課題研究報告書

ハードウェアにおける高速なオーディオフィンガープリントを用いた楽曲全体のマッチング手法

1810193　山名　友也

主指導教員　井口　寧 教授
審査委員主査　井口　寧 教授
審査委員　田中 清史 教授
金子　峰雄 教授
鵜木 祐史 教授

北陸先端科学技術大学院大学
先端科学技術研究科
(情報科学)

令和 2 年 2 月

概 要

オーディオフィンガープリンティング（AFP）技術は, 楽曲の知覚的な特徴を用いて生成されるフィンガープリント ID（FPID）によって楽曲の識別を行う技術である．この技術はインターネット上で使用されているが，昨今の高速なインターネット上で使用するには高速な楽曲の検索機能が必要あり、楽曲データから FPID を生成する時間は重要である．

離散ウェーブレット変換（DWT）を用いた高速でロバスト性の高い FPID を生成する HiFP2.0 アルゴリズムがある．これは，基底関数に Haar ウェーブレットを用いることで整数の加減算とビットシフトのみで処理を構成でき，FPGA などのハードウェア上で高速な処理を実現する．HiFP2.0 アルゴリズムは楽曲の冒頭 2.97 秒間のサンプルデータのみを使用して FPID を生成ので，冒頭 2.97 秒間に問題が起こった場合，楽曲検索が正しく行えない可能性がある．

そこで，本研究では，HiFP2.0 の改良として，サンプル抽出の範囲を楽曲データ全体に広げ，楽曲データ全体から一定量のサンプルデータを周期的に抽出し，FPID を生成する HiFP2.1 アルゴリズムを提案する．

HiFP2.1 アルゴリズムでは，サンプルを抽出する領域（chunk 領域）を楽曲データ全体に細分化し散在させることで確率的に楽曲データ中の問題発生部位を回避する．これによって HiFP2.0 アルゴリズムよりも高い検索精度を実現できる．

このアルゴリズムは，FPGA などのハードウェアに実装する場合，chunk 領域のサンプルデータを抽出する実装形式として 2 つのパターンが考えられる．一つは FPGA 上で送信さえてきたデータに対して逐次的に行うもの，一つは FPGA を制御するソフトウェアドライバ側で事前に行うものである．

この時，FPGA 上で行うものにおいては，固定的な回路を形成してパイプライン化などで並列処理できることが利点であるが，ホストとなる PC とハードウェアとの通信がボトルネックとなる．一方で，ソフトウェアで実装する場合は，ハードウェアでの実行時のようなインターフェイスを介した通信時間は考慮しなくてよいが，メモリアクセスなどが頻発すると実行時間が大きく増加してしまう欠点がある．

これらの実装形式を比較した場合，chunk 領域数が増加するとソフトウェアではメモリアクセスの回数などが大きく増加して FPGA 上で行った場合より大きなボトルネックとなってしまう．

このことから本研究では FPGA 上で行うものを選択し，提案した．

また，HiFP2.1 を実装する場合，最適な chunk 領域の楽曲データ全体における chunk 領域の個数を決定しなければならない．

前述した FPGA 上での chunk 領域の抽出方法においては，chunk 領域数が増加しても全体のデータ通信処理量に大きな差が生じない．また，HiFP2.1 が chunk 領域の分散による問題部位の確率的な回避を特徴としている．それに加え，離れた特徴量同士の比較から生成される FPID ビットの割合が大きく増加数とエラーが

起きやすいことが考えられる．

よって，本研究では chunk 領域数 1024 の実装として提案を行った．

このような方式で実装した HiFP2.1 は，HiFP2.0 アルゴリズムに比楽曲データ全体を使用するため，検索精度が向上するが，実行時間が悪化してしまうと考えられる．

そこで本研究では，実際に HiFP2.0 および提案手法の HiFP2.1 を実装し，検索精度及び実行時間について実験を行い，その性能比較を行う．

本研究では提案手法である HiFP2.1 アルゴリズムを FPGA に実装し，提案を行った chunk 領域の抽出方式および chunk 領域数の決定方法についての調査と検証を，AFP による検索精度や実行速度の観点から行った．この検証では，HiFP2.1 のソフトウェア実装，ソフトウェアドライバ上で chunk 領域を抽出する HiFP2.1 の FPGA 実装，HiFP2.0 の FPGA 実装を比較対象とした．

結果として，まず，提案を行った chunk 領域の抽出方法および chunk 領域数の決定方法が最適であると結論付けられた．また，検索失敗率は大幅に改善し最大で HiFP2.0 における検索失敗率と比べ，0.000175 倍となった．一方で，実行時間は 30 秒の楽曲データにおいて HiFP.2.0 の実行時間に対して最大で 2.069 倍になった．

今後の展望としては，HiFP2.1 アルゴリズムの並列化を行いたい．また，楽曲データ全体に対するサンプル抽出範囲の大きさと検索精度の関係性を調査し，実行速度を抑えることが出来るかどうかを検証し，見込みがあれば実装したい．

Keywords: オーディオフィンガープリント, FPGA, 音響技術，離散ウェーブレット変換，ブロードキャストモニタリング

Abstraction

There is a HiFP2.0 algorithm for generating fast and robust fingerprints based on the discrete wavelet transform (DWT). The HiFP2.0 algorithm generates FPIDs using only the first 2.97 seconds of a song. The HiFP2.0 algorithm generates FPIDs using only the sample data of the first 2.97 seconds of the song, so if a problem occurs in the first 2.97 seconds, the song retrieval may not be performed correctly.

Therefore, in this study, as an improvement of HiFP2.0, I propose the HiFP2.1 algorithm, which extends the scope of sample extraction to the entire music data and generates FPIDs by periodically extracting a certain amount of sample data from the entire music data.

In the HiFP2.1 algorithm, the area where samples are extracted (chunk area) is subdivided and scattered over the entire music data to probabilistically avoid problematic areas in the music data. As a result, the proposed method can achieve higher search accuracy than the HiFP2.0 algorithm.

As a result, I have implemented the proposed method. It was concluded that the method of extracting chunk regions and the method of determining the number of chunk regions proposed at the same time were optimal. In addition, the search failure rate was greatly improved, and the maximum search failure rate was 0.000175 times higher than that of HiFP2.0. On the other hand, the maximum execution time was 2.069 times longer than that of HiFP.2.0 for 30-second music data.

Keywords: Audio Fingerprinting, FPGA, Acoustic Techniques, Discrete Wavelet Transform, Broadcast Monitoring

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究の背景

1.1.1　オーディオフィンガープリント技術

オーディオフィンガープリント（AFP）技術とは，楽曲データの知覚的な特徴を抽出してコンパクトな表現の ID データ（FPID：フィンガープリント ID）とするものである．

流通している楽曲データから生成した FPID を，元となった楽曲のメタ情報と関連付けしてまとめたデータベースを構築することで，未知の楽曲データがクエリとして与えられた場合にもその FPID とデータベースの FPID 群との比較によって類似度から楽曲データを識別する．

特に，本研究で取り扱う AFP の FPID は数キロビットのビット列から構成されるベクトルデータであり，FPID 間のハミング距離による BER（Bit Error Rate: 異なるビットの割合）によって類似度を算出できる．

図 1.1 および図 1.2 に示す通り，異なる楽曲同士の FPID 間の BER は確率的に 50%に，同一楽曲同士の FPID の BER は 0%に近づく．この楽曲データ間の関係性によって BER の示す傾向に大きな差があることを利用して楽曲の識別を行う．

AFP では人間の耳で認識されるような楽曲データの知覚的特徴を用いて FPID を生成するため，楽曲データの単純な音質の劣化に高いロバスト性を示す．

1.1.2　本研究で想定する AFP 検索システムの利用例

本研究では，AFP を使用した楽曲データ検索システムはの利用例として，インターネット上において，例えば，ラジオやテレビ番組などで放送された楽曲の情報を検知し自動で権利処理を行う「使用曲目報告」[2] や，ストリーミング放送などにおいて利用楽曲に基づいて自動で広告を選出し表示する「自動広告付与」など，ユーザが楽曲データの識別結果を用いたサービスを受けるために完全な楽曲データファイルを入力するような場面を想定する．ここで，完全な楽曲データファイルとは，一般に流通する音楽 CD などに格納されていたり音楽配信サイトなどで販売されているような楽曲データを指す．

よって，本研究では，楽曲データの時間的シフトやトリミングなどの加工は想定しない．

これらの利用例では，楽曲データの検索処理技術としては例えばユーザビリティの観点などから昨今の高速なインターネット環境に対応できることが求められる．これらのことを考慮すると，FPID 生成のためのアルゴリズムやその実装もできるだけ高速であることが望ましい．

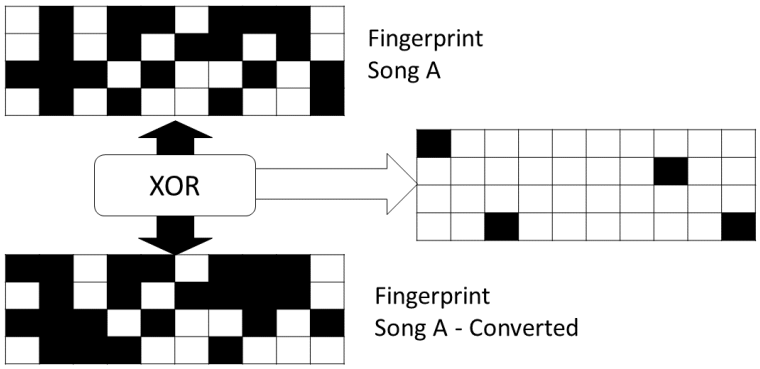


図 1.1: 同一楽曲同士でのオーディオフィンガープリント

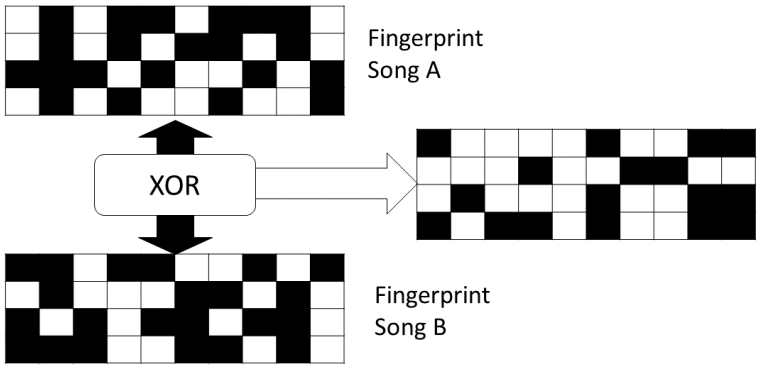


図 1.2: 異なる楽曲同士でのオーディオフィンガープリント

1.1.3 本研究において想定されるオーディオフィンガープリントシステムの構成

AFP の分野においては，楽曲データからの FPID 生成技術と FPID による楽曲データ検索技術から成るシステムを想定している．その検索精度や検索時間はそれぞれのアルゴリズムに依存する．本研究では，特に高速処理が可能な FPID 生成アルゴリズムおよびその実装について扱う．

また，本研究における提案手法は，最終的にサービス事業者のサーバ内の組込みシステムとして導入されることを想定する．その場合，図 1.3 のようなシステム構成が考えられる．

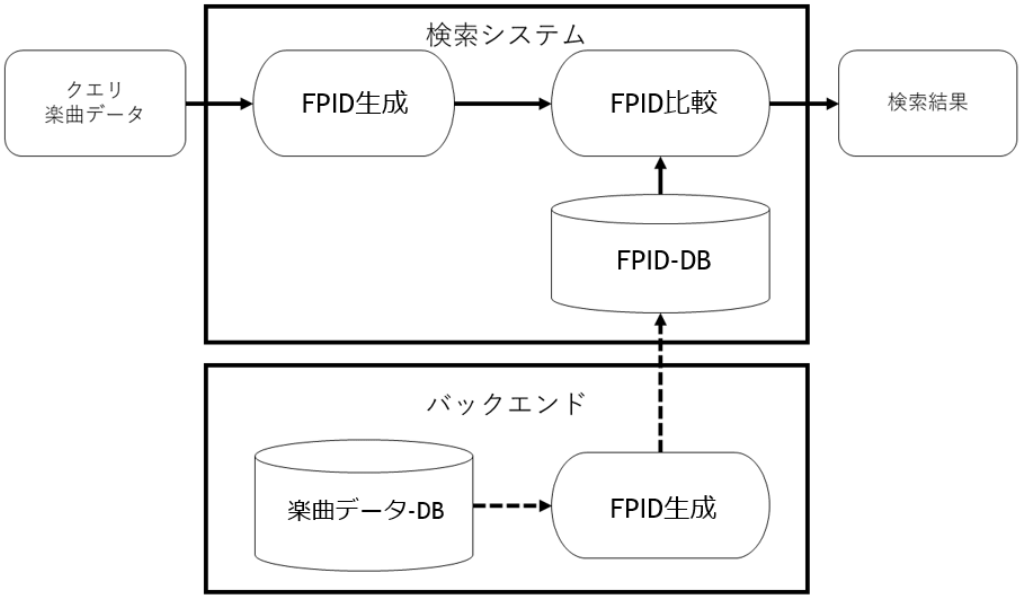


図 1.3: 楽曲検索システムの構成図

ネットワークを介してクエリ楽曲データが検索システムに到着した場合，クエリ楽曲データから FPID が生成され，事前にシステム内の楽曲データ-DB から構築された FPID-DB 内の FPID 群と比較され，類似度の高い FPID に紐づけられた楽曲データが検索結果として出力される．バックエンドは組込みシステムと接続されたコンピュータやクラウドシステムなどが想定される．バックエンドで管理されている楽曲データ-DB が更新されるごとに更新データの FPID 生成が行われ，検索システム内の FPID-DB に追加，或いはすでに格納されている FPID との交換が行われる．

また，ネットワーク通信においては通信の集中などによる輻輳の結果，ネットワーク中継器内部での通信パケットの溢れや，通信ノイズによるパケットデータの欠損などの発生が考えうる．しかし，TCP 通信などにおいては一定時間，送信パケットが到着したことを通知する受信側からの ACK 信号が送信側へ到着しない

場合，送信側は伝送に問題が発生したと見做して ACK 信号が返ってきていない部分のパケットを再送する仕組みになっている．よって，サーバ側でネットワーク通信を介して楽曲データなどを受信した場合，そのデータにサーバ内のソフトウェアなどがアクセスできるタイミングではすでにパケットロスによるデータ欠損は解決されている．よって，サービス事業者のサーバ内の組込みシステムとして楽曲データを扱う場合，パケットロスなどによるデータ欠損を考慮する必要はない．

また，本研究における AFP システムを使用する場合には，FPID-DB の FPID は，個別のシステム内で生成する形態，クラウドシステム上などで一元的に管理される楽曲データ-DB から生成されたものをネットワークを介して各システムに配信する形態，あるいは同種のシステム同士で生成したものを共有する形態を想定している．また，FPID の比較は基本的に同じアルゴリズムから生成されたもの同士でしか正しく行えないため，同一のアルゴリズムに基づくシステムのみが FPID の共有を行える．詳しくは第 2 章の 2.2 で詳述するが，FPID 生成技術およびそれを利用した AFP 技術としてはバイナリ方式，ランドマーク方式，コンピュータビジョン方式，DAL 方式など様々な方式がある．これらの方式ではそれぞれ FPID そのものの定義やデータ形式が異なるため，統一された AFP の規格というものは現状存在していない．であるため，FPID の共有などによる利便性の点について考えた場合，同じアルゴリズムに基づくシステムが統一的に普及していることが望ましい．

