特別研究報告書

マルチメディア命令と区間再利用による音声認識の高速化

指導教員　　富田 眞治 教授

京都大学工学部情報学科

勝野　徳

平成17年2月10日

マルチメディア命令と区間再利用による音声認識の高速化

勝野　徳

内容梗概

近年、計算機の小型化および高速化が進み、動画像や音声などマルチメディアデータを扱う機会が増えている。また、音声認識技術が発達し認識精度が向上することにより，様々な場面において音声認識が普及しつつある。しかし、音声認識には非常に多くの処理時間を必要とし、より効率的な処理による高速化が求められている．本研究では，計算機アーキテクチャの視点から，区間再利用の適用およびマルチメディア命令の利用による高速化の可能性を探った．

区間再利用とは，一連の命令列に対応する入力および出力を記憶しておき，再度同一入力により同一命令区間を実行する際には，記憶しておいた出力を再利用することにより命令区間の実行を省略する高速化手法である。映像や音声データを処理する場合，サブルーチンやループといった命令区間が，類似の値を入力とする計算を繰り返し実行している可能性があるため，区間再利用による高速化が期待できる．音声圧縮を行う MP3 エンコーダに関する既存研究では、入力値の僅かな増減が出力に大きな影響を与えないこと，および，出力の差が聴覚にほとんど影響を与えないことを利用して、入力値の曖昧化による再利用率の向上を図り，実行命令数を最大 25%削減している。一方、マルチメディア命令は、ビット長の長い専用レジスタを用いることにより、1 命令が扱うことのできるデータ量を増やすことで高速化を実現する命令である．Intel 系列のプロセッサでは，MMX，SSE，SSE2，SSE3、AMD 系列のプロセッサでは 3DNow!，SPARC アーキテクチャでは，VIS がそれぞれ対応する．

本研究は、大語彙連続音声認識システム Julius を対象とした。このシステムは、音響モデル、単語辞書、言語モデルを用い、認識結果に至る確率を隠れマルコフモデルに基づいて求めることにより元の音素を特定し、音素の系列から単語や言語を識別する。処理時間を詳細に分析した結果、出力確率計算，すなわち，入力音声データの特徴量と候補となる音素の音響モデルを比較して各音素の出力確率を求める計算の一部が、処理時間全体の 4 割を消費していた。まず，ソースプログラムを一切変更せずに，ハードウェアのみによる関数およびループの再利用および事前実行を適用した結果，約 30 %高速化された。

次に，ソースプログラムを変更し，曖昧化することによる再利用率の向上を試みた。対象とした命令区間は（１）直前の時間フレームに特定した音素から，現フレームの各音素候補へ遷移する確率を求める関数，および（２）出力確率計算により求めた確率値を累積して出力確率を返す関数の２つである。前者については、浮動小数点数である音素を整数インデックスにより関数へ渡している部分を本来の浮動小数点数に戻して渡すことにより，入力パターン数を削減することができると考えた。また、浮動小数点数の有効桁数を削減しても出力結果への影響が小さいと考え、入力値の曖昧化を行った。後者についても同様の変更を行った。この結果、命令区間（１）では，曖昧化前には再利用の効果がないのに対して 50%のサイクル数を削減し、（２）では 14%削減することができた。ただし後者は、入力値を大きさ順に整列することにより入力パターン数を削減し、再利用率を約 1%向上できたものの、整列処理の実行時間は増加したため，全体では 2%の高速化にとどまった。

一方、マルチメディア命令については、出力確率計算内において単精度浮動小数点数の減算、乗算、除算が繰り返されているいくつかの区間に対して命令列を SSE に置き換えることにより高速化を図る。音声データ特徴量、音響モデルの特徴量と共分散行列がそれぞれ同じ長さのベクトルで与えられることを利用し、計算を四つ同時に行うよう書き換えた。その結果、関数別に見ると最大で 38%、全体では 5%の高速化が実現できた。さらに、元のプログラムが高速化のために厳密な計算を行っていない部分について、あえて厳密に計算するよう変更した場合、その部分については SSE 化により 30%高速化でき、全体でも 15%高速化することができた。以上まとめると、厳密な計算を行わない通常実行の場合は、区間再利用およびマルチメディア命令による高速化を合わせて，全体で 7%高速化することができた。

A Speedup Technique for Speech Recognition using Multimedia Instructions Set and Region Reuse

Megumu Katsuno

Abstract

Recently, computers has become smaller in size and faster in speed, so that the multimedia data such as audio or video streams are easily handled. Above all, the speech recognition begin to spread widely due to the improvement in accuracy. However, the processing cost for high-level speech recognition is extremely large, then more efficient framework for processing is required. This paper focuses on several boosting techniques from the view of computer architecture, that is region reuse and multimedia instructions.

Region reuse is one of techniques to speed up. The set of input and output for an instruction region is memorized and reused if the region is encountered with the same set of input. When the image and the speech data are processed, speed-up using region reuse can be expected because there is a possibility that the instruction section such as subroutine and loop executes it repeating the calculation of which it inputs a similar value. In a research about MP3 encoder which compress audio, the number of execution instructions has been reduced by 25% or less by tolerant reuse technique. On the other hand, multimedia instructions is a popular mechanism to process several data stored in wide registers at once.

This paper investigates 'Julius' which is one of speech recognition systems. In this program, the phoneme is identified by calculating each probabilities where each phoneme becomes the candidate corresponding to the voice data exploiting HMM, sound model, word dictionary and language model. As a result of detailed analysis, the region that calculates output probability of each phoneme by comparing input voice data with sound model costs 40% of total execution time. The initial evaluation of original program exploiting region reuse and precomputation shows boosting 30% in speed.

Secondly, I modified the source code to improve the hit ratio of region reuse exploiting tolerant reuse technique. The target sections are following regions, those are (1)the function which calculates the probability of transition from a

phoneme in previous frame to each promising phonemes in present frame, and (2)the function which accumulates the probabilities and returns the result. As for the former, I considered that the number of input patterns could be decreased by using orignal floating point numbers as arguments instead of indirect pointers to the floating point numbers. Moreover, I exploited tolerant reuse degrading the accuracy of input floating point numbers, considering that the unexpected side effect against the output. Also the same technique is applied to the latter. As a result, 50% of cycles were eliminated though there is no effect without tolerant reuse in region(1). Besides 14% of cycles were eliminated. In the latter, although the hit ratio of region reuse could be increased by 1% with the technique that sorts the arguments in order and reduces the number of input patterns, the overhead of the sorting decreases the effect and results in only 2% speed-up against the original program.

Moreover, I rewrote the kernel codes of the output probability calculation using multimedia instruction set what we call SSE. In detail, four sets of floating pont data representing characteristics of speech data, sound model and covariance matrix are packed in 128bit XMM registers. As a result, functions boost maximum 38%, and the overall execution was boosted 5%. Additionally, considering the accurate calculation in original program which degrades the accuracy for speeding up, the section was boosted 30% by SSEs and whole execution was boosted 15%. This paper concludes that overall 7% speed-up is achieved exploiting both tolerant region reuse and multimedia instructions against the original algorism.

マルチメディア命令と区間再利用による音声認識の高速化

目次

第1章　はじめに

近年、計算機の小型化および高速化により、扱いが困難であった画像や音声などのマルチメディアデータを一般的なコンピュータにより取り扱うことが可能となってきた。最近では特に、携帯端末においてこのようなデータを扱うようになってきた。また、様々な場面において手入力のわずらわしさのない音声入力が求められることがあり、そのような場合、音声認識技術が重要になってきている。一方、音声認識の精度が向上し普及が進んでいるものの、音声認識は多くの処理時間を必要とし、高速化が重要な課題となっている。

高速化手法の一つに、マルチメディア命令を用いた命令列の最適化がある。マルチメディア命令とは、ビット長の長いレジスタを用いることにより、通常別々に行う複数の同じ命令を1命令により計算し、高速化を図る命令である。動画像や音声などのマルチメディアデータを高速に処理するために追加された命令であり、Intel社が開発したものとして、8本の64bitレジスタを用いて最大8個の整数演算を同時に実行できるMMXや、8本の128bitレジスタを用いて最大4個の単精度浮動小数点演算を同時に実行できるストリーミングSIMD拡張命令(SSE)などがある。同様に、AMD社が開発した3DNow!でも、整数演算だけでなく、浮動小数点数演算も実行できる。

