くの演算が省略出来るので、その効果は大きい。

4.1.5 曖昧再利用

曖昧再利用とは再利用時の入力一致条件に寛容性を持たせることで、既に記録した入力と出力の組に対してそれに近しい入力が与えられたような時、実際に命令を実行することなく、与えられたものに近い入力に対応した出力を返すという手法である。

この手法によって得られた出力はもちろん、ほとんどの場合において完全に正確な値ではない。しかし入力と出力に何らかの相関性があるような場合や入力の僅かな差が出力に影響を及ぼさないような場合では、出力の品質を大きく損うことなく再利用率を上げることが可能になり、結果として処理を高速化することが出来る。

ここではエンコーディング時の移動予測部全体について、フレーム内のピクセルデータの下位ビットを 2 ビット分マスクしたものと 5 ビット分マスクしたもの、2 種類の曖昧再利用を用いてそれぞれ評価を行うことにした。図 12 は 1 ビットから 3 ビットまでの曖昧化を行った場合のデータの変化である。曖昧化するビット数を増やすほど、値の種類が減っていることが分かる。

再利用機構側から見ると、曖昧化により入力の多様化が抑えられるので、再利用率が高まることが期待される。特に入力値が複数となるような関数やループの場合で大きな効果が期待出来る。

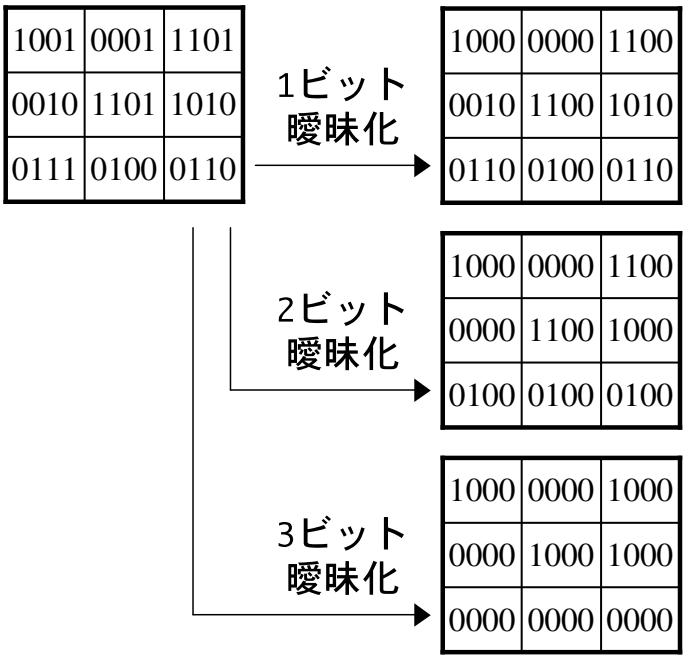

図 12: 曖昧化

```
y = (32*(x>=0 ? x : -x) + (d>>1))/d;
d = (3*m+2)>>2;
y = (y+d)/(2*m);
```

図 13: 量子化の演算

一方、曖昧化によって移動予測の正確さは失われるので、作成される動画の圧縮率には影響を及ぼす可能性がある。ただし、移動予測の項でも述べた通り画質には影響を及ぼさない。

4.2 量子化の高速化

mpeg2encode において、量子化は関数 quant によって実装されている。quant は DCT によって周波数成分となった 8*8 のピクセルブロックの値それぞれに対して、事前に決められた対応する周波数成分の量子化マトリックスと量子化係数の値で演算を行い、端数を切り捨てるという関数である。x を周波数成分、d を量子化マトリックスの値、m を量子化係数とした時、求める値 y を導く演算は図 13 のようなものになる。

演算の結果もし y が 255 よりも大きくなった場合、関数 quant は y を 255 に丸める。また入力時点で x が負値であった場合は、y も符号を変えて負値とする。

関数 quant では一度の関数呼び出しに対して全てのピクセルの値について演算を行うので、入力だけが異なる同じ演算が 64 回行われることになる。そこでこの 1 回の演算を関数として切り出して、再利用可能な区分となるようにした。