1.2 研究目的

HiFP2.0 はインターネット上でのリアルタイムな FPID 生成処理に必要とされる性能を満たすアルゴリズムである．しかし，HiFP2.0 はその性質上，ある楽曲データから FPID を生成する場合、楽曲開始 2.97 秒間のサンプルデータのみを固定的に使用するものである．つまり，ある 2 つの楽曲の FPID のハミング距離を算出する場合，どちらも楽曲データの冒頭 2.97 秒分の領域の特徴のみを用いているということになる．この状態では，冒頭 2.97 秒の部分は似通っているような実際は異なる 2 つの楽曲同士を同一の楽曲であると見做してしまったり，冒頭 2.97 秒の領域に音質の大きな劣化などの問題が発生した場合に同一楽曲と識別できなくなる恐れがある．

そこで本研究では，HiFP2.0 を改良し，冒頭 2.97 秒間のみから行っていたサンプルデータ抽出の範囲を楽曲全体に拡大することで，問題発生部位のサンプルデータが FPID 生成に大量に使用されることを確率的に回避し，楽曲検索の識別精度を改善することを考える．この時，FPID のサイズ増加による検索時間への影響を考慮し，生成する FPID のサイズは HiFP2.0 と同様である 4,096bit とする．この場合，その生成に必要となるサンプルデータ量も HiFP2.0 と同様になる．

よって，FPID 生成のためのサンプルデータは一定量でありつつも，それを楽曲データ全体から取得するアルゴリズムが必要である．本研究では，FPID 生成に使

用する一定量のサンプルデータ抽出を楽曲データ全体に拡大するため，複数の小さなサンプル抽出領域（chunk 領域）を楽曲データ全体に周期的に配置する HiFP2.1 アルゴリズムを提案する．

また，FPGA などのハードウェアに HiFP2.1 を実装する場合，まず，サンプルデータの抽出をどのように行うかを決定する必要がある．考えられる方式としては，一つは FPGA 上に送られてきた楽曲データ全体から逐次的に必要なサンプルデータを抽出するもの．もう一つは，FPGA を制御するホスト PC のソフトウェアドライバ上で事前に行うものである．これらの方式の決定は，HiFP2.1 の実装の際の実行速度に影響を与える．前者は固有の回路形成による並列処理の利点があるもののホスト PC と FPGA 間での通信というボトルネックを持つ．また，後者は外部との通信による遅延はないが，メモリアクセスなどが増加した場合にボトルネックになりうる．よって，本研究ではどちらの方式がより HiFP2.1 の実装に適しているかについての提案とその検証を行う．

さらに，HiFP2.1 を実装するにあたっては，楽曲データ全体に分散させる chunk 領域の最適な数を決定しなければならない．HiFP2.1 は楽曲データ全体に複数の chunk 領域を分散配置することで確率的な問題部位の回避を行うので，可能な限り chunk 領域数は多い方がよいと考えられる．一方で，実装方式などによっては chunk 領域数の増加が実行速度に影響を与えることも考えうる，よって，本研究では最適な chunk 領域数の決定方法についても提案を行い，検証する．

また，HiFP2.1 アルゴリズムは HiFP2.0 アルゴリズムに比べ楽曲データ全体のサンプルデータを抽出するため，検索精度は向上しうるが，実行時間が悪化してしまうことが想定される．よって，本研究では HiFP2.1 アルゴリズムの FPGA 実装に加え，HiFP2.0 についても実装を行い，その実行速度や検索精度について，比較，検証を行う．

1.3 まとめ

本稿の第 1 章では，研究の背景と研究目的を述べた．

本研究ではユーザが楽曲データの識別結果を用いたサービスを受けるために完全な楽曲データファイルを入力するような場面を想定する．そのような場合に高速に処理が行えるアルゴリズムとしての HiFP2.0 を元に，これを改良した HiFP2.1 アルゴリズムを提案する．その上で，楽曲データの冒頭 2.97 秒間のみを FPID 生成に使用する HiFP2.0 が音質劣化などの問題によって識別が正しく行えない状況において，HiFP2.1 の仕組みによってどの程度識別率の改善が見られ，速度が悪化してしまうかを調査する．

第 2 章では，オーディオフィンガープリント (AFP) の定義や楽曲のひずみの定義，HiFP2.0 アルゴリズムの定義，説明に加え，AFP 技術に関する複数の先行研究を紹介する．第 3 章では，HiFP2.0 の問題点である固定的なサンプル抽出範囲とそれに伴う弊害について指摘し，それを解決するための本研究の提案手法 HiFP2.1

アルゴリズムの楽曲全体からのサンプル抽出手法とその実装について述べる．また，実装の際に必要となる最適な実装形式の決定法，chunk 領域数の決定法についても述べる．第 4 章では，HiFP2.1 および HiFP2.0 の複数の方式での実装を用いて提案手法の評価実験を主に実行速度，計算精度の観点から行う．それによって，提案した HiFP2.1 の最適な実装方式や chunk 領域数についての検証も行う．そして，第 5 章でまとめと今後の課題を提示し，締めくくる．

第2章　オーディオフィンガープリントと関連研究

2.1　はじめに

この章では，本研究において取り扱うオーディオフィンガープリント（AFP）技術についてと，AFP における様々な FPID 生成技術の説明を行う．

さらに，その中でも特に本研究で中心的に取り扱う FPID 生成アルゴリズムである HiFP2.0 について述べる．その上で HiFP2.0 がその構造上抱える問題点についても述べる

最後に，HiFP2.0 などのオーディオフィンガープリント生成技術を実装する対象となる FPGA について説明し，まとめを行う．

2.2　AFP の FPID 生成技術

AFP 技術における FPID 生成技術は様々な種類のアルゴリズムが提案されている．特に，FPID の表現形式によって，AFP システムは主に以下の様な 4 種類に分類される．

- バイナリ形式
  特徴量の 2 点間の関係性をバイナリ表現のベクトルデータ化したものを FPID とする形式.
- ランドマーク形式
  特徴量の 2 点間の関係性をハッシュテーブル化したものを FPID とする形式. 特にピーク特徴量を用いる場合に使用される．
- コンピュータービジョン形式
  オーディオスペクトログラムの画像データを FPID とする形式.
- DAL（divide-and-locate）形式
  スペクトログラムをいくつかの小さな領域に分割したものを FPID とする形式.
- その他の形式

また，AFP 技術のアルゴリズムに広く用いられる特徴量抽出に対するアプローチとしては音声データのスペクトログラムへの分析に基づいたものが主である．そ

の上で，用いられるスペクトログラム及びその分析手法の違いを含め，以下に提案されている手法を整理する．

2.2.1 バイナリ形式の FPID 生成システム

バイナリ形式の AFP 生成システムとしては，次のようなものがある．

まず，音声データの時間-周波数における特徴量を利用して FPID を生成するタイプの AFP システムの原型を Cano ら [3] が発表したが，その後に続く同様の手法の中で最もポピュラーな手法として，Haitsma ら [4] のものがある． これは、エネルギースペクトルの時間的遷移を特徴量とし、解析手法として高速フーリエ変換 (FFT) を採用している。この手法では，音声信号に 370ms のフレームを 31/32 のオーバーラップとともにずらしつつ Hanning Window をかけて FFT を行い，300 Hz から 2000 Hz の間の 33 の周波数帯のエネルギーを 257 フレーム分得る．各フレームの連続した 33 の周波数帯間の隣接要素の差分 32 項目（サブ FP）を計算し，さらに各フレームの 32 のサブ FP それぞれについて各フレーム間で隣接要素の差分 256 通りを計算する．こうして 32 × 256 項目の特徴量が取り出され，それら特徴量の正負に 1/0 を割り当てることで 8192bit の FPID になる．また，Haitsma らはストレージ容量の圧迫を避けながら楽曲データの速度変更に対応する改良手法も提案している．[5] この手法では，主に周波数の不整合に対応するため，自己相関関数のシフト不変性を利用する．そのために 33 の周波数帯ではなく，512 の周波数帯からサンプリングされたパワースペクトルの自己相関を 33 にダウンサンプリングし，FPID に変換する，これらの研究で FPID 生成に用いている FFT は，離散 Haar ウェーブレット変換などと比べてハードウェア処理が遅く，高速なインターネットの環境には適していない．

Liu ら [6] は、ひずみが大きいクエリに対応するために，Haitsma ら [4] の手法と比較して FFT で出力された周波数帯域のエネルギーの時間シーケンスに離散コサイン変換 (DCT) を用いるマルチハッシュ (MLH) 法を提案した．L 個の DCT 係数は，L 個の連続したサブバンドエネルギーに対し離散コサイン変換により計算され，低次の k 個の係数のみが保持される．そして，k 個の DCT 係数のそれぞれについて、Haitsma ら [4] と同様の方法で 8192bit の FPID が計算される． k 個のハッシュテーブル内の FPID の様々な組み合わせは検索精度を向上させるが，処理に必要なメモリサイズは増加する．

Kamaladas ら [7] は，Liu ら [6] と同様にひずみの多い音楽データに対応するため，FFT の代わりにウェーブレット変換を用いて特徴量を抽出する方法を提案している．この方法では，370ms のフレームにハニング窓と 28/32 のオーバーラップによってウェーブレット変換を適用し，5 層のウェーブレットに分解する．これによって得られたゼロクロス比，エネルギー，分散，重心の値から，フレームごとに 8bit のサブ FP を得ることができる．この方法は高度に歪んだ音楽データに

対して高い性能を示し，同時に生成される FPID のビット数も少ない．しかし，本研究では FPID の生成速度についての言及がない．

Kim ら [8] は，実数値 FPID をバイナリ FPID に変換する際の情報損失やパフォーマンス低下を防ぐために，ペアワイズブースティングを用いた手法を提案している．この手法では，372ms のフレームと 186ms のオーバーラップによる FFT により，楽曲データから 186ms ごとに一次正規化スペクトルサブバンドモーメントの特徴を抽出する．それを学習済みのペアワイズブースティングによって弱い分類器として選ばれたバイナリ変換方法を用いてバイナリ FPID に変換する．この研究では，実行速度について触れられていない．

Ingo ら [9] は，モバイル機器をターゲットとして，低消費電力に着目した FPGA ベースの FPID 生成技術を提案している．本研究では FFT を用いているが，Haitsma らの場合と同様に，本研究で想定している環境では生成速度が十分に速いとは言えない．

Schreiber ら [10] は，FPID 中のロバスト性の高い部位のみを用いて DB を構築する方法を提案している．しかし，彼らは Haitsma らと同じ FPID 生成法を用いており，速度は十分に速いとは言えない．

Ibarrola ら [11] は，楽曲データ信号のエントロピー信号が音質劣化に対して高いロバスト性を持つことを利用して FPID を生成する方法を提案している．楽曲データのヒストグラムから算出されたエントロピー信号を一次微分して符号化し，バイナリ形式 FPID を生成する．この研究では，FPID 生成の速度について触れられていない．

荒木ら [12] は特徴量としてエネルギースペクトルの時間的遷移を用い，分析手法に離散 Haar ウェーブレット変換を採用して FPID を生成する手法（HiFP2.0）を提案している．この方法は，ハードウェアによる高速処理が行える．本研究では特に，このアルゴリズムについて扱う．

2.2.2 ランドマーク形式の FPID 生成システム

ランドマーク形式の AFP 生成システムとしては，次のようなものがある．

ランドマーク形式で最もポピュラーなものに，Wang ら [13] の研究がある．オーディオスペクトログラムのピークは，音楽データに起因するピークとノイズに起因するピークが別個に現れるという性質がある．これを元に，FFT によるオーディオスペクトログラムからある一つのピーク点（アンカー）と他の時間-周波数のピーク点同士をペアにして，2 者の周波数における相対距離と 2 者の時間における相対距離を量子化したものを一つのハッシュとし，複数のピーク点のものを連結する．これによってエネルギーピークを表すコンパクトなシグネチャを FPID として生成するロバストな AFP システムを提案している．しかし，約 20,000 曲のデータベースにおけるクエリ楽曲の検索時間は 5〜500 ミリ秒程度であり，インターネット上での高速処理には十分とは言えない．

また、Fenet ら [14] は Wang ら [13] の研究を元に，FFT の代わりに Constant-Q 変換（CQT）を用いて対数スケールの周波数軸の周波数スペクトルを得ている．この研究では，ピッチシフトに強いロバスト性を持っているが，1 秒の長さの信号に対しての実行速度は 280ms としており，本研究で求められる高速環境には向いていない．

Anguera ら [15] は，MASK(Masked Audio Spectral Keypoints) という手法を提案している．MASK は，時間周波数スペクトログラムにメルフィルタバンク処理を行い，その中から閾値以上の複数のピーク点を取る．これらのピーク点を中心とした 5 つのメル周波数帯にまたがって 190ms の MASK 領域を設定し，そこに含まれる 5 つのグループのエネルギー値の比較によって 22bit のバイナリ形式 FPID ビットを生成する．これと”楽曲データの ID”，”それぞれのピーク点までの経過時間”をインデックスし，FPID とする．この研究の手法はコンパクトかつロバスト性が高いが，実行速度について言及されていない．

Jia ら [16] の提案は Anguera ら [15] の提案を元にしており，時間-周波数スペクトログラムを時間軸をフレーム単位に分割し，周波数軸に 18 バンドに分割する，そして，各ピーク点の周囲に 5×19 個の MASK 領域を設定する．その領域において，ピーク点に対する４つのサブ領域グループのエネルギーベクトルに局所的線形埋め込み (LLE) を適用して次元削減を行う．そして，削減されたエネルギーベクトル間の大小比較を行い，バイナリ形式の FPID を生成する．この研究の手法はコンパクトかつ楽曲の速度変化への体制も高いが，FPID の生成速度については触れられていない．

Georger ら [17] は，オーディオスペクトログラム中のランドマークに基づいて時間軸上の拡大縮小に対してロバストな AFP 法を提案している．FFT によって時間-周波数スペクトログラムを取得し，それをハイパスフィルタに通し，タイムスライスごとの周波数のピーク点を抽出する．それらピーク点において，一定以上距離を持つ中で最も大きい 3 つのピーク点の組み合わせを選択，30bit で量子化し，特徴量とする．そして，これをタイムスライス分連結し，FPID とする．この研究では，FPID の処理時間については言及されていない．

Jie ら [18] は，検索時間とハッシュ衝突の時間短縮のために，時間-周波数スペクトログラムの最大値と最大増加量のピーク点から，あるピーク点（アンカー）において最近傍のピーク点とのペアと，その次に近いピーク点とのペアとの 2 段階のインデックスを生成する手法を提案している．これによって Shazam のそれよりも高精度で高速だが，実行時間は 1 クエリで 9,000ms であり，本研究で求める環境には満たない．

Cotton ら [19] は，音声トラックによる映像データの識別のために，Matching Pursuits (MP) アルゴリズムを用いた手法を提案している．MP は、疎な信号を過完全辞書の基底ベクトルの線形結合として近似符号化し，様々な時間周波数スケールで信号中の顕著な特徴を抽出する．この研究は計算複雑度が比較的高く，処理速度の面では本研究の環境での高速検索には有効ではない．

2.2.3 コンピュータービジョン形式の FPID 生成システム

コンピュータービジョン形式の AFP 生成システムとしては，次のようなものがある．

Ke ら [20] は，音声データのスペクトログラムを重なり合う二次元画像データとして扱い，AdaBoost による分類器の学習とそれによる識別の手法を提案した．事前に用意した教師データを元に，25,000 個のボックスフィルタの候補リストから 32 個を選んで組み合わせた分類器を学習し，識別に用いる．この研究は，ノイズに高いロバスト性を持つが，処理時間について明示されていない．

Baluja ら [21] は，Ke ら [20] の方法における学習アプローチの代わりに，重なり合うスペクトログラムの画像データを多解像度の Haar ウェーブレットに分解し，その上位 t 個の符号情報を min-hash を用いて FPID とする Waveprint システムを提案している．この研究は音質劣化に高いロバスト性を持つが，FPID 生成速度については触れられていない．