一方，別の高速化技術として，区間再利用が提案されている。区間再利用とは，一連の命令列を実行する際に命令区間における入力および出力を記憶しておき，再度同一入力により同じ命令区間を実行する際には，記憶しておいた出力の再利用により命令区間の実行を省略する高速化手法である。同様の計算を繰り返し実行するプログラムの場合，命令列の実行そのものを削減でき，大幅な高速化を図ることができる。特に，映像や音声などのデータには，局所的に類似した値が頻繁に出現する性質があるため，大幅な性能向上が期待できる。

音声データを扱うプログラムに対する区間再利用の適用例として，MP3エンコーダに関する報告がある[1]。通常の区間再利用に加えて，入力の一致判別に寛容性を持たせる曖昧再利用が提案されている。再利用区間における入力値に許容範囲を設け，入力のバリエーションを減らすことにより，同じ入力値の出現確率を高め，再利用率を向上させる。通常の実行に比べて区間再利用を適用した場合、サイクル数を最大25%削減している。

本稿では、音声認識システムの構造を分析し、区間再利用技術を利用した高

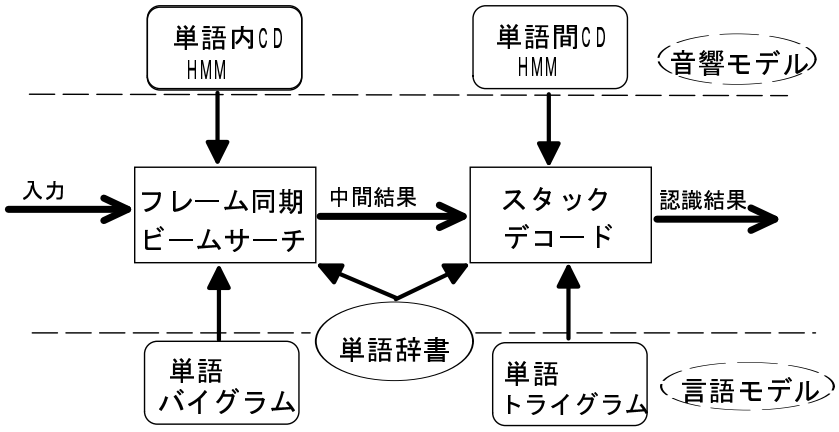

図 1: 認識システムの構成と処理の流れ

速化と、マルチメディア命令を用いた高速化の可能性を考察する。なお、評価にはSPARC アーキテクチャを想定し、マルチメディア命令については、SPARC の VIS 命令よりも高機能な Intel の SSE を仮定して評価を行った。

第 2 章　音声認識システム

本章では、音声認識システムの動作を分析する。その分析をもとに、区間再利用やマルチメディア命令が適用可能な部分について説明する。本研究では、音声認識システムとして、「大語彙連続音声認識システム Julius」を使用した。このシステムは、京都大学 河原研究室、奈良先端科学技術大学院大学音情報処理学講座で開発されており、オープンソースソフトウェアとして C 言語のソースが完全公開されている。音声認識の実現には，主に音声分析して特徴量を取り出す音声分析、特徴から音韻性を識別する尤度計算、音素の並びを考慮して言語的に正しく出力する言語探索の 3 つの処理から成る。Julius の構成と処理の流れについて図 1 に示す。Julius では、音響モデル、単語辞書、言語モデルを用いる。

2.88 秒の音声データ (以下「データ 1」とする)、15 秒の音声データ (以下「データ 2」とする) を入力したときの実行時間の割合を表 1 に示す。各時間は、times システムコールを用いて測定した。これらの時間とは別に、Julius の起動時にモデルファイルの読み込みに約 800ms かかっているものの、以降の評価では考慮しない。

表 1: ファイル入力から実行終了まで

実行内容	data1	data2
データ格納・解析	1.5	1.3
第 1 パス	95.4	74.5
第 2 パス	3.1	24.2
Total	100.0	100.0

表 2: 第 1 パスの詳細

実行内容	data1	data2
初期化	1.8	0.9
ビタビ計算	17.6	15.4
出力確率計算	76.5	79.6
ビームサーチ	3.9	3.8
終了処理	0.2	0.3
Total	100.0	100.0

以下では、処理時間の大部分を占める第 1 パスについて説明する。

2.1 第 1 パス

Julius は、音声データを 10ms の時間フレームに区切り、フレームごとにデータ解析を行い特徴量ベクトルを求める。この特徴量ベクトルを元に、時間フレームに沿って処理を行う (第 1 パス)。第 1 パスの実行時間の割合を表 2 に示す。表 2 に示す処理は上から順に時間フレーム数だけ繰り返され、時刻 $t-1$ において計算された結果を用いて時刻 t の処理を行う。初期化は、作業領域の初期化や直前フレームの結果の移動を行い、終了処理は、計算結果をもとに暫定結果の出力やエラーのチェックを行う。ビタビ計算、出力確率計算、ビームサーチについては後で詳述する。

以下では、まず音声認識に必要な概念である特徴識別と隠れマルコフモデルについて説明し、次に音声認識に用いられる手法であるビタビアルゴリズムとビームサーチについて説明する。

2.1.1 特徴識別

音声分析結果が x であったとき、話者が w と発したとすると、その事後確率は、以下の式により表現できる。

$$P(w|x) = \frac{P(x|w)P(w)}{P(x)} \tag{1}$$

この値を最大化する w を推定し、その時の値 w_x を識別結果とすることを考える。x はスペクトル特徴を表現するパラメータのベクトル系列となり、数十個

の浮動小数点数により表される。

$$w_x = \mathbf{argmax}_w P(x|w)P(w) \tag{2}$$

確率 $p(x|w)$ を計算するには，あらかじめ w のすべてのカテゴリを含む大量の音声データを収集し，各カテゴリ w ごとに生じた音響特徴 x の確率分布を調べておく。実際の確率分布計算においては大小関係のみを重視し、確率値の対数を取った対数尤度を用いる。

2.1.2 隠れマルコフモデル

精密な音声認識においては、確率モデルを用いた方法がよく用いられる。なかでも、HMM(隠れマルコフモデル) と呼ばれるモデルを用いて認識を行っている。通常のマルコフモデルでは出力シンボル系列から状態遷移系列を特定できるのに対し，出力シンボル系列が与えられても状態遷移系列を特定できないモデルを隠れマルコフモデルと言う。音声認識では出力シンボルが、音声分析された特徴ベクトルの時系列に相当する。

2.1.3 ビタビアルゴリズム

音素や単語に対する特徴量の時系列を $\mathbf{x} = \mathbf{x_1}, \cdots, \mathbf{x_T}$ としたとき，この系列が HMM M から生起する確率は， 以下の式により表現できる。

$$p(\mathbf{x}|M) = \sum_i \alpha(i, T) \tag{3}$$

$p(\mathbf{x}|M)$ について、$\alpha(i, T)$ の総和の代わりに最も確率値の高い状態遷移のみを用いる方法をビタビアルゴリズム と呼ぶ。この場合、$\alpha(i, T)$ は以下のように漸次的に計算される。

$$p(\mathbf{x}|M) = \max_i \alpha(i, T) \tag{4}$$

$$\alpha(i,t) = \begin{cases} \pi_i & (t = 0) \\ \max_j(\alpha(j, t-1)a_{ji})b_i(\mathbf{x_t}) & (t \geq 0) \end{cases} \tag{5}$$

a_{ji} は状態 i から状態 j への遷移確率、$b_i(\mathbf{x_t})$ は状態 i において $\mathbf{x_t}$ が観測される出力確率、π_i は状態 i の初期確率である。なお、実際のプログラムでは高速化のため、対数を用いて以下のように計算される。

$$\log \alpha(i, t) = \max_j(\log \alpha(j, t-1) + \log a_{ji}) + \log b_i(\mathbf{x_t}) \tag{6}$$

表 2 に示したビタビ計算に該当する処理は、$\max_j(\log \alpha(j, t-1) + \log a_{ji})$ を計算する。第 1 パスでの言語探索は、木構造化単語辞書のノードを遷移しながら進められる。各ノードは音素に対応し、直前、直後の音素も考慮した区別がされている。遷移されたノードは、ノード ID とともに、確率値や直前単語番号などを持ち、配列に格納されている。前フレームの結果領域と現フレームの作業領域のため、２つの配列が用意されている。前フレームの全てのノードに対して、単語辞書をもとに全ての遷移先ノードを求め、その全ての遷移先ノードに対し、順に $(\log \alpha(j, t-1) + \log a_{ji})$ を求める。複数のノードから遷移されるノードもあるため、その場合は全ての場合について計算を行い最も値の大きいものを残す。また、自分自身に遷移する場合もある。表 2 の通り、Julius の実行で最も時間がかかっている出力確率計算は、ビタビアルゴリズムの一部であり、前述した $b_i(\mathbf{x_t})$ を求める。前節のビタビ計算で求められた全てのノードに対し、出力確率値を計算する。出力確率計算についての詳細を後述する。