第 5 章　性能評価

本章では前章で述べた各手法を用いて、実際の処理速度が向上したかを評価する。評価には再利用機構を搭載した、単命令発行の SPARC-V8 シミュレータを用いた。各パラメータは表 1 の通り。なお、キャッシュ構成や命令レイテンシは SPARC64-III[7] を参考にしている。RF エントリ数-RB エントリ数-RB の読み書きアドレス数については、それぞれ 32-128-4096、32-256-2048、32-512-1024、16-1024-1024 について実際にシミュレーションを行い、32-512-1024 が最も再利用率が高かったことを予備調査において確認している。

MPEG のエンコーディングには mpeg2encode を利用し、オリジナルのソース、および前章に挙げた手法を適用したものをそれぞれ gcc-3.0.2 (-msupersparc -O2) によってコンパイルを行い、スタティックリンクにより生成したロードモジュールを用いた。

5.1　ブロック差分関数の評価

オリジナルのソース、関数の分割、およびポインタを用いた行毎の比較、1 ピクセルずつの比較を関数化、2 ピクセルずつの比較を関数化、3 ピクセルずつの比較を関数化について、それぞれ再利用なし、再利用あり、2 ビットの曖昧再利用、5 ビットの曖昧再利用を用いた場合の全体の実行サイクル数を計測した。オリジナルのソースで再利用を行わなかったものでの全体の実行サイクル数を 1 とした場合の、それぞれの実行サイクル数を図 14 に示す。同様に、オリジナルのソースで再利用を行わなかったものでのブロック差分関数の実行サイクル数を 1 とした場合の、それぞれの実行サイクル数を図 15 に示す。

図 14 および図 15 を見ると、まずオリジナルでも 5 ビットの曖昧再利用では高速化していることが分かる。これはピクセルデータを曖昧化して切り下げた結果、関数 dist1 が途中で終了する条件となる「これまでに得られた関数 dist1 での最良の値」も切り下げられ、関数 dist1 は途中で終了することが増えるからである。これは例えば全てのピクセルの値が 3 であるピクセルブロックと、全