Zhu ら [22] は、音楽データの 2 次元スペクトログラム画像上の SIFT（Scale-Invariant Feature Transform）特徴量を用いる時間拡張やピッチシフトに強い AFP 法を提案している．SIFT は画像上の一意な点を用いた画像間の類似度測定であり，画像上の物体識別に一般的に用いられている．この研究は，この関数を 2 次元スペクトログラム画像の類似度による音楽データの識別に適用する．ピッチシフトに対するロバスト性は高いが，FPID の生成速度への言及はない．

Malekesmaeili らは [23] は，楽曲データの 2 次元スペクトログラム画像上でタイムクロマ分析を用いて局所的に抽出した特徴量を元にした AFP 法を提案している．これは時間とピッチにおいて不変であり，SIFT よりも堅牢である．しかし，FPID の生成速度への言及はない．

これらスペクトログラムの画像データを使用する手法は，一般的に情報の削減率が低く，必要とされる記録媒体やメモリの容量が大きい．

2.2.4 DAL（divide-and-locate）形式の AFP 生成システム

DAL（divide-and-locate）形式の AFP 生成システムとしては，次のようなものがある．

そもそも DAL は，音声信号の特徴ベクトルのヒストグラムを FPID として使用する時系列アクティブ検索（TAS）[24] という信号検索法を元にしたものであり，音声信号の時間-周波数スペクトログラムをいくつかの小さな領域に分割したものを FPID とする．

Nagano ら [25] は，DAL において，スペクトログラムをベクトル量子化し，これを集めたベクトル量子化コードブックを FPID として用いることで探索を高速化する手法を提案している．これは雑音に対して高いロバスト性を持つが，DAS

やDALといった手法は長時間の音声信号の中からクエリの音声信号を高速に探索する場合に用いられる手法であり，本研究の研究目的とはそぐわない．

2.2.5　その他の形式のFPID生成システム

その他の方法として，以下のようなものがある．

Saravanosら [26] は，K-SVDアルゴリズムを用いて楽曲データベースを学習することでグローバル辞書を構築し，それに対してOMPアルゴリズムを用いることでロバストなFPIDを作成する手法を提案している．この研究は非常に高い識別率を持つが，計算複雑度が高い．

Khemiriら [27] は，ALISP（Automatic Language Independent Speech Processing）ツールを使用して，音声データをALISPユニットとしてセグメント化し，隠れマルコフモデルとして学習させたものを楽曲検索に使用する手法を提案している．検索にはBLAST（Basic Local Alignment Search Tool）を使用している．この研究は広告や楽曲に対し高い識別率を持つが，FPID生成時間に個別に触れられていない．

Ramalingamら [28] は，ガウス混合モデルによって音声信号を23msのフレームに分割，フレームのオーバーラップを1/2とし，短時間フーリエ変換にかけ，”Shannonエントロピー”，”Renyiエントロピー”，”スペクトル中心”，”スペクトルバンド幅”，”スペクトルバンドエネルギー”，”スペクトル平坦度”，”スペクトルクレスト係数”，”MFCC” を特徴量としてガウス混合モデルを学習させ，そのパラメータ集合θを各音声データのFPIDとして使用する．本研究では高い識別率と偽陽性率を誇るが，FPIDの生成速度への言及はない．

Seoら [29] は，正規化されたスペクトルサブバンドモーメントをFPIDに使用する方法を提案している．音声データを長さが371.5msで1/2がオーバーラップしたフレームに分割し，Hamming Windowによって周波数領域に変換する．そして，分割した帯域それぞれのサブバンド特徴量を取り出し，時間的に連続するように並べ，１次正規化あるいは２次正規化またはSFMしたものをFPIDとする．この研究はあくまで１次正規化，２次正規化，SFMの三者間で性能比較を行ったのみであり，速度についても言及されていない．

これらの手法は計算複雑度が高く，現在の処理速度では十分とは言えない．

2.3　本研究で取り扱うAFPの形式

本研究で取り扱うAFPにおけるFPIDとは，楽曲のデジタルデータから抽出したメロディなどの知覚的な特徴量を正規化し，コンパクトなビット列のベクトルデータとしたものである．このFPID同士のハミング距離からBERを算出し，その値の低さを類似度として同一楽曲を識別できる．

ある 2 つの楽曲データファイルを比較する場合に正規化された FPID を使用することは，様々な利点を持つ．

ここでは，ロバスト性，検索高速性，スケーラビリティを挙げる．

ロバスト性とは，FPID の生成に人間の聴覚で判別されるような知覚的な特徴量を用いているため，楽曲データにある程度の音質的な劣化が生じた場合においても高い識別能力を維持できるということである．

高速検索性とは，AFP は FPID 生成の過程において楽曲識別に不要な情報を削減する機能を持っており，データ量とメモリ占有率を抑制するため，高速な検索を実現することが出来るということである．

スケーラビリティとは，FPID は知覚的な特徴から生成される正規化されたコンパクトなベクトルデータであり，保存形式の違いなどを考慮する必要がないため，大規模なデータベース構築でもサイズの増加を抑制できるということである．

2.3.1 本研究で扱う FPID 生成アルゴリズム：HiFP2.0

ここでは，本研究で取り扱う HiFP2.0 アルゴリズムおよび HiFP2.0 が用いる楽曲データについて説明する．

2.3.1.1 楽曲データの形式と非可逆的変化

楽曲データの保存形式変換を行うと，データ圧縮などで知覚的な特徴量にも非可逆的な変化が発生する場合があり，そこから生成される FPID のビットエラーの要因となりうる．この様な事態が発生すると，楽曲の識別率が下がってしまう．

本研究ではこのような非可逆的変化に対するロバスト性を取り扱う．

また，本研究では，実際に FPID 生成の対象となる楽曲データの保存形式はアルゴリズムの性質上，PCM 形式の wav ファイルのみである．データの非可逆的変化の識別率に対する影響の調査のために他の保存形式も用いることがあるが，この場合も保存形式を wav ファイルに変換する．

また，本研究は FPID 生成技術部分を主に扱うため，保存形式間の変換速度の問題は取り扱わないものとする．

2.3.1.2 wav ファイルの構造

本研究では，FPID 生成に用いる楽曲データの保存形式として，一般に広く普及している PCM 形式の音楽データフォーマットである wav ファイルを用いる．wav ファイルは，汎用メタファイルフォーマットである RIFF(Resource Interchange File Format) に準拠しており，音声データをタグとともにチャンクという単位で格納する．wav ファイルの構造は図 2.1 のようになっている．

RIFF識別子	4byte	“RIFF”(0x52494646)
ファイル サイズ	4byte	
フォーマット	4byte	“WAVE”(0x57415645)
fmt識別子	4byte	“fmt “(0x666D7420)
fmtチャンクのバイト数	4byte	16(0x10000000)
音声フォーマット	2byte	1(0x0100)
チャンネル数	2byte	2(0x0200)
サンプリング周波数(Hz)	4byte	44.1kHz(0x44AC0000)
1 秒あたりバイト数の平均	4byte	88,200(0x88580100)
ブロックサイズ	2byte	4(0x0400)
ビット／サンプル	2byte	16(0x1000)
サブチャンク識別子	4byte	“data” (0x64617461)
サブチャンクサイズ	4byte	
データ	*	

fmtチャンク

dataチャンク

図 2.1: 想定する Wav ファイルの構造

ヘッダは，wav ファイルが RIFF に準拠することを示す“RIFF”(0x52494646) の識別子から始まる．その後はファイルサイズの数値，フォーマットを示す“WAVE”(0x57415645) の識別子と続く．次に，これ以後のチャンクからフォーマット (fmt) チャンクであることを示す“fmt “(0x666D7420) の識別子があり，それ以後はファイルの各種フォーマットが記載される．さらに次には，これ以後のチャンクからデータチャンクであることを示す“data”(0x64617461) 識別子と波形データのバイト数が示され，その後は，ファイルがステレオ 16bit である場合，時系列順に”左側の音声”→”右側の音声”の順で 16bit の符号付き整数でサンプルデータが格納されている．

2.3.1.3 HiFP2.0 の FPID 生成アルゴリズム

HiFP2.0 のアルゴリズムを図 2.2，2.3 およびアルゴリズム表 2.1，2.2 ，2.3 に示す．アルゴリズムは荒木ら [12] の論文から引用する．

前述した通り，HiFP2.0 が想定する楽曲データ（wav ファイル）の音声信号は 16bit のサンプルデータの集合である．また，この wav ファイルのサンプリング周波数は 44,100Hz とする．HiFP2.0 では，楽曲データの冒頭に無音の領域が存在する可能性などを考慮して楽曲データの先頭から 2 秒間を破棄領域としてスキップし，その後の約 2.97 秒間のサンプルデータ 131,072 を抽出する．そして，そのサンプルデータ列に対し離散ウェーブレット変換のサブバンド分解処理を行う．

ここで，離散 Haar ウェーブレット変換は次のような式で成り立つ．

$$Hi(i) = \frac{wav(2i) - wav(2i+1)}{2} \tag{2.1}$$

$$Lo(i) = \frac{wav(2i) + wav(2i+1)}{2} \tag{2.2}$$

一般的に離散 Haar ウェーブレット変換においては，隣り合う要素同士について算術平均および移動平均の処理を行う．算術平均を行った場合，大きな変化をなだらかにするので，Lo-Pass-Filter（低域周波数を通過させ，広域周波数を遮断するフィルタ）を通過させてダウンサンプリングを行ったことと等価になり，移動平均を行った場合，大きな変化を強調するので，Hi-Pass-Filter（広域周波数を通過させ，低域周波数を遮断するフィルタ）を通過させてアップサンプリングを行ったことと等価になる．これによって，ダウンサンプリングした要素の配列（Lo(i)）と，アップサンプリングした要素の配列（Hi(i)）の 2 つに分解する．ここで生成された Lo 配列に対し更に離散 Haar ウェーブレット変換を複数回行うことで，より低域の周波数成分を抽出することができる．これをサブバンド分解と呼ぶ．

HiFP2.0 では抽出されたサンプル群を 8 サンプルずつに区切る．その 8 サンプルに対して離散ウェーブレット変換のサブバンド分解を 3 回行い，最終的に生成された 1 サンプル分の Lo 要素を特徴量として利用する．この Lo 特徴量を並べた Lo 配列において，3 つ飛ばしの特徴量間で大小比較を行い，前後のどちらが大きいかに 1 と 0 を割り当て，1bit 分の FPID 要素（FPID ビット）とする．抽出したサンプルデータ 131,072 から生成される 4,095bit 分の FPID ビットを並べた FPID 配列の末尾に 0 の 1bit を加えて，4,096bit の FPID として完成する．前述したように，HiFP2.0 で使用される離散 Haar ウェーブレット変換は以下に示すように整数の加減算，シフト演算のみで構成され，結果として取り出され Hi 要素の特徴量を用いた FPID 要素の生成のための処理も比較処理のみである．よって，階層的なパイプライン処理が行え，専用の回路を構築して高速にハードウェア処理を行うことが出来る．

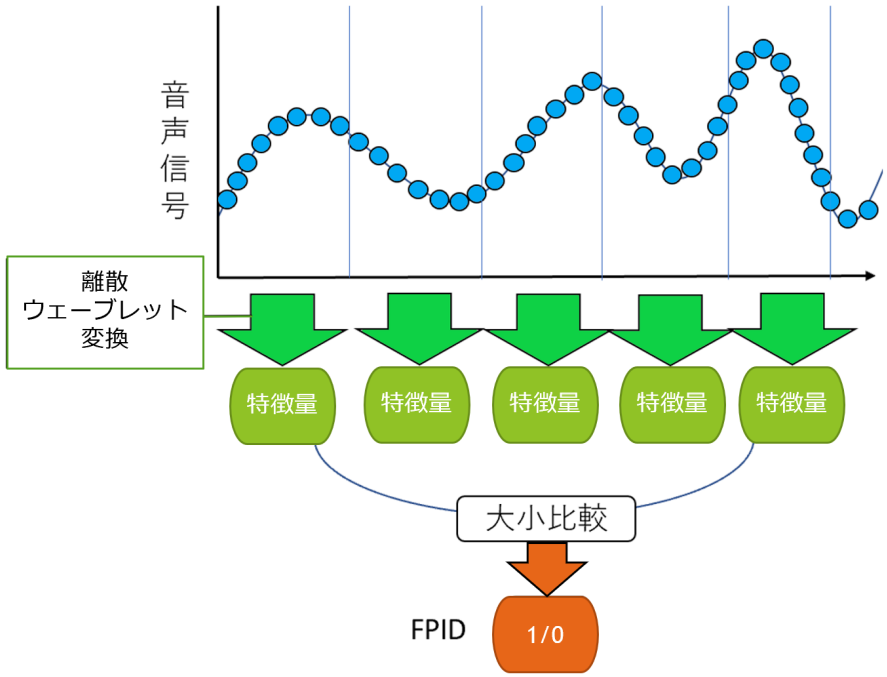


図 2.2: HiFP2.0 の図解

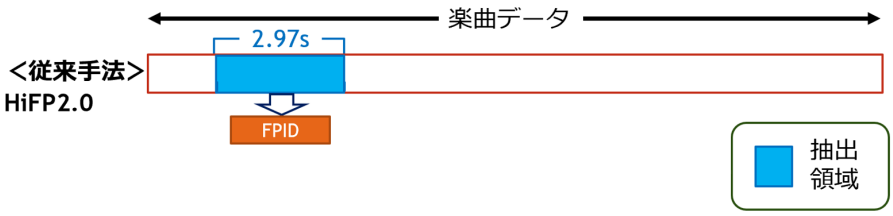


図 2.3: HiFP2.0 の抽出領域図解

アルゴリズム 2.1 HiFP2.0 のアルゴリズム [12]

Input: $wav[]$: wav 楽曲波形データ $(16bit/sample)$

Output: $AFP(4096bit)$

1: $n = 131072$(使用するサンプル数);
2: $m = 4, 96$(生成する $FPID$ のビット数);
3: $Hi[], Lo[] \leftarrow DWT(wav[], 0, n)$;
4: $AFP[] \leftarrow GAFP(Lo[], m)$;
5: $returnAFP$

アルゴリズム 2.2 Haar ウェーブレット変換よるサブバンド分解アルゴリズム (DWT) [12]

Input: $wav[], n, m$
Output: $Hi[], Lo[]$
while $n >= m$ **do**
  $n = n/2;$
  for $i \leftarrow 0; i < n; i+ = 1$ **do**
    $Hi[] = (wav[2 \times i] - wav[2 \times i + 1])/2;$
    $Lo[] = (wav2[2 \times i] + wav[2 \times i + 1])/2;$
  end for
  $wav[] \leftarrow Lo[];$
end while
$returnHi[], Lo[];$

アルゴリズム 2.3 GAFP [12]

Input: $Lo[], m$
Output: $AFP(m)$
for $i = 0; i < m - 4; i+ = 4$ **do**
  $temp = Lo[i] - Lo[i + 1];$
  if $temp > 0$ **then**
    $AFP[i] = 1;$
  else
    $AFP[i] = 0;$
  end if
end for
$AFP[m/4 - 1] = 0;$
$returnAFP;$