2.1.4 ビームサーチ

ビタビ計算においては、$\alpha(i, t)$ の計算の際にすべての状態の組み合わせを考慮せず，選ばれる可能性の少ない状態の組み合わせを無視する。具体的には，各時刻 t において $\alpha(i, t)$ の値の大きなもののみを記録しておき，次の時刻では後続可能な状態のみについて確率計算を行う。これをビームサーチと呼ぶ。言語モデルと組み合わせる連続音声認識では，HMM の連結により状態遷移の組み合わせが増大するため必須の技術となる。Julius では、出力確率計算までに求められる確率値は時間フレームごとに平均して 1800 近くある。このうち次の時間フレームにおいて用いる結果を確率値の高いものから 600 個だけ残しておき、後は無視する。

2.2 Julius の関数構造

データ 2 を入力した場合の各関数の所要時間と全体における割合、および、内側の関数の実行時間も加えた総実行時間を表 3 に示す。時間は gprof を用いて測定する。

Julius の関数の内、関数 compute_g_beam_pruning の処理に最も時間がかかっている。本関数は、outprob_state → calc_tied_mix → gprune_beam という階層構造で呼ばれ、第 1 パスおよび第 2 パスで確率計算の核となる計算を行う。出力確率計算の中核部分は関数 gprune_beam において処理される。関数 gprune_beam

表 3: 関数内の処理時間

関数名	時間 (秒)	割合 (%)	総時間 (秒)
compute_g_beam_pruning	4.51	40.7	4.51
get_back_trellis_proceed	1.03	9.3	6.24
outprob_state	0.91	8.2	7.16
addlog_array	0.61	5.5	0.61
outprob_style	0.54	4.9	4.57
next_word	0.47	4.2	1.39
calc_tied_mix	0.42	3.8	6.21
compute_g_beam_updating	0.29	2.6	0.29
...	...	...	...
gprune_beam	0.09	0.8	5.17
max_successor_prob	0.05	0.5	0.05
...	...	...	...
Total	11.10	100.0	—

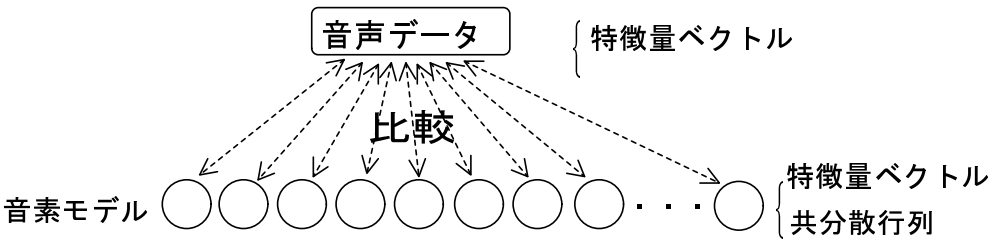

図 2: 音声データとモデル

の動作の概要を図 2 に示す。出力確率計算においては、音声データより抽出した時刻 t における特徴量ベクトルと、ビタビ計算で求められたノード番号が入力として与えられる。一つのノードに対して複数の音素モデルが存在し、各音素モデルは、音声データの特徴量ベクトルに対応するベクトルと共分散行列をもつ。各音素モデルに対して音声データと比較を行い確率を求める。本研究では、音素モデルの数を 64、特徴量ベクトルのサイズを 25 とした。比較を行う計算式は次のとおりである。(Const：定数)

$$(\sum_{\mathrm{i}=0}^{24} S_{\mathrm{i}} + \mathrm{Const})/(-2) \tag{7}$$

$$S_{\mathrm{i}} = (\mathrm{vec[i]} - \mathrm{mean[i]})^2/\mathrm{var[i]} \tag{8}$$

ここで、vec を音声データの特徴ベクトル、mean をそれに対応するモデルの値、var を共分散行列とする。以上の式を全ての音素モデルに対して求め、求められた 64 個の値の内、小さいものを 2 個加算したものが、入力に与えられた音声データとノードより求められた確率値となる。

しかし、実際に全ての計算を行うと非常に多くの時間がかかってしまうため、Julius では計算時間短縮のために以下のような工夫がなされている。まず、モデルの内一つを選び以下の計算をして長さ 25 の配列を求める。ただし、$0 \leq j \leq 24$

$$th[j] = \sum_{i=0}^{j} S_{\mathrm{i}} \tag{9}$$

さらに、モデルから別の一つを選び以下の計算を行い配列の値を更新する。

$$th[j] = \max(th[j], \sum_{i=0}^{j} S_{\mathrm{i}}) \tag{10}$$

この 2 式を表 3 内の関数 compute_g_beam_updating により行う。この時、式 (7) の計算も行っている。次に、関数 set_dimthres により閾値 $th[j] = th[j] + Const$ を求める。その後、関数 compute_g_beam_pruning により、残りのモデル 62 個に対して式 (7) を求める。ただし、$i = 0$ から $i = j$ までの和が、$th[j]$ を越えた時点で計算を終え $\log 0$ を返す。compute_g_beam_updating によりあらかじめ 2 つのモデルに対し式 (7) を求めているため、compute_g_beam_pruning が全て $\log 0$ を返すとしても、少なくとも二つの確率値が求められている。

以上のようにして、計算量を大幅に削減することができる。しかし、厳密な計算ではなくなる場合がある。たとえば、i が小さい時は急増するが i が大きくなるとほとんど増えない場合である。この厳密性を保つために定数 $Const$ が用いられており、多少急増しても処理を中断しないようになっている。そのため、この定数の設定値が重要になる。

第 3 章　高速化技術

本研究が用いた高速化技術について説明する。なお、前述したように、評価には SPARC アーキテクチャを想定した。ただし、マルチメディア命令については

SPARC の VIS(Visual Instraction Set) よりも高機能な Intel の SSE(Streaming SIMD Extensions) 命令を仮定して評価を行った。

3.1 区間再利用

SPARC ABI に基づいて記述されたプログラムに対して、まずは関数単位の再利用を適用する機構について説明する。本機構は、関数の入出力情報を再利用表を用いて記憶し、同一関数が同一入力値により呼び出された場合に関数の処理を省略する。このため、関数の入力や出力が何であるかの特定を要する。一般のプログラムにおいては、関数内で関数が呼ばれるなど、多重構造を形成することがある。関数 A が関数 B を呼ぶ場合、A から B への引数は B への入力、返り値は B からの出力になる。A の局所変数は、A の入出力ではないものの、ポインタを通じて B の入出力になり得る。B の局所変数は、A、B いずれのの入出力にもならない。また、大域変数は、A、B のどちらの入出力にもなり得る。以下に、再利用表の構成と関数再利用機構の動作について述べる。

3.1.1 再利用表の構成

再利用表とは、関数の入力および出力の情報を記憶しておく表である。再利用ウインドウ (RW)、関数管理表 (RF)、入出力記憶表 (RB) から構成される。

再利用ウィンドウ (RW) は、入れ子の関数を再利用するために、どの関数がどの関数を呼び出したかを保持する。具体的には、現在実行中かつ登録中の RF および RB の各エントリをスタック構造として保持する。この様子を図 3 に示す。なお、RW のエントリ数は有限であるものの、一度に登録可能な多重度を越えて関数が呼び出された際には、外側の関数から順次登録を中止し、より内側の関数を RW エントリに加えることにより、多重構造に追随する。また、ある関数の実行および登録中に、再利用可能な関数に遭遇した場合は、登録済みの入出力をそのまま登録中エントリに追加することにより、RW の深さを越える多重再利用を可能とする。

関数管理表 (RF) は、RB に登録されている関数のアドレス、RB エントリに共通する主記憶読み出しアドレスおよび書き込みアドレスを保持する。図 4 に１つの関数を再利用するために必要なハードウェア構成を示す（関数再利用では、図中の網掛け部分を使用しない）。各エントリ先頭のフラグ V は、各エントリが「登録可能」「登録中」「登録済」のどの状態にあるかを表す。LRU カウンタは、RF エントリの入れ換えのためのヒントとして用いられる。入出力記憶