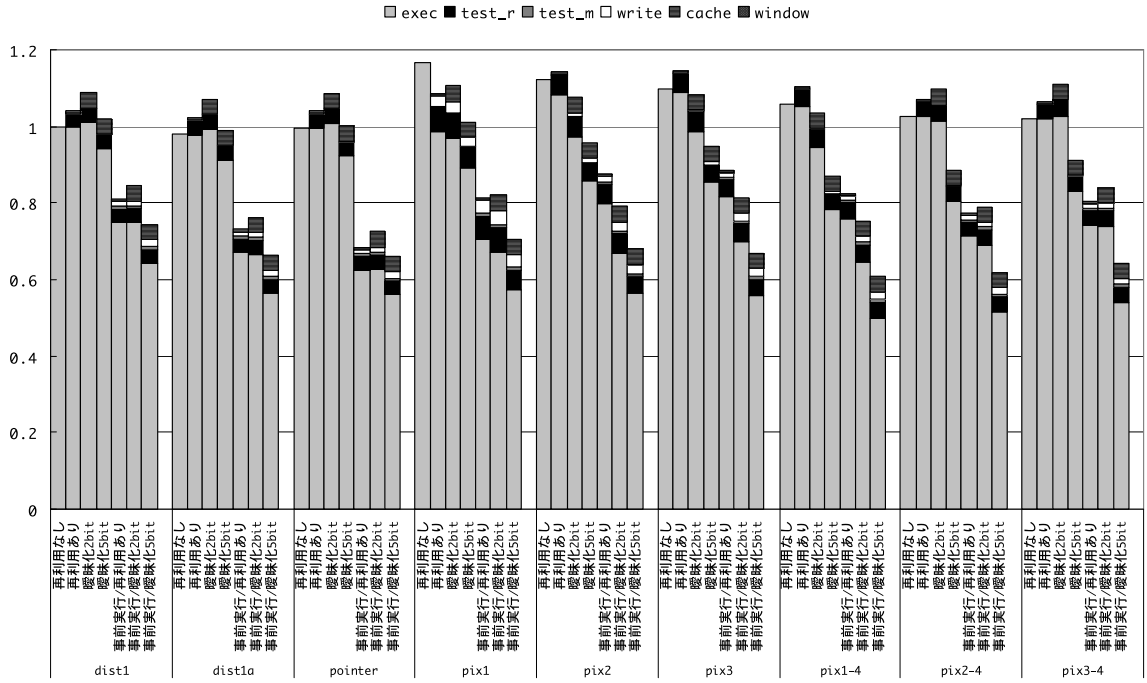

図 14: プログラム全体から見た再利用評価

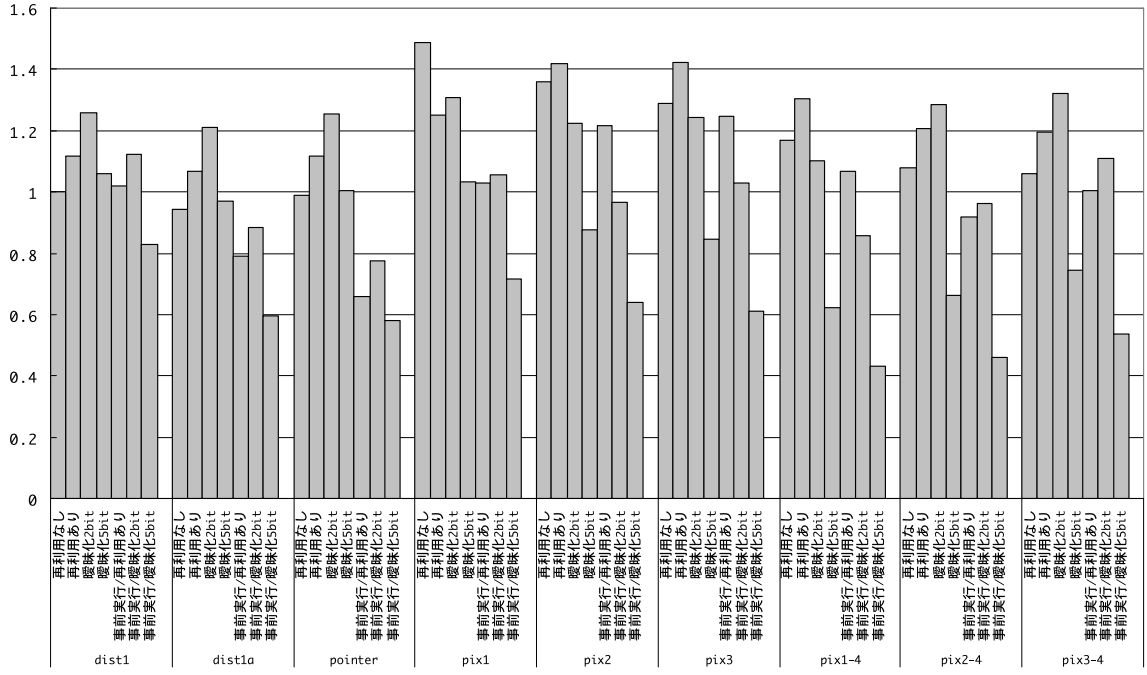

図 15: ブロック差分関数の再利用評価

D-Cache 容量	64KBytes
ラインサイズ	64Bytes
ウェイ数	4
Cache ミスペナルティ	20cycles
Register Window	4sets
Window ミスペナルティ	20cycles/set
ロードレイテンシ	2cycles
整数乗算レイテンシ	8cycles
整数除算レイテンシ	70cycles
浮動小数点乗算レイテンシ	4cycles
単精度浮動小数点除算レイテンシ	16cycles
倍精度浮動小数点除算レイテンシ	19cycles
RF エントリ数	32
RB エントリ数	512/RF
Read アドレス	1024/RF
Write アドレス	1024/RF
RB(引数) ⇔ Register 比較	8words/cycle
RB(Read) ⇒ Cache 比較	4Bytes/2cycles
RB(Write) ⇒ Cache 書込	4Bytes/cycle
RB(返り値) ⇒ Register 書込	8words/cycle
SSP ローカルメモリ	64KBytes