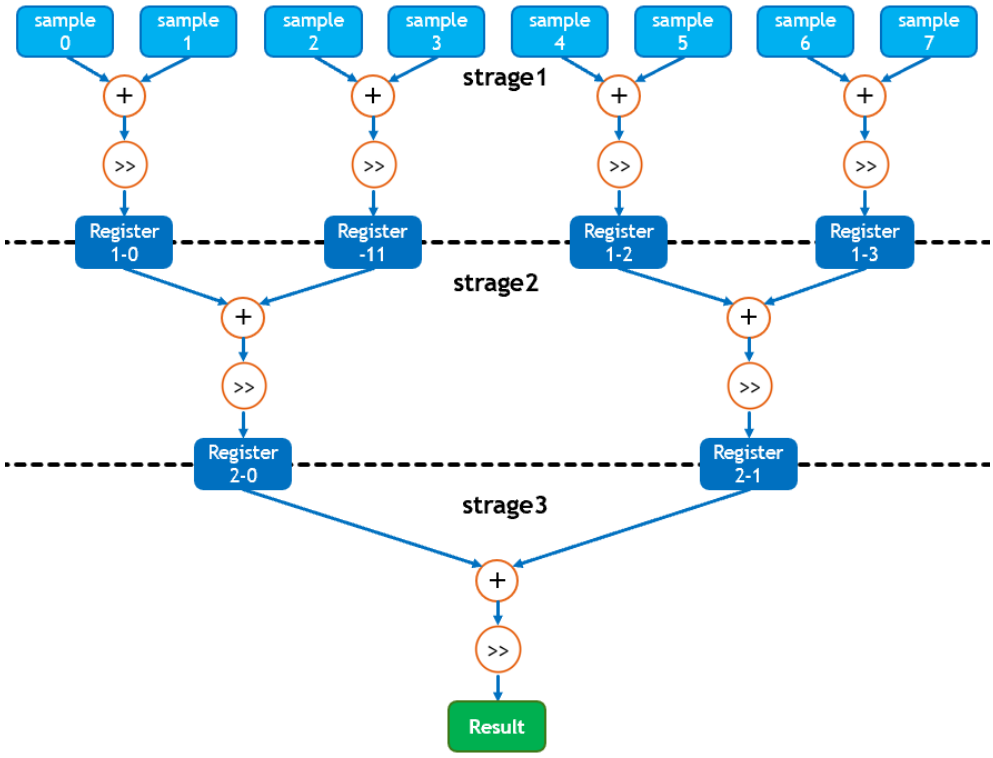


図 2.4: 離散ウェーブレット変換の回路図解

2.3.2 HiFP2.0 の問題点

HiFP2.0 は，処理の高速性を確保するために音楽データの先頭から約 2.97 秒分を固定的に使用する．よって，この約 2.97 秒間に著しい音質の劣化などの問題が発生した場合，楽曲の識別を正しく行うことが出来ない可能性が高い．楽曲の保存形式変換に合わせた音質劣化などは楽曲全体に影響を及ぼしうるが，楽曲によって変換による音質劣化が冒頭 2.97 秒間の部分に大きな影響を与える曲とそうでない曲が存在し，前者の識別は音質劣化によって難しくなる．

そこで，本研究では，サンプルデータを抽出する範囲を楽曲データ全体に拡大させることで，確率的に問題の発生した部分を回避することで，この問題を解決する．

その一方で，楽曲データ全体を使用するため，実行時間が悪化してしまうと想定される．本研究では，その悪化の程度も調査する．

2.3.3 FPGA について

2.3.4 FPGA の構成

本研究においては，提案手法を実装するハードウェアとして，FPGA を使用する．

FPGA（Field Programmable Gate Array）とは，ユーザが用途に合わせて任意の論理回路を設計し，それをもとに実際に回路の再構築を行うことができる集積回路である．

FPGA は，以下の要素から構成されている．

- 論理ブロック
  - LUT（Look up table）
  - セレクタ
  - フリップフロップ
- スイッチブロック
- コネクションブロック
- I/O ブロック
- 配線路

FPGA 上では論理ブロックが格子状に配置され，その間をマトリクススイッチ及び配線路が接続している．ユーザは，内蔵されている SRAM に作成した回路構成データをインプットし，FPGA に読み込ませる．FPGA は読み込んだデータを元にスイッチブロックの切り替えて構成要素同士を結線することで，任意の回路を構築する．

FPGA は固定的な回路を形成し，パイプライン化などによって処理の並列化を行うことで高速かつ低消費電力を実現できる．ただし，現状，規模の大きなアルゴリズムなどを実装するにあたってハードウェアリソース量不足の問題を抱える場合が多い．また，ASIC などと比較すると，初期設計コストが抑えられる利点がある一方で，ある程度以上に量産化が行える場合は総生産コストとしては ASIC に劣る部分がある．

2.3.5 FPGA のにおける回路設計

ユーザは FPGA にインプットする回路情報データを生成するために Verilog HDL や VHDL などの HDL（Hardware Description Language）あるいは C 言語などからの高位合成によって論理回路を記述する必要がある．（高位合成では C 言語などから HDL がコンパイルされる．）これを元に統合開発ツールを用いて論理合成，テクノロジマッピング，配置配線などを行い，回路構成データとして出力する．

2.4 まとめ

この章ではオーディオフィンガープリントについての詳しい説明とその生成に使用されるアルゴリズムについての説明，フィンガープリントに関する関連研究，特に HiFP2.0 についての説明と問題点を述べ，HiFP2.0 が実装を想定している FPGA についても説明した．次の章からは，この章で触れた HiFP2.0 の特性上の問題点を改良する本研究の提案とその実装について述べる．

第3章　HiFP2.1とその実装

3.1　はじめに

前章で述べた通り，HiFP2.0 には冒頭 2.97 秒間のサンプルデータを固定的に使用する特性上，その部分に問題が発生した場合に楽曲の識別が正しく行えない可能性が高いという問題点がある．

この章においては，その問題点の改良のため，本研究で提案する楽曲データ全体に一定量のサンプル抽出領域（chunk 領域）を周期的に分散配置し，そこからサンプルデータを抽出し FPID 生成することで確率的に問題発生部分を回避するアルゴリズムである HiFP2.1 の構造とその実装について述べる．

また，HiFP2.1 の実装においては，chunk 領域抽出をどのように行うのか，といった点について FPGA 上で行う形式とソフトウェアドライバ上で行う形式の 2 つがある．この決定は，HiFP2.1 実装の実行時間にも影響しうる．そこで，それぞれの利点や欠点について述べ，最適な形式の決定方法を提案する．それに加え，それぞれのパターンでのアルゴリズムがどのような形になるかを実際に示す．

また，抽出方法と同様に，HiFP2.1 の実装のためには chunk 領域の個数も決定しなければならない．HiFP2.1 が chunk 領域を楽曲データ全体に分散させて問題発生個所を確率的に避けるものである性質上，chunk 領域数は検索精度に影響する．そこで，chunk 領域数の最適な決定方法を提案する．

さらに，提案された chunk 領域抽出方法に基づく HiFP2.1 を実装する際のホスト PC および FPGA の構成と，ホスト PC-FPGA 間の通信方法，ホスト PC から FPGA へ入力されるサンプルデータの FPGA 上での取り扱い方法，FPID の生成方法について述べる．最後に，本研究での提案における HiFP2.1 のアルゴリズム上の各パラメータ設定について述べ，まとめる．

3.2　提案手法

HiFP2.0 は，楽曲データの冒頭 2.97 秒間を固定的に使用する特性上，その個所に問題が発生した場合，楽曲の識別を正しく行えない可能性が合うという問題がある．そこで本研究では，検索での使用時に実行時間が増加することを防ぐために生成される FPID のサイズは HiFP2.0 と同様としながらも，それに必要となるサンプルデータの抽出範囲を冒頭 2.97 秒間に固定するのではなく，楽曲全体に拡

大する．必要なサンプルデータを抽出する個所を複数の小さなサンプルデータ抽出領域（chunk 領域と呼ぶ）として分割，楽曲全体に周期的に配置することで，問題発生個所を確率的に回避する HiFP2.1 アルゴリズムを提案し，実装する．

これによって，著しい音質の劣化などの問題が発生している部分を確率的に回避することによって識別失敗率を低下させ，検索精度を向上させる．

そして，HiFP2.1 の実装時に決定しなければならない chunk 領域の抽出方法と最適な chunk 領域数を決定方法も同時に提案する．

また，この HiFP2.1 における HiFP2.0 と比べた場合，楽曲検索における精度の向上が見込まれるが，楽曲データ全体を使用する特性上，実行時間の悪化するものと考えられる．よって，その向上率及び悪化率を比較，調査する．

3.2.1 提案する HiFP2.1 のアルゴリズム

HiFP2.1 の構成は，次のようになっている．図 3.1 および図 3.2 に示すように，楽曲データ全体から一定サイズの連続的なサンプル群を周期的に抽出し，そのサンプルデータから FPID を生成する．抽出するサンプルデータ数は，検索に使用する場合に FPID のサイズが 4,096bit に固定されていると効率的に検索を行えるため，131,072 で固定する．ここで，一定領域から抽出するサンプルデータ群を「chunk 領域」，chunk 間の抽出しない読み飛ばす領域を「gap 領域」と呼称する．また，楽曲の前後 2 秒間は，その区間が無音である場合を考慮してサンプル抽出の領域に含めない．この領域を「discard 領域」と呼称する．また，対象とする楽曲データは 8 秒（サンプル数 352,800）以上とする．HiFP2.1 が楽曲全体から抽出領域のパラメータを決定するための式は以下のようになる．

$$length_{chunk_total} = length_{chunk} \times number_{chunk} = 131072 \tag{3.1}$$

$$length_{gap} = \frac{length_{wave_total} - length_{discard} - length_{chunk_total}}{number_{chunk}} \tag{3.2}$$

$$length_{chunk} = \frac{length_{chunk_total}}{number_{chunk}} \tag{3.3}$$

$$number_{chunk} = number_{gap} \tag{3.4}$$

ただし， (3.5)

$$number_{gap} = number_{chunk} = 2^n \tag{3.6}$$

かつ (3.7)

$$length_{wave_total} \geqq 352800 \tag{3.8}$$

ここで， (3.9)

$$1 \leqq n \leqq 11 (n \text{ は任意の自然数}) \tag{3.10}$$

ここで，図 3.2 でも示されている通り，$length_{chunk_total}$ は全 chunk 領域の全サン

プルデータの個数の合計，$length_{chunk}$ は個々の chunk 領域のサンプルデータの個数（以下，chunk サンプル数），$length_{gap}$ は個々の gap 領域のサンプルデータの個数（以下，gap サンプル数），$length_{wave_total}$ は音楽ファイルの全サンプルデータの個数，$length_{discard}$ は discard 領域のサンプルデータの個数（以下，discard サンプル数），$number_{chunk}$ は chunk 領域の個数（以下，chunk 領域数），$number_{gap}$ は gap 領域の個数（以下，gap 領域数）を指す．

また，これ以後，chunk 領域に属するサンプルデータを chunk サンプル，gap 領域に属するサンプルデータを gap サンプルと呼ぶ．

HiFP2.1 は，楽曲データファイルのヘッダから取り出した楽曲全体の長さの数値と，事前に決定しておいた chunk 領域数から自動的に gap 数と chunk サンプル数と gap サンプル数を算出する．

その後，図 3.2 で示されるように，その数値を元に楽曲データ全体からサンプルデータを抽出する．最終的に全ての chunk 領域で合計 131,072 のサンプルデータ（緑枠の青い長方形の集合）が抽出され，離散ウェーブレット変換のサブバンド分解処理が行われ（図の DWT 部分），8 サンプルから一つの特徴量が生成される（DWT 最下段の青い四角形）．それを 3 つ飛ばしで比較が行われ，その大小関係に 1/0 の対応付けを行う（オレンジの四角形）．その 1bit データは FPID の構成要素であり，これが 4,096bit 連結され，AFP の FPID（最下段の 1,0 の集合）が生成される．

3.2.2 実機実装の構成

3.2.2.1 提案する実機実装の構成

本研究の FPID 生成システムの内部構成図を図 3.3 に示す．提案手法では，ホスト PC 内に組み込まれた FPGA 上にシステムをインプリメンテーションし，ホスト PC 側から実装したソフトウェアドライバーを用いて FPGA のシステムと通信する．ホスト PC は PCIeExpress3.0 のインターフェイスを介して接続されている．ホスト PC はソフトウェアドライバ（図 3.3 の C++Driver）によって楽曲データを先頭から順に FPGA 側へ PCIexpress インターフェイスを介して送信する．（図 3.3 の 1）FPGA は受信した入力データから，HiFP2.1 のアルゴリズムを元に生成した回路によって chunk 領域のサンプルデータのみを抽出し FPID 生成処理を行う．（図 3.3 の 3-4）そして，FPGA は生成した FPID を同様のインターフェイスを通してホスト PC 側に送信する．（図 3.3 の 6）PCIe の制御とデータ通信の扱いについては，次節で詳しく扱う．

本研究における実装部分としては，これら PCIe の制御を既存の IP コア（Xillybus IP Core [30]）を用いて行う．その上で，それらと協調動作する HiFP 回路を実際に設計し，実装した．HiFP 回路については，HiFP2.0 における離散ウェーブレッ

ト変換のサブバンド分解およびFPIDビット生成のアルゴリズムは既存手法の概念を用いつつも，実際の実装におけるそれらのコードの設計およびHiFP2.1への拡張部分の設計は論理回路設計の訓練を目的として独自に行った．また，論理回路設計にあたって使用したハードウェア記述言語は記述量の少なさの観点からVerilog HDLを使用した．

3.2.2.2 PCIeとの通信

FPGAボードとホストPCはPCI Express3.0というI/Oシリアルインターフェイスを介して接続されている．その構成を引き続き図3.3を使って示す．PCI Expressの通信制御は，Xillybus Ltd.によるXillybus IP Core [30]（以下，Xillybus）を使用して行う．Xillybusは，PCIexpressによるDMAなどのデータ通信機能を統合的に提供するIPコアである．様々なFPGAプラットフォームに対応しており，Xilinx社のVirtex-7のPCI express Gen3に対応するIP CoreはPCIeレーンが最大で8xで，最高データレート6.6 GB/sである．Xillybusは，ユーザロジックとFIFOを介して双方向でデータをやり取りする．（図3.3の”FIFO(IN)”に伴う2-3および”FIFO(OUT)”に伴う4-5）この構成によって，ユーザロジックとPCI expressの通信制御を分離して設計が行えるようになっている．XillybusはOSの起動時にドライバーによってホストPCに認識される．これによってホストPCのメモリ空間にXillyus用のDMAバッファが割り当てられる．この領域にFPGAがアクセスすることでDMA通信が行われる．その上で，Xillybusはホスト PC側のdev領域に，ホストPC側でDMAバッファへデータを読み書きするためのデバイスファイルを自動生成する.（図3.3の”Write File”および”Read File”）ユーザ側は，このデバイスファイルに書き込み（図3.3の1），書き込み処理（図3.3の6）を行うことで，PCI expressでの通信を行うことが出来る．Xillybusは一種類のFPGAに対して複数のリージョンが用意されているが，今回の研究では最高データレート向けのリージョンXXLを使用した．