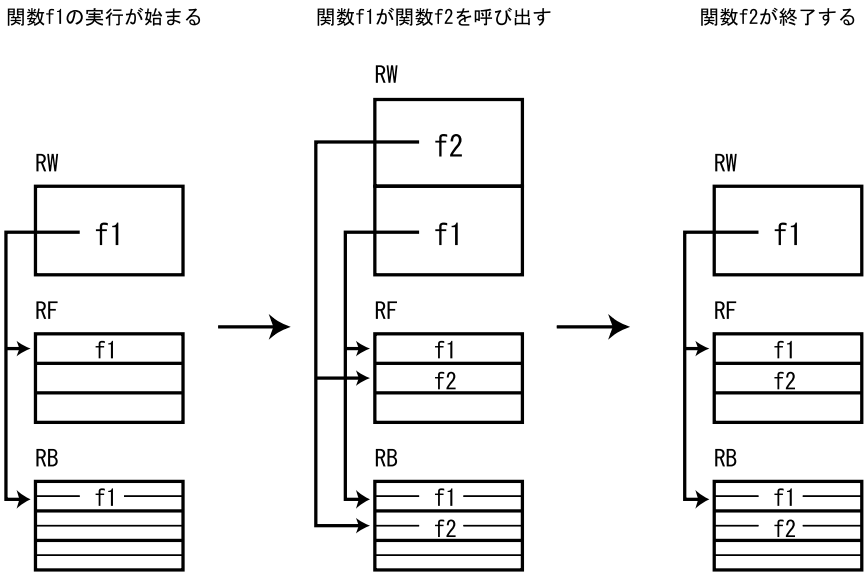

図 3: 再利用ウィンドウ (RW)

表 (RB) は、引数 (V:エントリ状況、Val.:値)、主記憶値 (Mask:読み出しデータ／書き込みデータの有効バイト、Val.:値)、返り値 (V:エントリ状況、Val.:値) および、関数呼び出し直前の%sp の値を保持する。1 つの関数あたりの RB エントリ数は一定であり、関数の並び順は RF と一致する。また、主記憶読み出しおよび書き込みデータのアドレスは RF が保持しており、同一関数におけるアドレスは列ごとに同一である。それぞれのデータのサイズは 4 バイトであり、主記憶読み出しデータと書き込みデータの有効バイトを MASK 値によって表す。読み出しアドレスは RF が一括管理し、マスクおよび値は RB が管理するのは、読み出しアドレスの内容と RB の複数エントリを CAM により一度に比較する構成を可能とするためである。RB 各エントリ先頭のフラグ V および LRU カウンタは RF の場合と同じように用いられる。

3.1.2 関数再利用機構の動作

関数を呼び出す call もしくは jmpl 命令が実行されると、再利用機構は関数の先頭アドレスが RF に登録されているかどうかを調査する (1)。登録されていた場合には、その関数に対応する RB から、引数が完全に一致するエントリを選択し (2)、関連する主記憶アドレスすなわち少なくとも 1 つの Mask が有効である読み出しアドレスをすべて参照し (3)、一致比較を行う (4)。すべての入力が一致した場合に、登録済の出力（返り値、大域変数、上位関数の局所変数）を書き戻すことにより (5)、関数の実行を省略することができる。

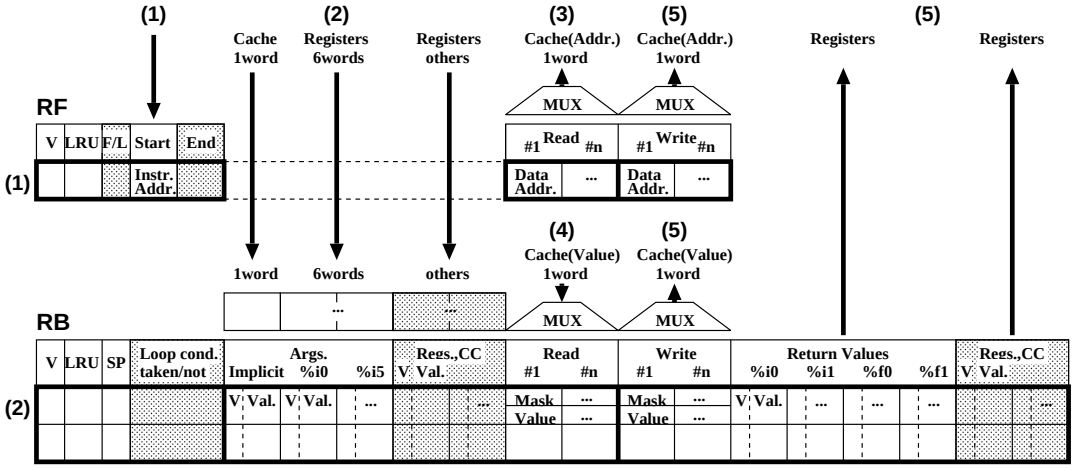

図 4: 関数管理表 (RF)、入出力記憶表 (RB)

RB にすべての入力が一致するエントリが登録されていない場合、引き続いて関数本体の実行を開始する。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していく。読み出しが先行した引数レジスタは関数の入力として、また、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。その他のレジスタ参照は登録する必要がない。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、書き込みは出力として登録する。関数から復帰するまでに次の関数を呼び出したり、登録すべき入出力が入出力表の容量を越えた、引数の第 7 ワードを検出した、あるいは、途中でシステムコールや割り込みが発生したなどの問題が発生した場合、登録を中止する。復帰を示す jmpl 命令によって関数呼び出しが終了すると、RB の状態を「登録中」から「登録済」に変更する。最後にこの RB エントリのインデックスを RW から降ろし、RB への登録を完了する。

また、RF に関数アドレスが登録されていない場合は、RF エントリの登録から行う。RF や RB がすべて埋まっている場合には、LRU アルゴリズムに従って既存のエントリを追い出す。

3.1.3 事前実行

これまでに説明した再利用を行うプロセッサ（Main Stream Processor：以下 MSP と略する）とは別に、命令区間の事前実行により RB エントリを有効化するプロセッサ（Shadow Stream Processor：以下 SSP と略する）を複数個設けることにより、さらなる高速化を図った。事前実行機構の概要を図 5 に示す。RF、RB、主記憶は全プロセッサが共有する。RW、演算器、レジスタ、キャッシュは

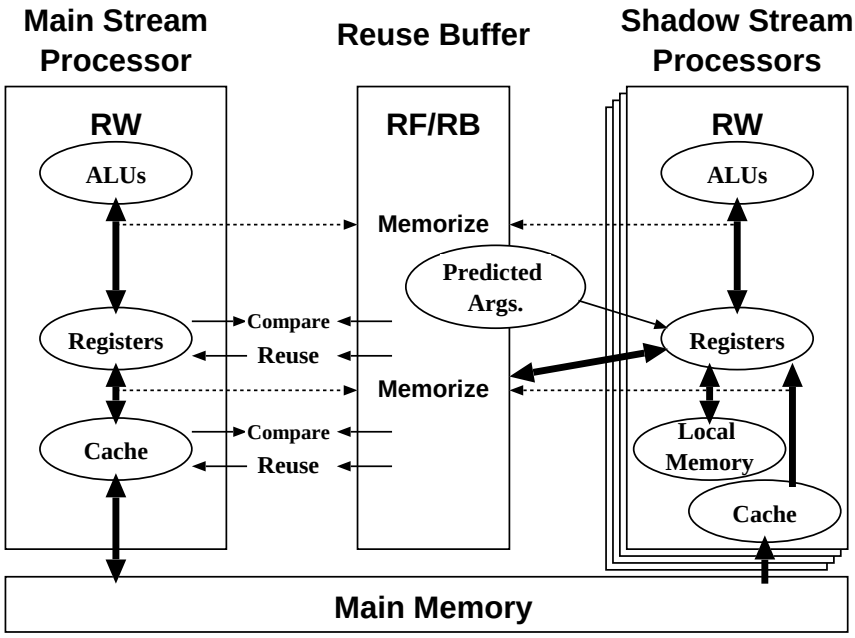

図 5: 並列事前実行機構

各プロセッサごとに独立して設ける。破線は、MSP および SSP が RB に対して入出力を登録するパスを示している。事前実行では、MSP および SSP に対して、いかに主記憶一貫性を保つかが課題となる。特に、予測した入力パラメタに基づいて命令区間を実行する場合、主記憶に書き込む値が MSP と SSP で異なる。これを解決するために、図 5 に示すように、SSP は、RB への登録対象となる主記憶参照には RB、また、その他の局所的な参照には SSP ごとに設けた局所メモリを使用することとし、キャッシュおよび主記憶への書き込みを不要とした。もちろん、MSP が主記憶に対して書き込みを行った場合には、対応する SSP のキャッシュラインが無効化される。具体的には、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 と同様にアドレスおよび値を RB へ登録する。以後、主記憶ではなく RB を参照することにより、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できる。局所的な参照については、読み出しが先行することは、変数を初期化せずに使うことに相当し、値は不定でよいことから、主記憶を参照する必要はない。なお、局所メモリの容量は有限であり、関数フレームの大きさが局所メモリを越えた場合など、実行を継続できない場合は、事前実行を打ち切る。また、事前実行の結果は主記憶に書き込まれないため、事前実行結果を使って、さら