表 1: シミュレーションのパラメータ

てのピクセルの値が 2 であるピクセルブロックを比較する場合、簡単のためピクセルブロックのサイズを 3*3 とすると、曖昧化を行わない場合は関数 `dist1` の結果は 9 であり、以後は 9 よりも小さな値を返すピクセルブロックを探すことになる。一方 1 ビットの曖昧化を行うと、それぞれのピクセルの値が 3 だったピクセルブロックは、全て 2 になる。結果、関数 `dist1` の返り値は 0 となり、これよりも良い値を返すピクセルブロックは存在しないため、以降関数 `dist1` は呼び出されるとすぐに終了する。このようにして、曖昧化により全体の計算量は減ることになる。一方 2 ビットの曖昧再利用ではこうした影響は軽微である上、曖昧化の演算が増えているため遅くなっている。

続いてそれぞれの再利用手法を見ると、分割を行った関数 `dist1a` はサイクル数の削減に明らかに貢献していることが分かる。再利用ありでオリジナルと比較すると、約 10%ほど高速化している。ただし曖昧化による高速化はオリジナルで見られるものと大差無く、曖昧化によって再利用率が大きく上昇したわけではないことが分かる。

ピクセルずつの比較を関数化した場合は、`pix1`、`pix2`、`pix3` の全てについ

て、再利用を行った場合の総サイクル数がオリジナルより劣っている。これは関数呼び出しのオーバーヘッドなどにより、そもそも再利用なしでの総サイクル数が大きく増えているからである。ただし、特に 1 ピクセルずつの比較についてレジスタ読み込みが増加しており、再利用を行おうとする機会は増えていることは分かる。このことはピクセル比較の関数化全てについて、曖昧化により大きくサイクル数を削減していることからも分かる。曖昧化を行わない場合は入力の多様性が一番低い pix1 が最も高速である。その一方で曖昧化 5 ビットでは、入力数が多く多様である pix3 が逆に最も高速になる。また、事前実行を行わなかった場合は再利用率が低下しているものの、他の手法に比べるとその度合は少ない。

また、1 引数に 4 つのピクセル値を詰め込んだ pix1-4、pix2-4、pix3-4 は、それぞれ pix1、pix2、pix3 より高速化している。再利用なしでも高速化しているのは、一度の関数呼び出しで比較するピクセル数が増えた結果、関数呼び出しの回数が減り、オーバーヘッドが削減されたからである。そして pix1-4 は全ての手法の中でも最も高速であった。pix2-4 や pix3-4 は一度の関数呼び出しで 8、12 のピクセル値を参照することになり、入力の多様性が非常に高く、再利用率が上がらないことが原因と予想される。事前実行を行わなかった場合では、比較するピクセル数が多いものほど更に遅くなっている。pix3-4 などでは再利用時に、事前実行で得た結果を用いる場合が多いということが分かった。

一方、ポインタを用いた pointer では dist1a より高速化している。ただし、dist1a と同様に曖昧化による高速化は大きくなく、やはり曖昧化の処理のため、2 ビットの曖昧化などではむしろ遅くなっていることが分かる。また、事前実行を用いることで著しい高速化が期待されたものの、残念ながら他の手法と比較して圧倒的に速くはなっていない。これは理想的な移動予測先を発見した後については、関数 dist1a が呼び出される度、関数 pointer1 によって一行分の比較を行った後、既に得た結果より悪いと見て事前実行で得た値を用いることなく、関数 dist1a が終了するからであることが分かった。一方、事前実行を用いない場合は、実行中に参照するアドレスが変化するため、予想通り全く高速化していない。

なおこのエンコーディングを通して、生成される MPEG ファイルサイズは曖昧化 3 ビット程度まではオリジナルとあまり変わらない圧縮率になっていることが分かった。しかし曖昧化 4 ビットでは圧縮率が若干ながら低下し、5 ビット