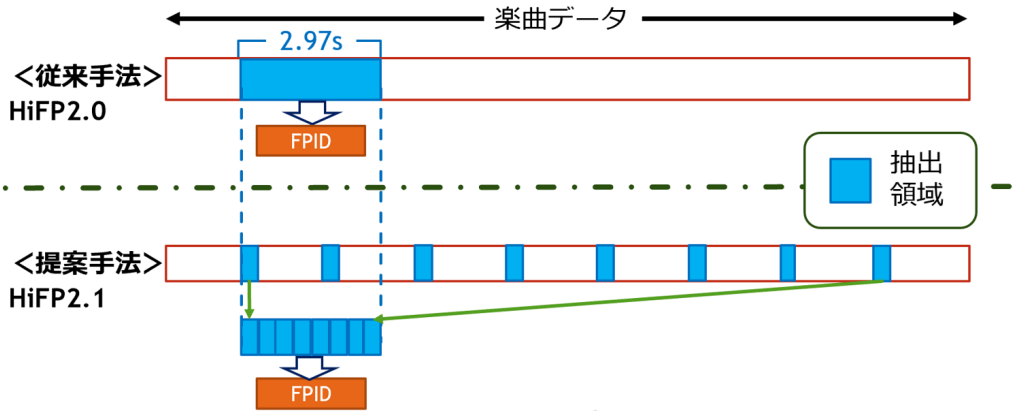


図 3.1: HiFP2.0 と HiFP2.1 の抽出領域比較図解

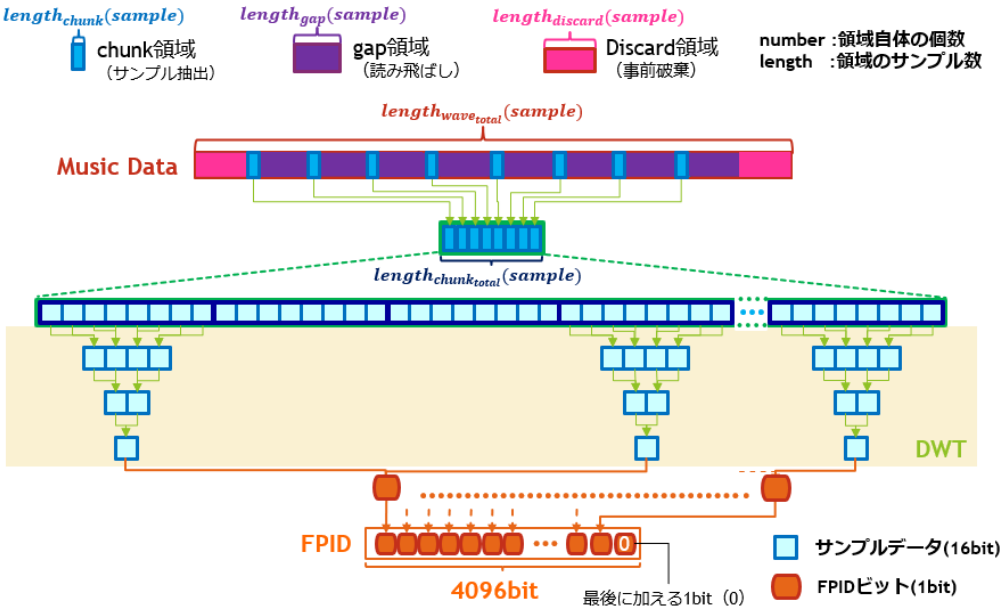
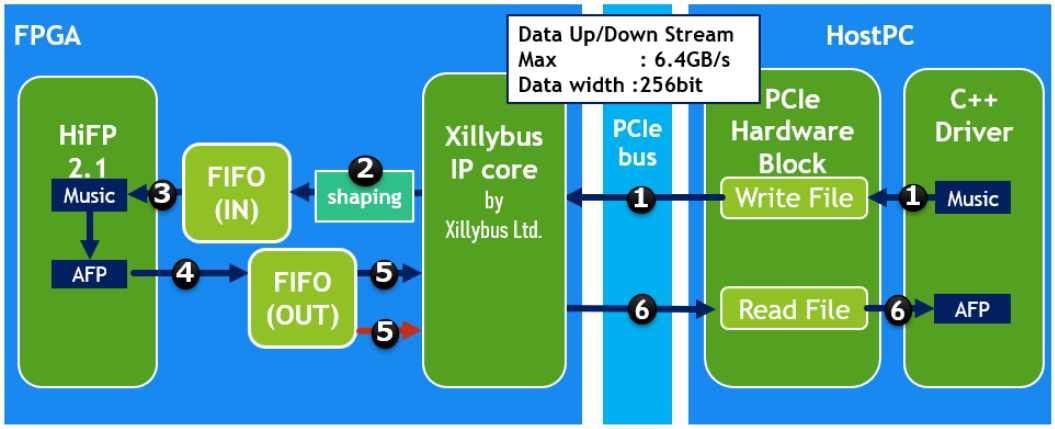


図 3.2: HiFP2.1 のアルゴリズム全体図図解



1. hostPC上でC++driverがXillybus IP CoreによるWriteファイルへ書き込み
2. XillybusがFIFO(IN)に受信したデータを256bitずつ逐次的に入力。
3. HiFP2.1が256bitずつデータを取り出して片耳分(128bit)を処理し、FPID(全4096bit)に。
4. FPIDを完成次第32bitずつFIFOに入力。
5. FIFO(OUT)の容量4096bitがfullになったら、Xillybusに通知。
6. hostPCはFIFO内のデータをReadファイルから書きだす。

図 3.3: ホストと FPGA の通信図解

3.2.2.3 実装の環境

この節で示したような実装にたいする実際の開発環境およびホスト PC，使用した FPGA ボードについて表 3.1，3.2 に示す.

項目	説明
CPU	Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz
メモリ	16GB
ストレージ	ATA WDC WD10EZEX-00B 1000GB
OS	Ubuntu 18.04.1 LST
コンパイラ (C++)	gcc 7.3.0
開発ツール (FPGA)	Vivado 2019.1

表 3.1: ホスト PC および開発環境の性能

項目	説明
製品名	Xilinx Virtex ®-7 FPGA VC709 コネクティビティ キット
FPGA	Vertex-7 XC7VX690T-2FFG1761C
ロジックセル	693,120
DSP スライス	3,600
メモリ (Kb)	52,920

表 3.2: FPGA ボードの特徴

3.2.3 chunk 領域抽出の実装形式の決定方法

3.2.3.1 chunk 領域抽出の実装形式のパターン

HiFP2.1 アルゴリズムを実機実装する場合，楽曲データ全体から周期的に一定量の chunk サンプルを抽出して，そこから FPID を生成する．この chunk 領域抽出をどこで行うかによって以下の 2 パターンの実装が考えられる．

- ソフトウェアドライバ上でサンプル抽出を行う FPGA 実装
- FPGA 上でサンプル抽出を行う FPGA 実装

”FPGA 上でサンプル抽出を行う FPGA 実装”は，FPID 生成に使用する chunk サンプルの抽出を FPGA 上の HiFP2.1 回路で行うパターンである．このパターンの実装では，ソフトウェアドライバは楽曲データを構成するサンプルデータの殆どを FPGA 側に送信する．そして，FPGA 内の HiFP2.1 回路はソフトウェアドライバ側から順次送信されてくるサンプルデータから chunk サンプルのみを後述のデータ位置補正を適応しながら抽出し，それを使って FPID を生成する．

”ソフトウェアドライバ上でサンプル抽出を行う FPGA 実装”は，ソフトウェアドライバ側が FPGA 側へデータを送信する前に chunk サンプルの抽出を行うパターンである．このパターンの実装では，ソフトウェアドライバは FPGA 側へサンプルデータを送信する前処理として楽曲データ全体から chunk サンプルを抽出して配列にまとめる．そして，まとめた全 chunk サンプルの合計 131,072 のサンプルデータを FPGA 側に送信し，FPGA はそれを使って FPID を生成する．FPID を生成する．この実装の場合，HiFP2.0 にはない HiFP2.1 固有の処理は全てソフトウェアドライバ側で行い，FPGA 側に送るデータ量も 131,072（2.97 秒間分）のサンプルデータになる．よって，FPGA が持つべき処理機能は HiFP2.0 と全く同じになるので，FPGA 上の実装としては HiFP2.0 と同じ回路を使用する．

これら 2 つの実装の全体図を図 3.4 内にそれぞれ示す．この図は，各実装形式におけるホスト PC-FPGA 間の通信処理と，それに伴うサンプルデータの流れを示したものである．中央の”hostPC”と”FPGA”の図形は実装に使用される実機を表しており，この 2 つは PCIexpress インターフェースで接続されている．”C++Driver”の領域は実機の”hostPC”に対応しており，ホスト PC 上でのソフトウェアドライバの楽曲データの処理について表している．一方．”FPGA”の領域は実機の”FPGA”に対応しており，FPGA 内部でのサンプルデータの処理を表している．”FPGA”内の”Sample buffer”は，DWT 処理にかける chunk サンプルを一時的に格納しておくためのバッファレジスタであり，”FPID”は生成された FPID を 4,096bit 分格納しておくバッファレジスタである．どちらの実装においても，楽曲データのサンプルデータの一部が”hostPC”から”FPGA”に向かって Stream 送信 (データ量:16bit × 16sample=256bit) され，”FPGA”側で左耳のデータのみ取り出されて 8 サンプルになり，”Sample buffer”に一時格納されたのちに DWT 処理をされて 1bit の FPID になって，”FPID”に格納されるまでは同一である．”FPID”に格納されるまでは

同一である．ただ，”FPGA 上でサンプル抽出を行う FPGA 実装 ”では ”chunk 領域抽出 ”は ”FPGA ”側で行われており，”ソフトウェアドライバ上でサンプル抽出を行う FPGA 実装 ”では，” C++Driver ”側で行われているという違いがあることが分かる．

また，”ソフトウェアドライバ上でサンプル抽出を行う FPGA 実装”は chunk 領域データのみを送信するため，後述の実データ位置補正による ”chunk 領域判定 ”による chunk 領域を選ぶ機能がない．また，この章の最後に，これら 2 つの実装形式をアルゴリズムとしたものを示す．本研究では，これらの実装形式のどちらが最適であるかについてその決定方法を提案する．

3.2.3.2 実装形式の決定

本研究の提案手法においては，HiFP2.1 アルゴリズムを実装する場合にはサンプルデータ抽出の実装形式を事前に決定しておかなくてはならない．

本研究では，実装時における実時間の実行速度に着目して，実装パターンの決定方法を提案する．ソフトウェアでの処理によって chunk サンプルの抽出を行う場合，chunk 領域数が 2 の累乗で増加するごとに chunk 領域がより楽曲データ全体に散在することになるため，chunk サンプルの抽出処理としての楽曲データが展開されたメモリへのアクセスと，サンプルデータを格納する処理の回数が指数関数的に増加すると考えられる．また，HiFP2.1 アルゴリズムにおいては，chunk 領域数を任意に設定できる．HiFP2.1 の chunk 領域を楽曲データ全体に分散させることで音質の劣化した部分を確率的に回避するという性質上，基本的には chunk 領域に大きな数を取って chunk 領域をより細分化して分散させることで劣化した楽曲データにおける AFP による検索精度は向上すると考えられる．その上で，”FPGA 上でサンプル抽出を行う FPGA 実装”においては，FPGA 側に送信するサンプルデータは，楽曲データ全体中の最後尾の chunk 領域末尾のサンプルデータまでであり，楽曲データ全体に対して gap サンプル数 1 つ分を送信しない形になる．chunk 領域数が増加すると gap 領域数も増加するが，楽曲データ全体のサンプルデータの数は一定であるために，gap サンプル数は細分化され小さくなる．後述するように chunk 領域数は 2 の累乗の数値となるが，chunk 領域の 2 の累乗で増加するごとに，送信しなくてよい末尾の gap 領域のサンプル数は前の段階の半分になる．よって，chunk 数増加に伴う送信するサンプルデータの数の増加率は対数関数的になり，chunk 領域数が増加するにつれて処理時間の増加率は緩やかとなるため，大きな差が出ないと考えられる．

これらの事柄を踏まえて，本研究では HiFP2.1 アルゴリズムの実装形式として，可能な限りメモリへのアクセスおよびサンプルデータの抽出処理の回数が少なくなり，なおかつ chunk 領域数が増加しても処理速度に大きく変化しない「FPGA 上でサンプル抽出を行う FPGA 実装」を提案する．

3.2.4 最適な chunk 領域数の決定方法

3.2.4.1 最適な chunk 領域数のパターン

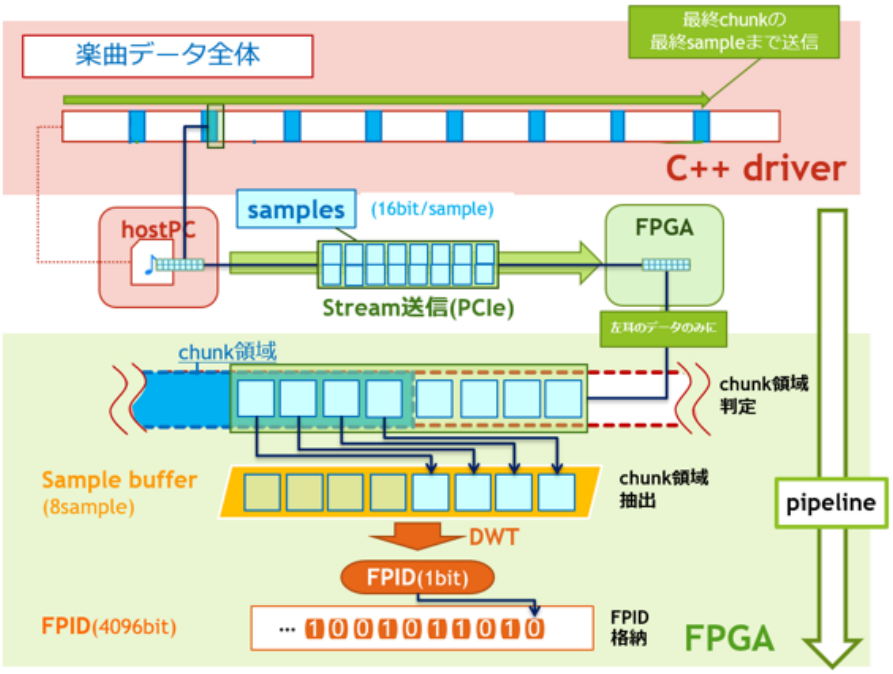
前述したとおり，実装する HiFP2.1 においては，FPID 生成に使用するサンプルを抽出する chunk 領域数を任意に設定することが出来る．

ただし，FPID 生成に使用するサンプル数は 131,072 で固定されており，131,072 は 2 の累乗である．このことを前提として，ハードウェアに実装された場合に chunk サンプル数などを計算する際に 131,072 を各 chunk 領域に均等に割り振るため，chunk 領域数も 2 の累乗である必要がある．また，その上で局所的な特徴量の変化傾向を 1 つの chunk 領域の範囲からが少なくとも FPID1bit は取得するために 1 つの chunk 領域には少なくとも 8*4*2=64 サンプルが含まれる必要がある．その上で，特徴量の大小比較は 3 つ飛ばしで行われる．よって，chunk 領域数は最大でも 131,072/64=2,048 となる．

であるため，各型式の実装における chunk 領域数のパターンは 2，4，8，16，32，64，128，256，512，1,024，2,048 の 11 通りになる．