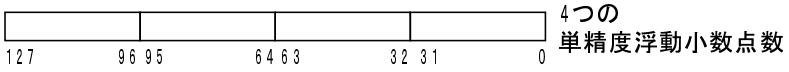

図 6: 128 ビット・パックド単精度浮動小数点データ型

に次の事前実行を行うことはできない。

事前実行に際しては、RB の使用履歴に基づいて将来の入力を予測し、SSP へ渡す必要がある。このために、RF の各エントリごとに小さなプロセッサを設け、MSP や SSP とは独立に入力予測値を求めることにする。具体的には、最後に出現した引数（B）および最近出現した 2 組の引数の差分（D）に基づいて、ストライド予測を行う。なお、$B + D$ に基づく命令区間の実行は MSP がすでに開始していると考える。SSP が N 台の場合、用意する入力予測値は、$B + D * 2$ から $B + D * (N + 1)$ の範囲とした。

3.1.4 曖昧再利用

出力に影響が現れない範囲において入力を曖昧化することにより，入力パターン数を減らし，再利用効率を上げることが考えられる。入力を曖昧化する主な手法には，よりビット数の少ないデータ型へのキャストや，下位 bit の幾つかを無視するものがある。

3.2 マルチメディア命令

次の高速化手法として、Intel 社の開発したマルチメディア命令 SSE を利用した。

ストリーミング SIMD 拡張命令 (SSE) は、高度な 2D および 3D グラフィックス、モーションビデオ、画像処理および音声認識などのアプリケーションの高速化のために開発されたものであり、8 個の 128bit レジスタ (XMM) と 32bit の制御およびステータス・レジスタ (MXCSR) を用意し、128 ビット・パックド単精度浮動小数点データ型を扱うことができる (図 6)。一方、SSE2 では XMM レジスタを用いて、倍精度浮動小数点数の並列処理を行うことができ、SSE3 ではさらに、一つの XMM レジスタ内の単精度浮動小数点値同士を処理する水平演算や複素数演算の高速化などが実装されている。

SSE は大きく以下の命令からなる。

- データ転送命令
- 算術演算命令

- 論理演算命令
- 比較命令
- シャッフル命令
- 変換命令

データ転送命令には、XMM レジスタ間、XMM レジスタ-メモリ間で 4 つのパックド単精度浮動小数点数ダブル・クワットワード・オペランドを転送する命令 MOVAPS がある。ただし、メモリアドレスは、16 バイトにアライメントが合っていなければならない。この条件を必要としない転送命令として MOVUPS がある。また、ソースオペランドの上位 64 ビットをディスティネーションオペランドの下位 64 ビットに転送する MOVHLPS やその逆方向に転送する MOVLHPS などがある。MOVHLPS では、ディスティネーションオペランドの上位 64 ビットは変化しない。算術演算命令は 2 つのパックド単精度浮動小数点オペランド同士を計算するものと、最下位のダブルワードのみ計算するものと大きく二つに分けられ、前者は命令の接尾が ADDPS、SUBPS のように PS(Packed Single-precision floating-point) となり、後者は ADDSS のように SS(Scalar Single-precision floating-point) となる。後者の場合、ディスティネーションオペランドの最下位ダブルワード以外のビットは変化しない。論理演算命令は、128 ビットそれぞれに対して 2 つのレジスタ間の論理演算を行う。他の命令のように最下位のダブルワードのみ演算を行う命令はない。比較命令は、4 つの浮動小数点値それぞれに対し 2 つのオペランド間の比較演算を行い、結果をディスティネーションオペランドに返す。この CMPPS のほかに、最下位の単精度浮動小数点値のみ比較演算を行う CMPSS もある。値は、比較が真の場合は 32 ビット全て 1 となり、比較が偽の場合は全て 0 となる。シャッフル命令は、2 つのパックド単精度浮動小数点オペランドの内容をシャッフルまたはインターリーブし、その結果をデスティネーション・オペランドに格納する。変換命令は、単精度浮動小数点フォーマットとダブルワード整数フォーマットの間で変換を行う。この他に、シフト命令などがある。

第 4 章　高速化技術の適用と評価

区間再利用、マルチメディア命令とも、高速化のための区間を特定する必要がある。本章では 2.2 での分析を元に、高速化可能性の高い区間について具体

表 4: 関数 max_successor_prob の入力パターン数

	call 回数	入力パターン数	割合
整数で call(オリジナル)	139532	15024	10.8
曖昧化なし	139532	13699	9.8
小数点以下 3 桁切り捨て	139531	12222	8.7
小数点以下 2 桁切り捨て	139486	7673	5.5
小数点以下切り捨て	138987	6094	4.9

的な高速化手法を述べ、適用結果を述べる。まず、区間再利用を用いた高速化手法を説明し評価する。次にマルチメディア命令を用いた高速化手法を説明し評価する。

4.1 区間再利用の適用

まず、区間再利用を用いた高速化手法について説明する。

4.1.1 max_successor_prob

本関数は、表 2 のビタビ計算の中の一部を処理する。中でも、式 (6) 内の $\log a_{ji}$ を計算する。入力に、ノード番号と遷移元単語番号として整数値を 2 つ与え、確率値を求める。表 3 の通り、全体に占める割合は 0.5%と小さいが、同じ引数の値で呼び出されることが多く、再利用による高速化が期待できる。また、この関数内では引数を用いて木構造化辞書から浮動小数点数を参照し、その値が計算に用いられている。そのため、関数を呼び出す直前に木構造化辞書を参照し、浮動小数点数を引数とするように書き換えることが可能となる。求めた浮動小数点数は同じ整数値について同じ値になるだけでなく、異なる整数値についても同じ値となることがあり、入力パターン数が減少する。また、この浮動小数点数を曖昧化することにより、さらに入力パターン数が減少する。このときの入力パターン数を表 4 に示す。入力音声はデータ 1 とする。ただし、曖昧化は直感的に理解しやすいよう小数点以下 x 桁を切り捨てる形式とする。実行の結果、小数点以下切り捨てると認識に影響を与えることがわかった。そこで、小数点以下 2 桁切り捨ての場合 (P2) とオリジナルの場合 (P1) を評価する。

4.1.2 addlog_array

この関数は、データ 2 の実行において 1228119 回呼び出されている。2.2 で述べたように、関数 gprune_beam の計算結果である確率値を表す二つの浮動小数点数を引数に与え、以下の計算を行う。

$$\log(\exp(x) + \exp(y)) \tag{11}$$

ここで、x, y は確率値を表す対数値とする。厳密に言えば、3 つ以上の対数値を引数に与えることを想定し引数を配列にすることで拡張性を持つが、式 (11) と仮定しても精度には問題ない。表 3 を見るとこの関数に 0.61 秒かかっているが、引数は二つだけと仮定して拡張性を奪うと 0.30 秒にまで短縮できる。そこで、評価には引数を二つだけと仮定したものを用いる。

内部で呼び出される値も含め、変化しうる入力値は引数で与えられる浮動小数点数 2 つだけなので、入力パターンが限定され、再利用の可能性は高いと考えられる。ちなみに、いくつかのデータをもとに調べたところ、曖昧化による認識の精度は問題にならない程度である。また、入力値によりソートすることで入力パターンは削減可能である。

そこで、addlog_array について以下の場合を評価する。

- オリジナルの場合 (Q1)
- 曖昧化により整数に変換した場合 (Q2)
- 曖昧化により整数に変換した後、数値によって並び替えた場合 (Q3)

4.2 区間再利用による評価

区間再利用を用いた評価は、SPARC Simulater によるサイクル数のカウントにより行う。評価には Julius プログラムを gcc -msupersparc -O2　によりコンパイルし，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。評価に用いた各パラメータを表 5 に示す。なお，命令レイテンシは SPARC64-III[7] を参考にした。

また、シミュレータでは処理に時間がかかるため、データ 2 のような長い音声を用いるのは非現実的なので、データ 1 を入力音声とする。

Julius に区間再利用を適用したときの実行時間を図 7 に示す。再利用を行わない場合の実行時間を 1 として、関数のみ区間再利用を適用する場合と、関数とループに区間再利用を適用する場合の実行時間を示す。