	再利用なし	再利用あり
オリジナル	1	0.81
関数切り出し	1.01	0.78

表 2: 量子化関数の再利用評価

ではオリジナルに比べると 1%程度大きなファイルサイズになっている。ただし入力となる画像セットや枚数により、この値は変化する可能性がある。

5.2 量子化関数の評価

量子化関数の関数切り出しについて、全体のサイクル数を計算した。オリジナルのソースで再利用を行わなかった場合を 1 とした場合の、実行サイクル数を表 2 に示す。関数切り出しによりオーバーヘッドが発生しているものの、再利用によりオリジナルより全体の約 3%高速化していることが分かる。量子化がエンコーディング時間全体の約 7%を占めていたことを考えると、この手法を用いることにより量子化部分は約 25%高速化したことが分かる。

第 6 章 まとめ

本研究では MPEG エンコーダにおける、並列事前実行を用いた高速化手法を提案した。エンコーディング時間の分析から移動予測におけるブロック差分の測定、および量子化が高速化の可能性があると考え、関数の分割や切り出しの方法を示し評価を行った。

結果、ブロック差分の測定については 4 ピクセルの比較を一度の関数呼び出しで行うようプログラムを書き換えた上、曖昧再利用を併用することで、オリジナルと比較して全体の約 40%、ブロック差分関数部に限って見ると約 59%のサイクル数を削減することが出来た。ピクセル比較の関数化はポインタによる比較の関数化に対し、オーバーヘッドが大きく生じるという問題があった。しかし 1 つの引数に 4 つのピクセル値を詰め込むことでオーバーヘッドを抑える一方、曖昧再利用を用いることで入力の多様化を防ぎ、最終的にはピクセル比較の関数化がポインタによる比較の関数化より高速となった。この手法は事前実行無しでもブロック差分関数部の約 40%のサイクル数を削減しており、事前実行を用いてポインタにより行毎の比較を行った場合と同じ程度の高速化が得ら

れた。また曖昧化によるファイルサイズの増大は、軽微であることも見ることが出来た。その一方、量子化については、関数の切り出しにより全体の約 3%、高速化することが可能となった。これらの結果は互いに独立であるので、ブロック差分の比較、量子化のそれぞれにこれまで提案した手法を用いることにより、最大で約 43%のサイクル数を削減することが出来ることが分かった。

今後の課題としてはリアルタイムエンコーディングのための、高速化した場合における処理時間の保証が考えられる。

謝辞

本研究の機会を与えてくださった、富田眞治教授に深く感謝いたします。また本研究に限らず、様々なご指導を頂いた中島康彦助教授、森眞一郎助教授、津邑公暁助手、五島正裕助手に深く感謝いたします。

そして、いつも温かくご教鞭頂いた京都大学情報学研究科富田研究室のみなさんに心より感謝いたします。

参考文献

[1] 中島康彦, 津邑公暁, 五島正裕, 森眞一郎, 富田眞治: 動的命令解析に基づく多重再利用および並列事前実行, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 10(ACS 2), pp. 1–16 (2003).

[2] Álvarez, C., Corbal, J., Salamí, E. and Valero, M.: On the Potential of Tolerant Region Reuse for Multimedia Applications, *Proc. 15th International Conference on Supercomputing*, ACM Press, pp. 218–228 (2001).

[3] 竹村尚大, 津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: MP3 エンコーダの分析及び曖昧再利用の適用による高速化, 情処研報 2003-ARC-152 (HOKKE 2003), pp. 145–150 (2003).

[4] 津邑公暁, 清水雄歩, 中島康彦, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: ステレオ画像処理を用いた曖昧再利用の評価, 情報処理学会論文誌：コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3), pp. 246–256 (2003).

[5] Wang, K. and Franklin, M.: Highly Accurate Data Value Prediction Using Hybrid Predictors, *30th MICRO*, pp. 281–290 (1997).

[6] 津邑公暁, 中島康彦, 五島正裕, 森眞一郎, 富田眞治: 並列事前実行機構にお

ける主記憶値テストの高速化, 情報処理学会論文誌：コンピューティングシステム, Vol. 45, No. SIG 1(ACS 4), pp. 31–42 (2004).

[7] HAL Computer Systems/Fujitsu: *SPARC64-III User's Guide* (1998).