図 3.5 にそのバリエーションについて示す．

FPGA 上でサンプル抽出を行う FPGA 実装



FPGA 上でサンプル抽出を行う FPGA 実装

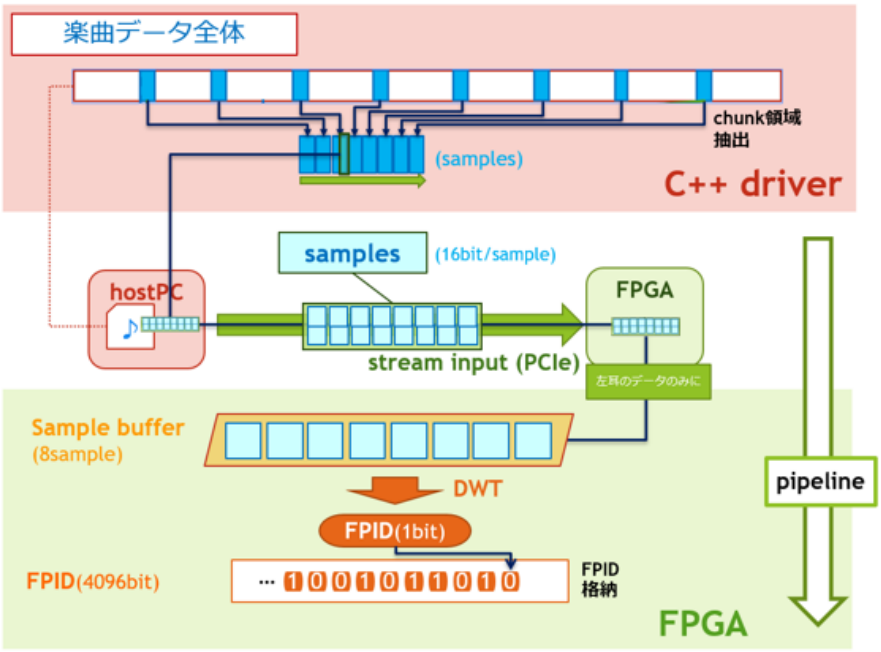


図 3.4: サンプル抽出タイミングの違いによる 2 つの実装全体図図解

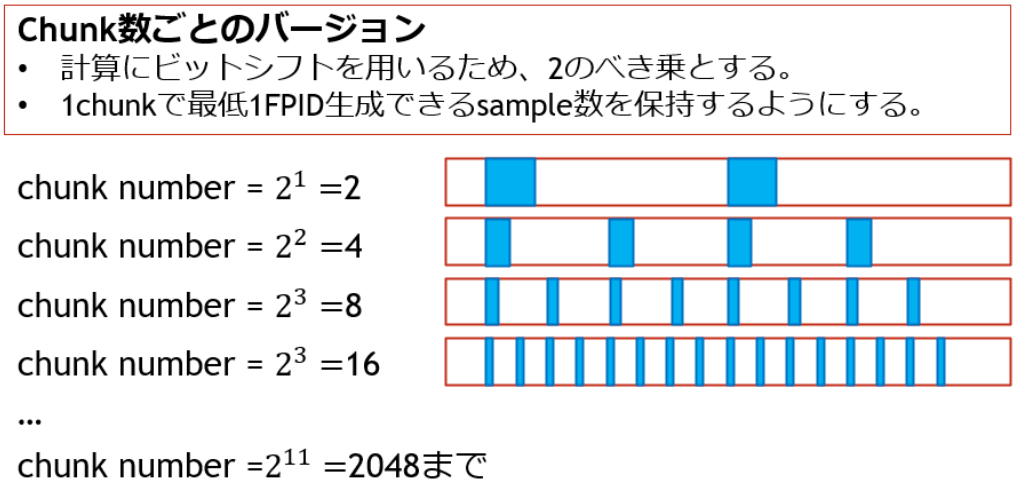


図 3.5: チャンク領域数のパターンの図解

3.2.4.2　最適な chunk 領域数の決定

本提案手法は楽曲の広範囲から FPID 生成のためのサンプルを抽出することで確率的に問題が発生しているサンプルの領域を回避するものである．そして，chunk 領域数の決定は，一つの chunk 領域でどれだけのサンプルを抽出を行うか，つまり範囲全体における chunk 領域の分散の度合いの決定でもある．であるので，chunk 領域数は，そのサンプルから生成した FPID による検索精度に影響する．

また，本提案手法である”FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装”では，FPGA 側に送信するサンプルはデータ全体の中の最後の chunk 領域の末尾までである．よって，chunk 領域数は実行速度に影響する．

ただ，前述したように「FPGA 上でサンプル抽出を行う FPGA 実装」においては，chunk 領域数の増加による処理時間の増加は対数関数的であると考えられる．であるので，本研究においては，最適な chunk 領域数の決定方法として可能な限り大きな chunk 領域数をとることとする．

ここで，chunk 領域数 2,048 の場合は，1 つの chunk 領域において DWT の結果算出される特徴量は 2 つであり，よって，１つの chunk 領域内のみで生成される FPID は 1bit のみとなる．つまり，2,048 の場合には 1 つの chunk 領域内部という近い位置の特徴量間で生成される FPID は 2,048bit となる．つまり，残りの 2,048bit は必ず離れた位置で生成された特徴量間の大小比較を行わなければならない．

楽曲データに著し音質の劣化などが発生した場合，近い位置の特徴量間の差異は，それが極めて短い時間（2 特徴量間はサンプリング周波数 44,100Hz の場合 1.45ms 程度）における特徴量の変化を表しているために，劣化しても特徴量の変化傾向を保持している可能性が高い．つまり，劣化前の特徴量の大小関係を保持している可能性が高い．一方で，離れた位置の特徴量間ではそれぞれのデータ同士の関連性が薄くなるために，近い位置の特徴量間と比べ劣化前の大小関係を正しく保持できている可能性が低くなる．

chunk 領域数 2,048 の場合は，離れた位置の特徴量間から生成される FPID の割合が 50%を越えている．50%という数値は，異なる楽曲同士の FPID 間で起こるビットエラー割合の大まかな平均値と同値である．特徴量間の大小関係を正しく保持できている可能性が低い FPID のビットの割合が 50%を越えている chunk 領域数 2,048 は最適な chunk 領域数とは言い難い．

よって，chunk 領域数を決定する際には，必ず連続した特徴量を生成できるようにすることで音質の劣化に対して大小関係を保持できる可能性の高い部分をある程度確保しつつ，広範囲に chunk 領域を分散させることで問題発生部分を出来る限り高い確率で回避できるようなバージョンを選ぶとよい．

これらの事柄を踏まえて，本研究では HiFP2.1 アルゴリズムの実装における最適な chunk 領域数として 1,024 を提案する．

3.2.5　実機における処理フロー

提案手法を FPGA に実装するにあたって，HiFP2.1 の回路としては図 3.6，図 3.7 に示すような処理を行うように回路を構築した．

図 3.6 は，HiFP2.1 実装におけるホスト PC と FPGA 間の処理とサンプルデータの流れについての図であり，図 3.7 は HiFP2.1 回路の動作フローと計算処理，判定処理にフォーカスした図である．

図 3.3 および図 3.6 で示されたように ホスト PC がソフトウェアドライバによって FPGA 内の Xillybus を介して入力 FIFO へ送信したデータ（図 3.6 の C++Driver 部分および stream input(PCIe)）は 16 サンプル（256bit）ずつ HiFP2.1 の回路へと送られ，その内，左耳の分の 8 サンプル（128bit）が実際の処理に使用される．これは，1 つ分の特徴量が生成されるサンプル数である．本実装では，図 3.7 で示すように，wav ファイルのヘッダに含まれている情報も FPGA 側で読み取る仕様になっている．まず，FIFO に入力されるデータから wav ファイルのヘッダに含まれるファイルの先頭を示す数値である “ RIFF ” (0x52494646) を読み取った時，さらにその後に続くファイルサイズを読み取り（図 3.7 の”ヘッダ読み込み”）その値から chunk 領域数および Gap 領域数を算出し（図 3.7 の”各種パラメータ計算”），HiFP2.1 の回路全体に起動信号を伝達する（図 3.7 の”スタートトリガ”）．回路内では入力サンプルのデータ全体での開始，終了位置を逐一計算しており，（図 3.7 の”chunk 領域位置計算”）入力された 8 サンプルが chunk 領域を含んでいるか否かを判定する（図 3.7 の”chunk 領域判定”および図 3.6 の”chunk 領域判定”）．chunk 領域を含んでいると判定した場合は，後述の抽出サンプル選択処理を行い，DWT 処理を行う chunk 領域サンプルを一時的に保持しておくための Sample buffer に格納される．（図 3.7 の”chunk 領域抽出”および図 3.6 の”chunk 領域抽出”）更に，8 サンプルから生成される特徴量を用いた FPID 生成における比較処理では，特徴量を 3 つ飛ばし毎に使用するため，現在取り扱っている特徴量が実際に ID 生成に使用されるかどうかを判定し（図 3.7 の”変換チャンク判定”），適切な特徴量のみを次のフローに送る．そして，前後の特徴量の大小比較を行い，その大小に 0 と 1 を対応付けることによって 1bit 分の FPID を生成する（図 3.7 の”ID 生成”および図 3.6 の”DWT””FPID(1bit)”）．生成された FPID は 32bit ずつ出力 FIFO に格納され，順次 Xillybus を通して HostPC へと取り出される（図 3.6 の”FPID 格納”および図 3.7 の”FPID 出力”）．FPID が 4,096bit 生成されると，終了信号が伝達され，HiFP 回路が停止，初期化される．これらの工程のうち，処理位置計算から FPID 出力まではパイプライン処理が行われ，逐次的に入力データが処理されるようになっている．

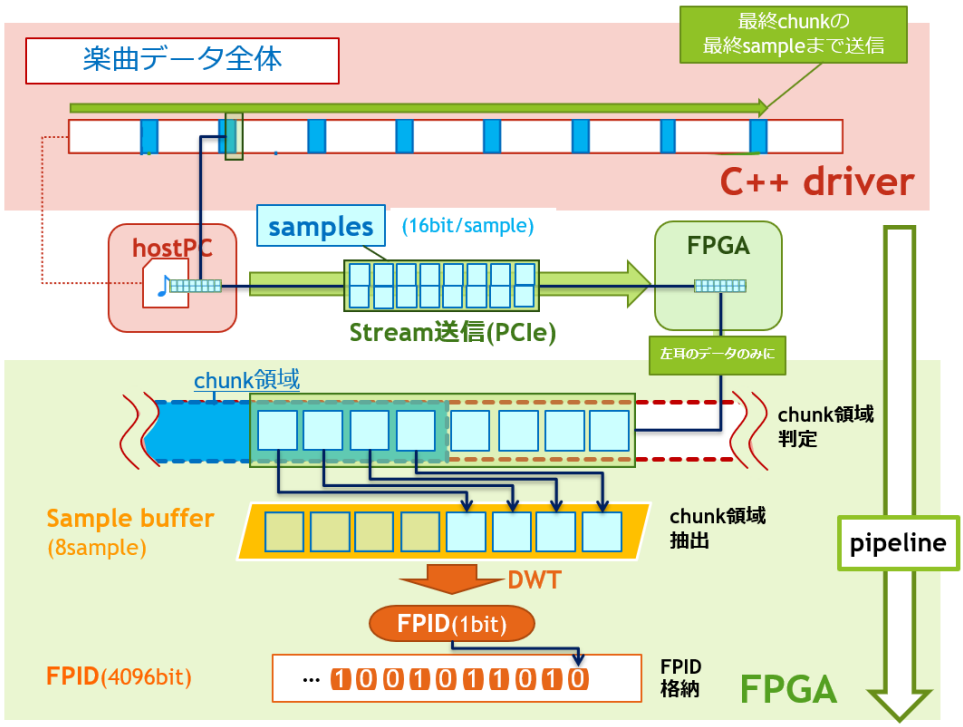


図 3.6: 提案手法（HiFP2.1）の実装全体図図解

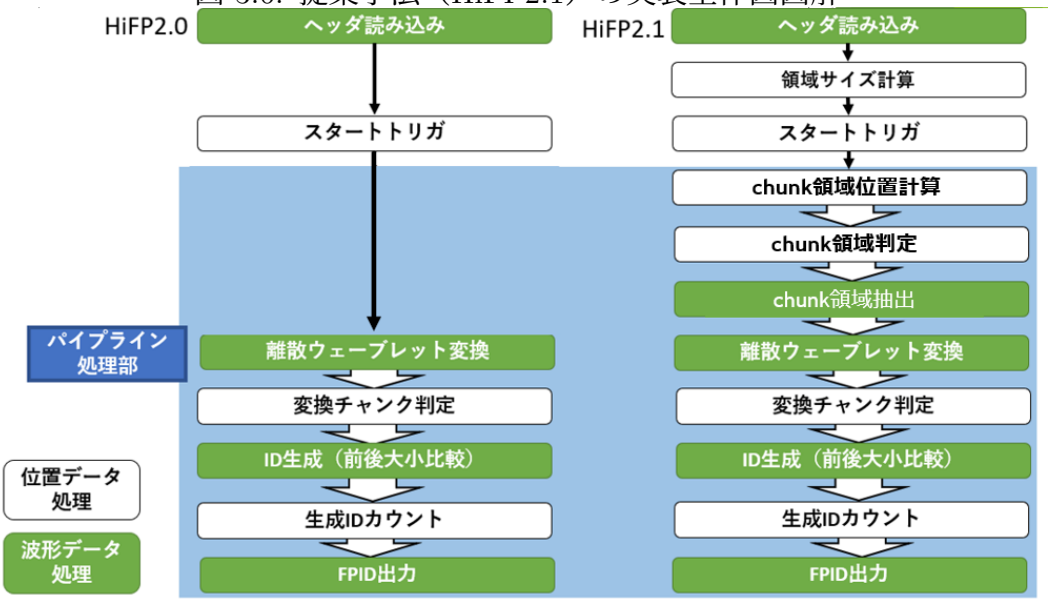


図 3.7: HiFP2.0 および提案手法（HiFP2.1）の実装上の動作フロー図

3.2.6　抽出サンプル選択処理

FPGA とホスト PC は，Xillybus IP Core と PCI express を通じて通信を行う．この時，リージョン XXL は 256bit 単位でデータ通信を行う．であるため，HiFP2.1 の回路も 256bit 単位でデータを処理する．これは，HiFP2.0 および HiFP2.1 が 1bit の FPID を生成するために必要なサンプル数は 1 サンプル 16bit が左右からの音声各 8 サンプル，合計 16 サンプル，256bit であるため，これは一度のデータ通信によって FPID1bit 分のサンプルを FIFO に格納できる計算になり，ホスト PC と FPGA 間での協調動作させるために都合がよいと言える．しかし，提案手法における実装の HiFP2.0，および HiFP2.1 は共に 256 で割り切れない bit 数のヘッダデータを扱う．また，HiFP2.1 においては chunk 領域間の gap 領域のサンプル数は可変長であるため，256bit 単位でデータを取得できても，その全てが FPID 生成に使用されるサンプルであるとは限らない．そこで，本研究では可能な限り少ないクロックで受信した 256bit（8 サンプル）のデータの chunk 領域と gap 領域の判別を行い，chunk 領域のサンプルのみを選択的に抽出するアーキテクチャを設計した．

このアーキテクチャは 8 サンプル分の入力バッファ，8 サンプル分の位置補正バッファ，chunk 領域判定ロジックを持っている．Xillybus から送信されてきた入力データを入力バッファがまず保持し，位置補正バッファはそれに 1 クロック遅れてデータを保持する．よって，これら 2 つのバッファは時間差で合計 2 クロックサイクル分の入力データを保持できる．1bit の FPID の生成に必要なサンプル数は 8 サンプルで各クロックサイクルごとに入力されるサンプル量も 8 サンプルであるので，まだ抽出されていない FPID1bit 分の chunk 領域内サンプルはこの 2 つのバッファ上に 1 度以上まとまって出現することになる．3.8 chunk 領域判別ロジックは，2 つのバッファがまだ抽出されていない FPID1bit 分以上の chunk 領域内サンプルを保持しているかどうかを毎クロックサイクルごとに判定し，含んでいる場合はそれを離散 Haar ウェーブレット変換回路に転送する．

そのロジックを以下に詳しく記述する．まず，判定ロジックが，その入力データにおいてどのサンプルデータが chunk 領域であるのかを識別し，抽出するサンプルデータの選択を行う．入力データが持ちうる chunk 領域のサンプル数は最大で 16，そのうち左のサンプルのみを扱うので 8 組である．そのパターンは,「入力データ全体が chunk 領域を含まない」「入力データ全体が chunk 領域である」「入力データの途中に chunk 領域の開始地点がある」「入力データの途中に chunk 領域の終了地点がある」の 4 パターンなので，そのいずれであるかを判定して，chunk 領域のサンプルデータのみをレジスタバッファに保存するようになっている．

その実装は，図 3.9 である．入力バッファレジスタには入力 8 サンプルの各サンプルごとに今回のクロックサイクルで DWT に使用する chunk 領域サンプルのみを右詰めにシフトさせるためのセレクタが配置されている．それらセレクタによって，補助バッファレジスタに残っている chunk 領域サンプルと合わせて 8 サンプル分の chunk 領域サンプルを確保できる必要個数分のみが右詰めにされた形にな