表 5: シミュレーション時のパラメータ

D1 Cache 容量	32 KBytes
ラインサイズ	64 Bytes
ウェイ数	4
レイテンシ	2 cycles
Cache ミスペナルティ	10 cycles
共有 D2 Cache 容量	2 MBytes
ラインサイズ	64 Bytes
ウェイ数	4
レイテンシ	10 cycles
Cache ミスペナルティ	100 cycles
Register Window 数	4 sets
Window ミスペナルティ	20 cycles/set
ロードレイテンシ	2 cycles
整数乗算〃	8 cycles
整数除算〃	70 cycles
浮動小数点加減乗算〃	4 cycles
単精度浮動小数点除算〃	16 cycles
倍精度浮動小数点除算〃	19 cycles
Read アドレス	256 word/RW
Write アドレス	256 word/RW
RF エントリ数	256
SSP 台数	3

MSP とは、プロセッサ 1 台で処理を行う機構で、MSP+SSP とは事前実行を行う機構である。4K、16K、64K は再利用表のエントリ数を表す。グラフ中の凡例はサイクル数の内訳を示しており，rs は命令サイクル数である。ts，ms はそれぞれ，再利用表とレジスタ，再利用表とキャッシュの比較に要したサイクル数である。ws は，命令区間の出力を再利用表からレジスタおよびキャッシュへ書き戻すのに要したサイクル数である。また，cm1，cm2，および wm は，それぞれキャッシュミスペナルティとレジスタウィンドウミスによるペナルティを

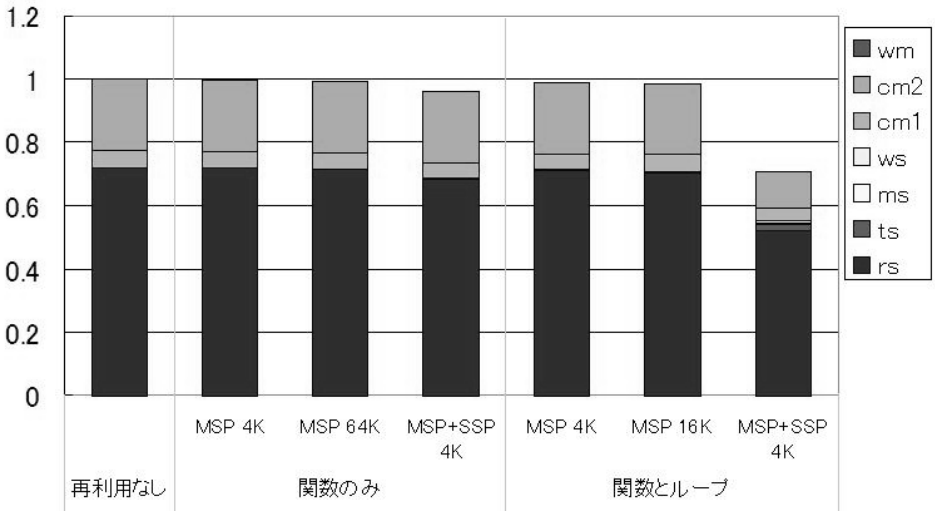

図 7: Julius に区間再利用を適用した場合の実行時間

表している。

図 7 より、エントリ数の増加による再利用効率の向上はほとんど期待できず、また、関数のみの場合は事前実行による高速化は期待できない。しかし、区間再利用の適用をループ区間に対して行うと、事前実行を行ったときに限り約 30%高速化されている。特に、命令サイクル数で 27%、2 次キャッシュミスペナルティの低減により 50%高速化されている。これより、Julius に区間再利用を適用すると、ループ区間に対する事前実行の適用が最も効果があるといえる。実際、関数のみに区間再利用を適用す場合、ループ区間を関数に置き換えると高速化されるが、全てのループ区間を関数に置き換えるのは現実的でないため、選択的に置き換えたとしても 30%を超えるのは困難である。　関数 max_successor_prob に区間再利用を適用した場合の実行時間と、関数の再利用率をそれぞれ図 8、図 9 に示す。再利用率を示すのグラフでは、濃い部分が実際に処理を行ったことを表し、薄い部分が再利用により処理が省略されたことを表す。

これらの図より、P2 では再利用率は約 50%削減できている事がわかる。しかし、全体の処理におけるこの関数の割合が小さいため、全体ではそれほど高速化されていない。また、図 9 を見ると、表 4 で示した重複するパターン数により期待される値と大きな差がある。特に、曖昧化を行わない場合 (P1) は再利用が全く行われていない。これには、次の理由が考えられる。

この関数はビタビ計算の中の関数であり、同じ時間フレーム内ではほぼ連続

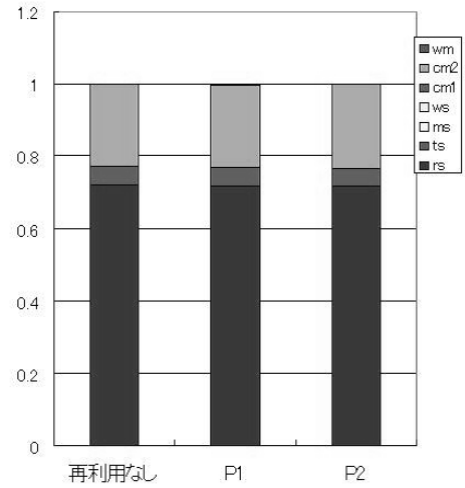

図 8: max_successor_prob に区間再利用を適用した場合の実行時間

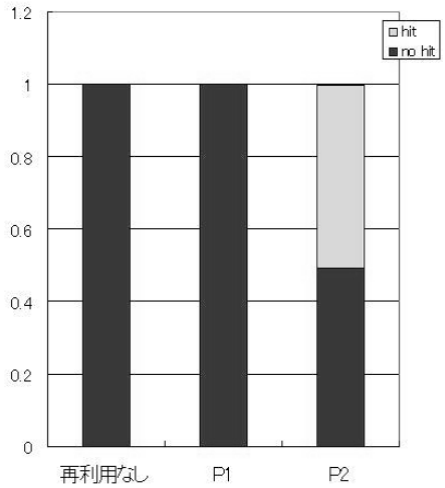

図 9: max_successor_prob に区間再利用を適用した場合の再利用率

して呼び出され、その回数は約 600 回である。しかし、Julius のアルゴリズム上、同じ時間フレーム内で同じ引数によって関数が呼び出される事はない。つまり、入力パターンが一致するのは異なる時間フレーム間となる。しかし、時刻 t のフレームでのビタビ計算と時刻 $t+1$ のフレームでのビタビ計算の間には出力確率計算を行う必要があり、この計算で再利用表を使用するために入出力表のエントリにおいて max_successor_prob の入出力値が別の関数の入出力値により置き換えられてしまう。そのため、前のフレームでの入力と同じ入力で呼び出されたとしても、入出力データが再利用表に残っていないため、再利用が全く行われない。エントリ数を 64K まで増やした場合も再利用は行われなかった。曖昧化を行う場合も同様で、同じ時間フレーム内での実行結果のみ参照できるので再利用率は期待した数値より低くなる。

関数 addlog_array に区間再利用を適用した場合の実行時間と、関数の再利用率をそれぞれ図 10、図 11 に示す。

曖昧化することにより、Q2 では再利用により削減される実行回数が 3%から 14%まで増加している。全体で見ても max_successor_prob に比べると実行時間が長い分高速化が実現されているが、それでも 2%程度しかない。また、Q3 では再利用率は 15%にまで僅かに増加しているが、ソートのための処理が増えるため全体ではサイクル数が増加している。