るようにセレクタを介した該当部分の配線がデータを出力する．今回のクロックで使用しなかった chunk 領域サンプルの個数は記録され，次のクロックまで保持される．それと同時に，現在の入力バッファレジスタのデータは次のクロックで使用するために補助バッファレジスタに受け渡される．位置補正バッファレジスタでは 1 クロックサイクル前に入力された 8 サンプルが格納されており，各サンプルごとに，chunk 領域のサンプルのみを左詰めにシフトさせるためのセレクタが配置されている．位置補正バッファレジスタでは 1 クロック前に DWT に使用されなかった chunk 領域サンプルのみが右詰めにされた形になるようにセレクタを介した該当部分の配線がデータを出力する．chunk 領域ではないサンプルデータ部分にはそれぞれのレジスタにおいて 0 が出力されるようになっており，2 つのレジスタのそれぞれのサンプルごとに足し合わせて一度レジスタに保存した後，離散 Haar ウェーブレット変換回路に出力する．

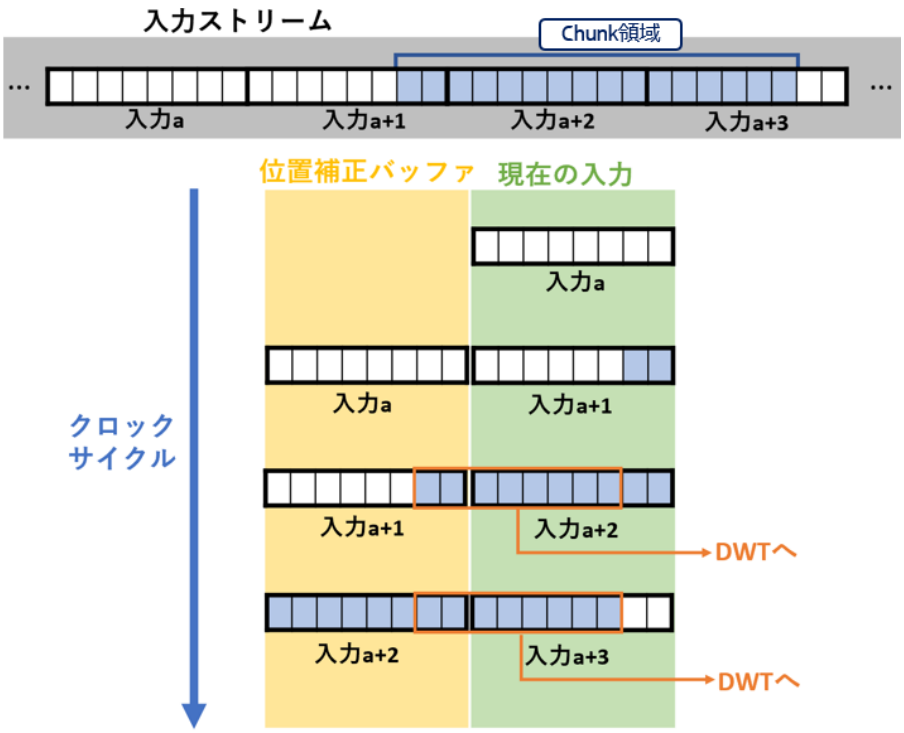


図 3.8: 抽出サンプル選択のイメージ図

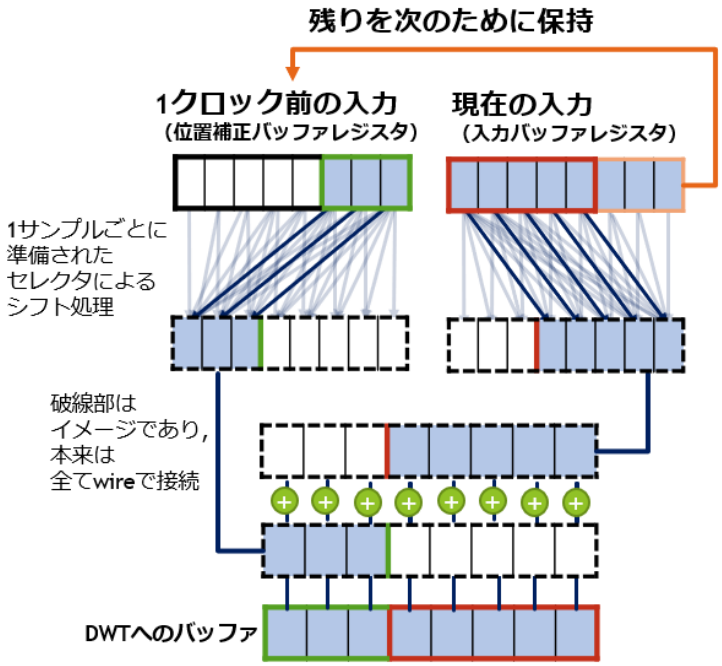


図 3.9: 抽出サンプル選択の実装のイメージ図

3.2.7　離散 Haar ウェーブレット変換回路

フィンガープリント生成のための離散 Haar ウェーブレット変換の回路はパイプライン処理によって行う．図に示す通り隣り合うサンプルを 2 つ 1 組として加算平均処理を行うことで，サンプル数が半分になる．この処理を 3 段階経ることで，8 サンプルの入力が 1 サンプル分の特徴量に変換される．その後，隣り合う特徴量同士の差分平均を算出し，その値の正負で FPID の 0/1 を決定する．HiFP2.0 においては，本研究での PCIe express での通信単位であるサンプル左右 8 組（256bit）において先頭から 8 サンプルを抽出し，隣り合うサンプル同士の 3 段階の加算平均，ビットシフト処理を経て，1bit の FPID となる．生成された FPID は，32bit が揃った段階で Xillybus の FIFO へ送られ，HostPC 側から 256bit ずつ取得される．

3.2.8　HiFP2.1 の実装全体のフロー

提案手法の FPGA への実装の全体像をブロック図として示すと図 3.10 のようになる．図 3.10 では，主にホスト PC から入力されたサンプルデータが FPID となってホスト PC へと送られていくフローを，それを制御するモジュールと信号とともに示している．

まず，”From HostPC”としてホスト PC から”FIFO(IN)”に入力されたデータのうち，ヘッダデータにあたる部分のみ”Read Header”モジュールに読み込まれ，内部のデータ長のパラメータが取り出される．そして，そのデータ長を用いて”Cal Length”モジュールにて各領域のパラメータが算出される．この各領域のパラメータは，”Cal Position”モジュールでの chunk 領域の位置計算などに使用される．次に，”Cal Position”モジュールと”Apply Buffer”モジュール”の二つを用いて抽出サンプル選択処理が行われている．”Cal Position”モジュールでは，図 3.11 で示されるように，入力データの 8 サンプルが楽曲データ全体のどの位置にあたるかをサンプル列の先頭座標（top_input）と末尾座標（bottom_input）で管理している（”Set Input”）．また，内包するサンプル全ての抽出が未完了の chunk 領域のうちの先頭のものに対して．入力データのサンプルが幾つ重なっているか，という値を算出しており（”Set Shift”），”Apply Buffer”モジュールでは，その値から実際の入力バッファレジスタと位置補正バッファレジスタそれぞれにおけるシフト数を算出している．”Combination and DWT”モジュールでは，”Apply Buffer”モジュールで決まった各シフト数を元に，入力バッファレジスタのサンプルのシフト（”Adjustment Currernt Input”）と位置補正バッファレジスタのサンプルのシフト（”Adjustment Previous Input”）を行い，それらを組み合わせる（”Combinaton Current and Previous Input”）．また，現在の入力サンプルで位置補正バッファレジスタを更新する．これにより揃った 8 サンプルを使用して，3 段階の離散ウェーブレット変換のサブバンド分解を行う（”DWT1-3”）．

”Cnt Reg”モジュールでは，”Cal Position”モジュールでの入力サンプル列の座標の進行に合わせて”Cnt Reg 4”カウンタを進行させる．これは DWT で生成される特徴量のうち 3 つ飛ばしのもの同士で比較処理を行うために，現在”Combination and DWT”モジュール”から出力されている特徴量が比較処理に使うものかどうかを判別するための 2it カウンタであり，2b’01 ずつ増加していき，カウンタが 2b’00 の時の特徴量同士を”Feature Extraction”モジュールで比較処理して，その大小に 1/0 のビットを割り当て，FPID ビットとして出力する．また，”Cnt Reg 32”カウンタは，”Cnt Reg 4”カウンタが一周するごとに 3b’001 ずつ増加する 3bit カウンタであり，”FIFO（OUT）”に入力する FPID ビットが 32bit 揃ったタイミングを得るためのものである．この信号によって，”FIFO（OUT）”への FPID ビットの入力が行われる．また，”Cnt Reg 4,096”カウンタは，”Cnt Reg 32”カウンタが一周するごとに 8b’00000001 ずつ増加する 8bit カウンタであり，生成された FPID ビットが 4,096bit 揃ったタイミングを得るためのものである．この信号と”FIFO（OUT）”の中身が全てホスト PC へと送信され空になった信号が揃った場合，全てのモジュールがリセットされる．

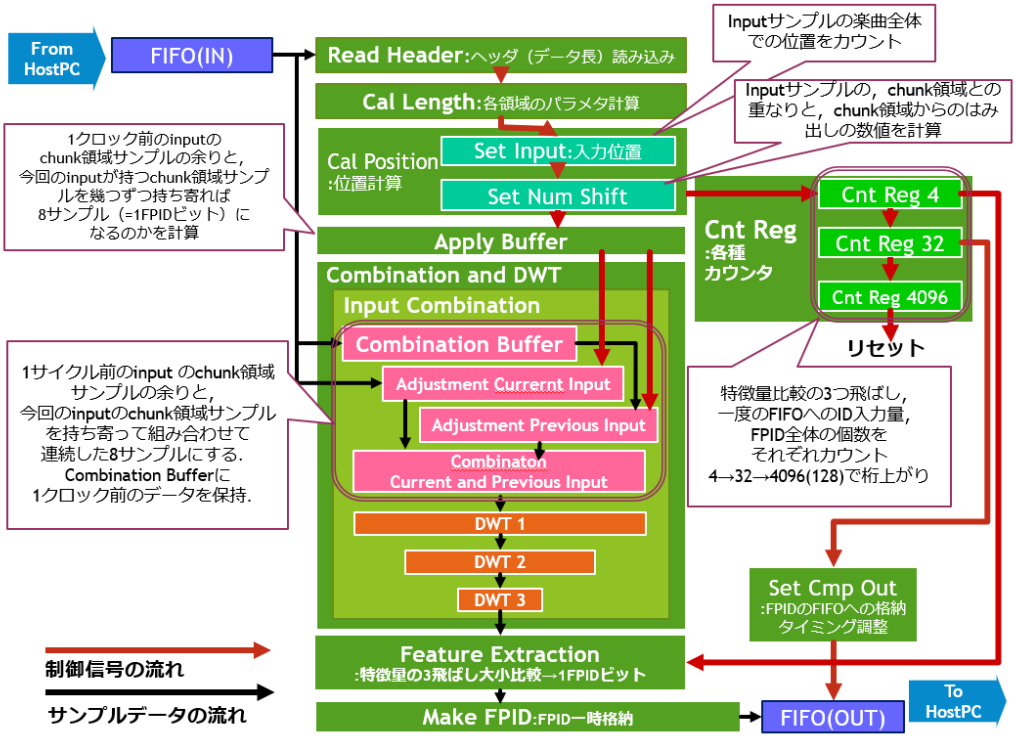


図 3.10: HiFP2.1 の実装全体のブロック図

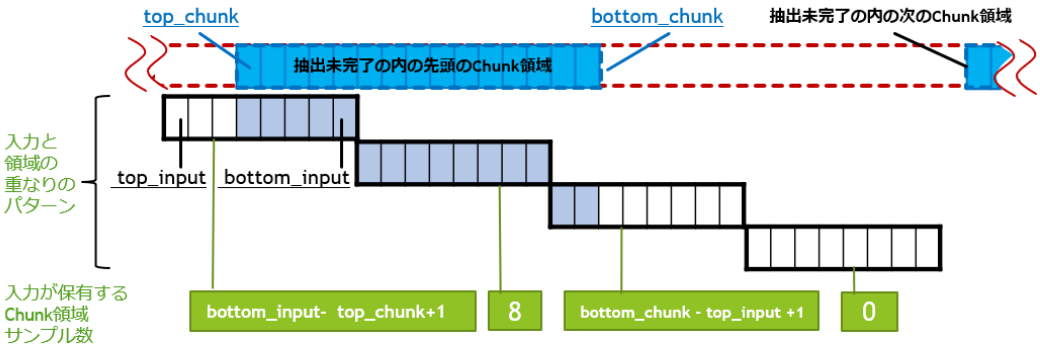


図 3.11: HiFP2.1 の Cal Position モジュールでの位置計算の図

3.2.9 HiFP2.1 のアルゴリズムの疑似コード

最後に，サンプル抽出方法についての 2 つの実装形式をアルゴリズムとして表す．それは，それぞれ次のようになる．

アルゴリズム 3.1 HiFP2.1 の FPGA 上でサンプル抽出を行う FPGA 実装のアルゴリズム

Input: $wav[]//wav$ 楽曲の右耳分の波形データ $(16bit/sample)$
Output: $AFP(4,096bit)//FPID$

1: $length\_wave\_total \leftarrow PCM$ データサンプル数;
2: $number\_chunk \leftarrow chunk$ 領域数;
3: $length\_chunk\_total \leftarrow 131,072$;
4: $length\_discard \leftarrow 88,200$;
5: $length\_chunk \leftarrow \frac{length\_chunk\_total}{number\_chunk}$
6: $length\_gap \leftarrow \frac{length\_wave\_total - length\_chunk\_total - length\_discard*2}{number\_chunk}$
　　$//$各種パラメータの計算
7: $top\_input \leftarrow 0, bottom\_input \leftarrow 7$;
　　$//$入力データの楽曲データ上の先頭，末尾座標
8: $num\_hold \leftarrow 0, num\_hold\_sub \leftarrow 0$;
　　$//$各バッファのデータ保持数
9: $sum\_chunk\_gap \leftarrow length\_chunk + length\_gap$;
　　$//chunk$ 領域数と gap 数の合計値
10: $top\_chunk \leftarrow 0, bottom\_chunk \leftarrow length\_chunk$;
　　$//$抽出未完了なものの中で先頭の $chunk$ 領域（ここでは $now - chunk$ と記述）の先頭，末尾座標
11: $buffer[]$;
　　$//$現在入力されているサンプル中の $now - chunk$ サンプルを保持
12: $buffer\_sub[]$;
　　$//$前回入力されたサンプル中の $now - chunk$ サンプルを保持
13: $buffer\_sum[]$;
　　$//buffer[]$ と $buffer\_sub[]$ の組み合わせを保持
14: $buf\_hold \leftarrow 0, buf\_sub\_hold \leftarrow 0$;
　　$//$各バッファのデータ保持数
15: $next\_flag \leftarrow 0$;
　　$//now - chunk$ の抽出が完了したフラグ
16: $flag\_buf \leftarrow 1$;
　　$//DWT$ 用のサンプルがそろったフラグ
17: $afp\_i \leftarrow 0$;
　　$//$現在生成している AFP の番号