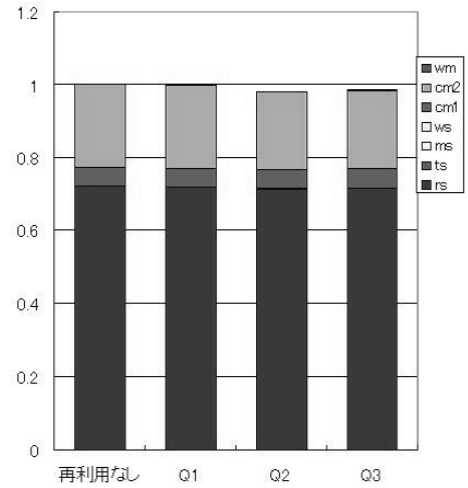

図 10: addlog_array に区間再利用を適用した場合の実行時間

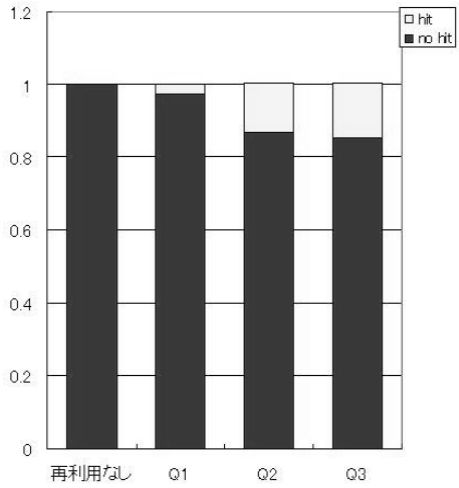

図 11: addlog_array に区間再利用を適用した場合の再利用率

4.3 マルチメディア命令の適用

一方、マルチメディア命令の適用による高速化については、出力確率計算に着目し、詳述する。

4.3.1 compute_g_beam_pruning

式 (8) では、単純な浮動小数点演算の繰り返しが行われている。具体的には、S_i が i ごとに独立であるため、並列処理が可能である。SSE を用いて並列化を行った部分の具体的なコードの一部を図 12 に示す。`%ebx,%edx,%ecx` はそれぞれ、vec,mean,var の配列を指すメモリアドレスである。また、XMM レジスタを`%xmm0`···`%xmm7` で表している。本稿で示すアセンブリコードは、第 1 オペランドがソースで、第 2 オペランドがディスティネーションとする。まず、行番号 1,2 において転送命令である MOVAPS を用いて、vec から XMM3 へ、mean から XMM1 へそれぞれ単精度浮動小数点数を 4 つずつ転送する。5 行目も同様である。3 行目の SUBPS により、4 つのパックド単精度浮動小数点数ダブル・クワットワード・オペランドそれぞれについて vec − mean をもとめ、結果を XMM3 へ格納する。同様に、MULPS では XMM3 の値である vec − mean を二乗してその結果を XMM3 に格納し、DIVPS では、XMM3 の値 $(\text{vec} - \text{mean})^2$ を var で割り、その結果を XMM3 に格納する。以上の動作により図 12 の命令列の結果が XMM3 に保存される。SSE を用いて演算を行った結果は、当然一つの XMM レジスタに四つ並行に格納されている。この 4 つの位置は XMM3[127-

```
1:  movaps (%ebx),%xmm3
2:  movaps (%edx),%xmm1
3:  subps %xmm1,%xmm3
4:  mulps %xmm3,%xmm3
5:  movaps (%ecx),%xmm2
6:  divps %xmm2,%xmm3
```

図 12: S_i の計算を並列化するコード

XMM0				$\sum_{i=0}^{j-1} S_i$
XMM3	S_{j+3}	S_{j+2}	S_{j+1}	S_j
XMM5	$th[j+3]$	$th[j+2]$	$th[j+1]$	$th[j]$

図 13: 並列処理終了後

96],XMM3[95-64],XMM3[63-32],XMM3[31-0] となり、それぞれに入っている演算結果を $S_{j+3}, S_{j+2}, S_{j+1}, S_j$ となる。また、このとき XMM0 の最下位ダブルワードには $i = j - 1$ までの S_i の和が入っており、XMM5 には閾値 $th[j]$ が転送されている。このときの XMM0、XMM3、XMM5 の状態を図 13 に示す。

本関数はデータ 2 に対する実行では、9304650 回呼ばれる。また、前節で述べたように、$\sum_{i=0}^{j} S_{\mathrm{i}}$ を求めるごとに閾値 $th[j]$ との比較を行う必要があり、SSE を使用する場合、計算を四つ同時に行うため、使用する前に比べ余分な計算をしてしまうことがある。そのことを考慮し、評価には以下のような場合を考える。

- オリジナル、閾値とのチェックを毎回行う (A1)
- 閾値とのチェックを 4 回に 1 回行うもの (B1)
- 最後まで計算するもの、厳密な計算 (C1)
- A1 と同じ動作をするように SSE を用いて書き換えたもの (A2)
- A2 と同じ動作をするように SSE を用いて書き換えたもの (B2)
- A3 と同じ動作をするように SSE を用いて書き換えたもの (C2)

A2 の具体的な方法を以下に述べる。

閾値とのチェックを行うためには $i = 0$ からそこまでの和を求める必要があ

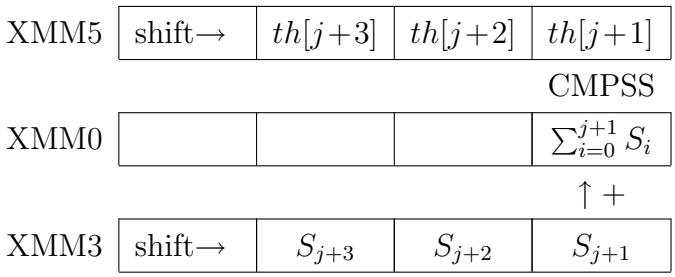

図 14: 一つずつ比較を行う方法

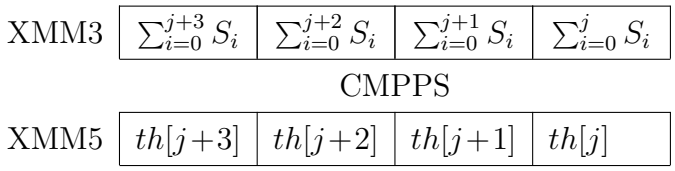

図 15: 一度に比較を行う方法

る。方法として、XMM3 の最下位ダブルワードを XMM0 に加えてその値を閾値と比較して小さければ XMM3 と XMM5 を右シフトを繰り返す方法と、S_i の和を XMM レジスタで計算し、四つ同時に閾値との比較を行う方法がある。これらの方法をそれぞれ図 14、図 15 に示す。また、図 14、図 15 に対応するアセンブリコードを図 16、図 17 に示す。図 16 の命令は、図 12 の命令のあとに 4 回繰り返される。一方、図 17 の命令は図 12 の命令のあとに 1 回処理される。アセンブリコードにおいて、右シフト、左シフト命令はそれぞれ PSRLDQ、PSLLDQ を用いる。第 1 オペランドでシフトするバイト数を指定し、第 2 オペランドでレジスタ名を指定する。比較は SSE 命令で行えるが、SSE 命令ではフラグをセットしないため分岐ができない。そのため、一度汎用レジスタに値を転送して論理演算を行いフラグをセットする。この時、転送命令として MOVMSKPS を用いる。MOVMSKPS 命令は、XMM レジスタのパックド単精度浮動小数点値から符号ビットを抽出して、4 ビット・マスクとしてフォーマットし、汎用レジスタに格納する。また、分岐は閾値を超えたときに.L80 にジャンプし、閾値を超えない場合はジャンプを行わず図 12、図 17 を繰り返す。

4.3.2 compute_g_beam_updating

処理時間は短いものの、compute_g_beam_pruning と同様の動作をする関数についても同じ手法を適用する。原則として、Julius のオリジナルのソースと

```
addss %xmm3,%xmm0
cmpltss %xmm0,%xmm5
movmskps %xmm5,%eax
test $1,%ax
jne .L80
psrldq $4,%xmm3
psrldq $4,%xmm5
```

図 16: 一つずつ比較を行う場合

```
addss %xmm0,%xmm3
movaps %xmm3,%xmm4
pslldq $4,%xmm4
addps %xmm4,%xmm3
xorps %xmm0,%xmm0
movlhps %xmm3,%xmm0
addps %xmm3,%xmm0
cmpltps %xmm0,%xmm5
movmskps %xmm5,%eax
or $0,%ax
jne .L80
psrldq $12,%xmm0
```

図 17: 四つの比較を同時に行う場合

同じ動きをする。compute_g_beam_updating はデータ 2 の実行において 300150 回呼びだされている。この関数は表 3 の通り、実行時間は短いものの、compute_g_beam_pruning と異なり必ず 25 個のパラメータ全て計算する。そのため、SSE の効果は高くなる。この関数については、以下のように動作を書き換える。

まず、モデルの中から 2 個を選びそれぞれについて

$$\sum_{i=0}^{j}((\mathrm{vec[i]}-\mathrm{mean[i]})^2/\mathrm{var[i]})(0 \leq j \leq 24) \tag{12}$$

を計算し、それぞれ $b[i]$、$c[i]$ と置いてから関数 set_dimthres により

$$th[i] = \max(b[i], c[i]) + Const(0 \leq i \leq 24) \tag{13}$$

を求めこれを閾値とする。

compute_g_beam_updating と set_dimthres について以下の場合を評価する。

- オリジナルの場合 (D1)(=A1)
- SSE を用いて書き換えたもの (D2)

一方、compute_g_beam_pruning と同様の関数で compute_g_safe という関数がある。この関数は、データ 2 の実行において 596688 回呼び出されている。また、実行時間は短いものの、閾値が一定のため毎回チェックする必要がなく SSE の効果が高くなる。関数 compute_g_safe について以下の場合を評価する。

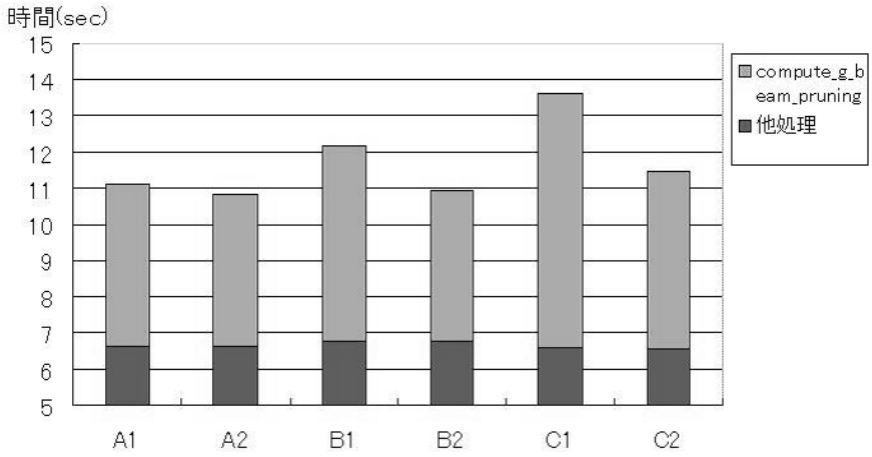

図 18: compute_g_beam_pruning に SSE を適用したときの実行時間

- オリジナルの場合 (E1)(=A1)
- SSE を用いて書き換えたもの (E2)

4.4 マルチメディア命令の評価

マルチメディア命令の適用による評価は、実行時における実時間を測定して行う。プロセッサには Intel Xeon を用いる。コンパイルは、SSE を使用しない場合は C プログラムを gcc -O2 により最適化して行う。また、SSE を使用する場合は、SSE を適用する関数群を gcc -S -O2 でアセンブリ言語に変換し、SSE を用いてアセンブリを書換え、as -o でオブジェクトファイルを生成してから他のファイルと結合して実行ファイルを作る。SSE を適用しない関数については全く同じ条件である。また、実行時間の測定は、gprof を使用する。

compute_g_beam_pruning に SSE を適用したときの実行時間を図 18 に示す。関数を書き換えると、動作が違ってくるため関数の call 回数やこの関数以外の部分にかかる処理時間が異なる。しかし、call 回数は増減が 0.2%以下、処理時間は 3%以下になり、この誤差はほとんど無視できる。図 18 の中で最も高速に動作しているのは A2 だが、関数 compute_g_beam_pruning のみに注目すると B2 が A2 より約 0.02 秒速くなっている。A1 を基準にすると、この関数の処理時間は A2 から順に、92.9%、120.2%、92.5%、156.5%、108.6%となる。A1 と A2 を比べてみると、マルチメディア命令を用いて並列化したことにより約 7%高速化されている。実際にアセンブリ言語の状態で、IA-32 命令のレイテンシの値 [6] を元に、式 (8) の計算にかかる命令列のレイテンシを計算すると、A1、A2 では

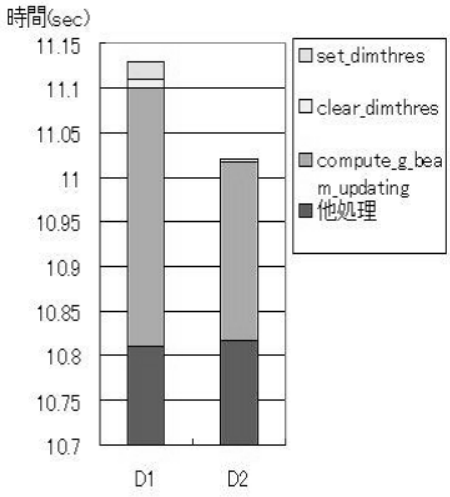

図 19: compute_g_beam_updating に SSE を適用したときの実行時間

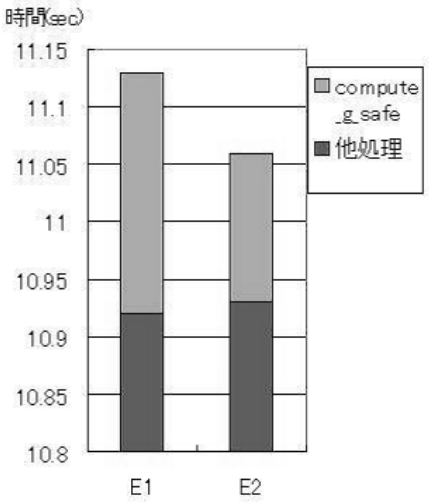

図 20: compute_g_safe に SSE を適用したときの実行時間

それぞれ 43 サイクル、104 サイクルとなる、ただし A2 では A1 の 4 回分 (4 ループ) の計算を 1 回 (1 ループ) で行っており、図 17 の計算もこのサイクル数に含まれる。また、平均すると、compute_g_beam_pruning の呼出し 1 回につきそれぞれ 5.3 回、2.0 回ループ内を回る。つまり、関数の呼出し 1 回につき平均それぞれ合計 232 サイクル、208 サイクル必要となる。計算すると、$208/232 = 0.897$、つまり約 10%しか削減できていないことになる。この関数内には式 (8) 以外の命令も前後に含まれるためサイクル削減率はさらに低くなり、7%は妥当な数値だと考えることができる。

また、B2 は A2 のように $S_{j+3}, S_{j+2}, S_{j+1}, S_j$ を求める必要がなく S_{j+3} のみを求めればよいため、その分が僅かに高速化されている。A2 と B2 ではループ数はほぼ同じになるが、A3 では最後まで計算するためループ数が大きく増える。しかし、閾値との比較をする必要がなく、図 17 のような計算をする必要もないため、全体では命令数を抑えることができる。その結果、A1 とほぼ同じ時間でより精密な計算ができる。

一方、compute_g_beam_updating、compute_g_safe に SSE を適用したときの実行時間をそれぞれ図 19、図 20 に示す。SSE を使用することで、D2 の関数群、E2 の関数での処理時間はそれぞれ D1、E1 を基準にすると 63.4%、61.9%に削

減できる。それぞれ時間にすると、0.12 秒、0.08 秒となる。ちなみに C1 を基準にした C2 の場合で削減率が 69.4%なので、それよりも効率が高いことが分かる。

これらの関数は、それぞれ独立に動作しているため同時にマルチメディア命令を適用することができる。その結果、全体で約 5%の高速化となる。

第 5 章　おわりに

本研究では、音声認識システム Julius に対し、マルチメディア命令、区間再利用を適用し、高速化を図る手法を提案した。まず、区間再利用を適用した場合については、事前実行を関数とループに対して行った場合に最大で約 30%高速化された。また、曖昧化を行うことにより、関数 max_successor_prob の再利用率を 50%向上させるなど、全体で約 2%のサイクル数を削減することができた。一方、単純な浮動小数点数演算が繰り返すことで確率値計算を行ったいる区間に対し、マルチメディア命令を適用した結果、全体で約 5%の高速化が実現できた。関数別に見ると、計算途中の値に制約がない関数においては約 36%高速化されている。また、計算過程の値に幅を設定している関数をについても、厳密な計算を仮定すると約 30%高速化され、この時全体では約 15%の高速化となる。区間再利用、マルチメディア命令を適用した区間はそれぞれ独立の処理を行っているため、全体で 7%高速化したと考える事ができる。

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の意を表します。

また，本研究に関して適切なご指導を賜った中島康彦助教授，森眞一郎助教授，五島正裕助手に深く感謝いたします。

さらに，日頃暖かく御鞭撻下さった京都大学工学部情報学科富田研究室の諸兄に心より感謝いたします。

参考文献

[1] 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).

[2] 鹿野清宏、伊藤克亘、河原達也、武田一哉、山本幹雄: 音声認識システム, オーム社 (2001).

[3] 京都大学: 大語彙連続音声認識プログラムの開発 http://winnie.kuis.kyoto-u.ac.jp/dictation/doc/lvcsr.pdf.

[4] Intel: IA-32 インテル アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル (2004).

[5] Intel: ストリーミング SIMD 拡張命令 2(SSE2) を使用した Viterbi デコーディングでの隠れマルコフ・モデルの評価 バージョン 2.0 (2000).

[6] Intel: インテル Pentium 4 プロセッサおよびインテル Xeon プロセッサ最適化リファレンス・マニュアル (2003).

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