18: **while** $bottom\_chunk < length\_wave\_total$ **do**
19: $next\_top\_chunk \leftarrow top\_chunk + sum\_chunk\_gap$;
//次の $now - chunk$ の先頭座標
20: $buf\_hold \leftarrow 0$;
/ * ここから 8 サンプル分を入力データとして扱う. 1 サンプルずつ処理し，入力の位置ずれも補正しながら $now - chunk$ のサンプルを抽出 * /
21: **for** $i \leftarrow 0; i < 8; i++$ **do**
22: **if** $top\_chunk <= bottom\_input \&\& top\_input < top\_chunk \&\& top\_chunk <= top\_input + i$ **then**
23: $buffer[i] \leftarrow wav[top\_input + i]$;
24: buf_hold++;
//入力 8 サンプルの後半の一部が $now - chunk$ に含まれる場合，現在処理しているのがそのサンプルデータなら抽出
25: **else** $top\_chunk <= top\_input \&\& bottom\_input <= bottom\_chunk$
26: $buffer[i] \leftarrow wav[top\_input + i]$;
27: buf_hold++;
//入力 8 サンプル全てが $now - chunk$ に含まれる場合，現在処理しているのがそのサンプルデータなら抽出
28: **else** $top\_input <= bottom\_chunk \&\& bottom\_chunk < bottom\_input \&\& top\_input + i <= bottom\_chunk$
29: $buffer[i] \leftarrow wav[top\_input + i]$;
30: buf_hold++;
31: $next\_flag \leftarrow 1$;
//入力 8 サンプルの前半の一部が $now - chunk$ に含まれる場合，現在処理しているのがそのサンプルデータなら抽出. $now - chunk$ の末端を入力データが扱っているということであるので抽出完了とし $now - chunk$ の対象を次に移す
32: **else** $bottom\_chunk < top\_input \&\& bottom\_input < next\_top\_chunk$
33: $buffer[i] \leftarrow 0$;
34: $next\_flag \leftarrow 1$;
//入力データ全体が gap 領域内にあり, $now - chunk$ より入力データの座標が進んでいる場合，現在の $now - chunk$ を抽出完了とし, $now - chunk$ の対象を次に移す
35: **else**
36: $buffer[i] \leftarrow 0$;
37: **end if**
38: **end for**
39: **if** $buf\_sub\_hold > 0$ **then**
40: **for** $i \leftarrow 0; i < buf\_sub\_hold; i++$ **do**

```
41:         buffer_sum[i] ← buffer_sub[(8 − buf_sub_hold) + i];
42:       end for
43:       for i ← 0; i < 8 − buf_hold; i++ do
44:         buffer_sum[(8 − buf_hold) + i] ← buffer[i];
45:         buf_hold − −;
46:       end for
47:       flag_buf ← 1;
48:     else      buf_hold == 8
49:       for i ← 0; i < 8; i++ do
50:         buffer_sum[i] ← buffer[i];
51:       end for
52:       flag_buf ← 1;
53:     end if
54:     if buf_hold > 0 then
55:       for i ← 8 − buf_hold; i < 8; i++ do
56:         buffer_sub[i] ← buffer[i];
57:         buf_hold − −;
58:       end for
59:       for i ← 0; i < 8; i++ do
60:         buffer_sub[i] ← 0;
61:       end for
62:       buf_sub_hold ← buf_hold;
63:     else
64:       buf_sub_hold ← 0;
65:     end if
66:     if  flag_buf == 1 then
67:       hi, lo ← DWT_HiFP21(buffer_sum[]);
68:       if lo − lo_buf < 0 then
69:         AFP[afp_i] = 1;
70:       else
71:         AFP[afp_i] = 0;
72:       end if
73:       lo_buf ← Lo;
74:       afp_i++;
75:       flag_buf ← 0;
76:     end if
77:     if next_flag == 1 then
78:       top_chunk ← top_chunk + sum_chunk_gap;
79:       bottom_chunk ← bottom_chunk + sum_chunk_gap;
```

```
80:       next_flag ← 0;
81:    end if
82:    top_input ← top_input + 8;
83:    bottom_input ← bottom_input + 8;
84: end while
85: AFP[0] = 0;
86: returnAFP;
```

アルゴリズム 3.2 DWT_HiFP21

```
   Input: wav[]
   Output: hi, lo
 1: n ← 8;
 2: while n > 1 do
 3:    n = n/2;
 4:    for i ← 0; i < n; i+ = 1 do
 5:       Hi[] = (wav[2 × i] − wav[2 × i + 1])/2;
 6:       Lo[] = (wav2[2 × i] + wav[2 × i + 1])/2;
 7:    end for
 8:    wav[] ← Lo[];
 9: end while
10: hi ← Hi[], lo ← Lo[];
11: returnhi, lo;
```

アルゴリズム 3.3 GAFP

```
   Input: Lo[], m
   Output: AFP(m)
 1: for i ← 0; i < m − 4; i+ = 4 do
 2:    temp = Lo[i] − Lo[i + 1];
 3:    if temp > 0 then
 4:       AFP[i] = 1;
 5:    else
 6:       AFP[i] = 0;
 7:    end if
 8: end for
 9: AFP[m/4 − 1] = 0;
10: returnAFP;
```

3.3 まとめ

この章では，HiFP2.0 における FPID 生成時の使用サンプルデータの局所性に基づく識別率低下の問題を解決するため，楽曲データ全体に chunk 領域を周期的に分散配置し，そこからサンプルデータを抽出し FPID 生成することで確率的に問題発生部分を回避するアルゴリズムである HiFP2.1 を提案し，その実装について述べた．

また，HiFP2.1 を実装では，chunk 領域抽出をどのようにして行うかに基づいて，FPID 生成に使用する chunk 領域サンプルの抽出を FPGA 上の HiFP2.1 回路で行う”FPGA 上でサンプル抽出を行う FPGA 実装”とフトウェアドライバ側が FPGA 側へデータを送信する前に chunk サンプルの抽出を行う”ソフトウェアドライバ上でサンプル抽出を行う FPGA 実装”2 つの実装形式がありえた．そこで，この章ではそれら 2 つの実装形式のうち HiFP2.1 の実装に最適である実装として”FPGA 上でサンプル抽出を行う FPGA 実装”を提案した．これは，chunk 領域数増加に伴う通信データ量増加の対数関数性などを根拠として，chunk 領域数の増加する場合において”FPGA 上でサンプル抽出を行う FPGA 実装”におけるホスト PC-FPGA 間の通信時間のボトルネックよりも”ソフトウェアドライバ上でサンプル抽出を行う FPGA 実装”のメモリアクセス量が増加がより重くなるという考察に基づく．

次に，HiFP2.1 の実装のためには chunk 領域の個数も決定しなければならないことについて言及し，最適な chunk 領域数の決定方法についても提案を行った．提案としては，chunk 領域数 1,024 が最適であるとした．これは，chunk 領域数増加に伴う通信データ量増加の対数関数性などを根拠として，chunk 領域数が増加するごとに処理時間増加が緩やかになることと，HiFP2.1 の特性上，可能な限り chunk 領域数を増加させることが検索精度増加につながること，および離れた特徴量同士から生成された FPID ビットがエラーを起こしやすく，その割合が高くなりすぎると検索精度に影響することに基づく．

さらに，提案された chunk 領域サンプルの抽出方法に基づく HiFP2.1 を実装する際のホスト PC および FPGA の構成と，ホスト PC-FPGA 間の通信方法である PCIexpress と Xillybus　IP Core について，ホスト PC から FPGA へ入力されるサンプルデータの FPGA 上での取り扱い方法，FPID の生成方法について述べた．最後に，本研究での提案における HiFP2.1 のアルゴリズム上の各パラメータ設定について述べた，

次章では，検索精度および実行速度の観点からこの実装システムを他の実装パターンや HiFP2.0 の実装などと比較し，その検証や考察などを行う．

第4章　提案手法の評価と考察

4.1　はじめに

前章では，本研究における HiFP2.1 アルゴリズムの実装方法についての提案を述べた．この章では，前章で提案したバージョンの手法とその他のバージョンの手法を比較し，その性能について検証する．そして，HiFP2.1 では楽曲データ全体を使用するため，実時間における処理時間は増加すると考えれるため，提案手法と HiFP2.0 アルゴリズムと提案手法を比較し，評価と考察を行う．

4.2　評価の目的と実験内容

本研究においては，HiFP2.1 アルゴリズムおよびそのハードウェア実装について提案を行っている．

HiFP2.1 の実装形式および各実装形式での各チャンク数を網羅する形でリスト化すると，以下の通りになる．

1. ソフトウェア（C++）のみで実装した HiFP2.1

   chunk 領域数 2．4，8，16，32，64，128，256，512，1,024，2,048

2. FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装

   chunk 領域数 2，4，8，16，32，64，128，256，512，1,024，2,048

3. ソフトウェアドライバ上でサンプル抽出を行う HiFP2.1 の FPGA 実装

   chunk 領域数 2．4，8，16，32，64，128，256，512，1,024，2,048

本研究における提案手法は，この中における「FPGA 上でサンプル抽出を行う FPGA 実装 HiFP2.1」の「chunk 領域数 1,024」である．

また，比較に使用する HiFP2.0 は,「FPGA 上でサンプル抽出を行う HiFP2.0 の FPGA 実装」を用いる．

この項では，実験より得られる 3 つの評価軸を元に評価を行う．評価軸は以下の 3 点である．

- 実時間における FPID 生成処理の実行時間
- FPID による検索精度
- リソース使用量および電力消費量

4.3 実験条件

4.3.1 実験環境

本研究では，ホスト PC に PCIexpress インタフェースで接続された FPGA に各種実装をインプリメンテーションし，ホスト PC 側のソフトウェアドライバによる制御によって FPID 生成処理を行う．

その実験環境は表 4.1 および表 4.2 のようになっている．

項目	説明
CPU	Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz
メモリ	16GB
ストレージ	ATA WDC WD10EZEX-00B 1000GB
OS	Ubuntu 18.04.1 LST
コンパイラ (C++)	gcc 7.3.0
開発ツール (FPGA)	Vivado 2019.1

表 4.1: ホスト PC の性能

項目	説明
製品名	Xilinx Virtex ®-7 FPGA VC709 コネクティビティ キット
FPGA	Vertex-7 XC7VX690T-2FFG1761C
ロジックセル	693,120
DSP スライス	3,600
メモリ (Kb)	52,920

表 4.2: FPGA ボードの特徴

また，本研究で使用する全ての楽曲データは実装した簡易自動楽曲ファイルジェネレータから生成したものを使用している．これについては次節で説明する．

4.3.2 簡易自動楽曲ファイルジェネレータについて

この実験において使用する 10,000 曲の楽曲データセットの準備のため，簡易的な自動楽曲ファイルジェネレータソフトウェアを Python を用いて実装した．このジェネレータは，Google 社傘下の DeepMind 社が開発，発表したディープニューラルネットワークアーキテクチャである WaveNet [31] を参考に，AnalyticsVidhya

で公開されているアルゴリズム [32] を元にニューラルネットワークのアルゴリズムを改良して構築した．

WaveNet は音声合成を行うためのディープニューラルネットワークアーキテクチャの一つである．WaveNet はパラメトリック TTS モデル（TSS:Text to Speech）を採用している．これは，発声データ群と文法規則を元にテキストデータから音声信号を合成するものである．モデルとして，発声データを入力信号，テキストデータを教師信号として音声信号の学習を行う．また，これは音声合成以外にも音楽合成に対しても適応できる．であるため，このジェネレータでは学習するデータセットとして Classical Piano Midi Page [33] で配布されている著作権切れの楽曲の midi オーディオファイル 58 曲を教師データとし，midi オーディオファイルのノートとコードを学習の対象とし，新しい midi オーディオファイルを生成する．midi ファイルの取り扱いには MIT が公開している music21 ライブラリを使用した．

WaveNet は Dilated 1D Causal Convolution を採用している．これによって音声データのサンプル（ジェネレータではノートとコードのセット）を時系列を伴った一次元配列とみなし，確率分布を学習する．Dilated 1D Causal Convolution では，Causal Convolution と Dilated Convolution という 2 つの畳み込みニューラルネットワークの技術が用いられている．

Causal Convolution は，時系列データにおいて，ある時刻 t における推論処理では，時刻 t-1 より前の要素 x_1,$\cdots$,x_{t-1} のみを畳み込み処理の要素とするものである．そのため，自己回帰構造を持っておらず，高速に時系列情報に基づいた学習が可能である．

Dilated Convolution は，層が深くなるごとにカーネルの要素間により大きな zero padding を挿入する．これによって，層が深くなるごとに入力データのより広い範囲から畳み込む要素を得ることが出来る．これによって，受容野を拡大し，より広い範囲の時系列情報を学習できるようになっている．

このジェネレータでは，midi ファイル内の各ノートとそのコードを 1 対 1 で組み合わせる．それの組み合わせにそれぞれ固有の ID（整数）を割り振り，連想配列とする．これによって元の Midi ファイルの時系列データを数値による時系列データに変換し，その時系列情報を含む確率分布を学習させた．

また，このジェネレータの開発環境は表 4.3 の通りであり，学習したモデルのアーキテクチャは表 4.4 の通りであり，このモデルの表記は tensorflow によって出力されたものを用いている [1].

また，入力データとして，Classical Piano Midi Page からランダムに指定した”Beethoven, Ludwig van”,”Haydn, Joseph”,”Schumann, Robert” の 3 ページで配布されている全ての midi ファイル，合計 58 曲を用いた．Epoch 数は 50, 学習時の最終的な loss 値は 2.7350，検証時の最終的な val_loss 値は 3.0069 となった．出力としては，30 秒前後の 10,000 曲の midi ファイルを生成した．

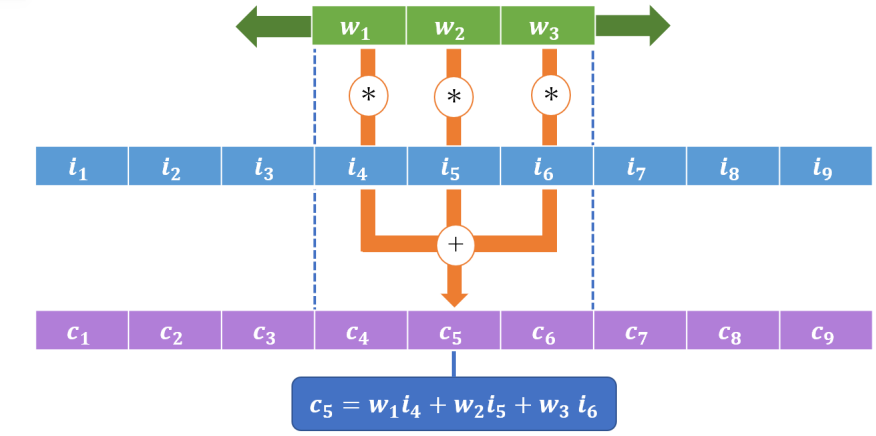


図 4.1: 1D Convolution の図解

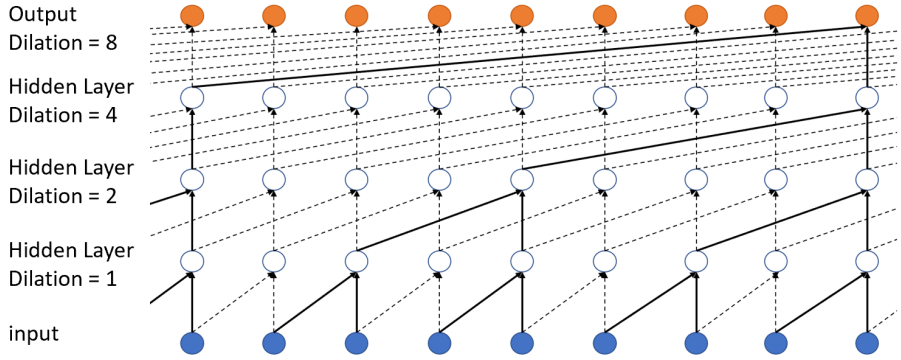


図 4.2: Dilated 1D Causal Convolution の図解

項目	説明
CPU	Intel(R)Core(TM)i7-8650U CPU @ 1.90GHz
メモリ	16.0GB
GPU	GeForce GTX 1060
ストレージ	SAMSUNG MZFLW256HEHP-000MV
OS	Windows10 PRO 1903
開発言語	Python (Anaconda 1.9.12 for Python 3.7 version)
機械学習ライブラリ	TensorFlow 1.14, Keras2.3.1
Midi 編集ソフトウェア	MuseScore 3.4.2.9788
Midi 変換ソフトウェア	MidiRenderer 3.8.0.0

表 4.3: 簡易自動楽曲ファイルジェネレータ実装における開発環境

Layer(type)	Output Shape	Param
embedding_1 (Embedding)	(None, 32, 100)	10300
conv1d_1 (Conv1D)	(None, 32, 64)	19264
dropout_1 (Dropout)	(None, 32, 64)	0
max_pooling1d_1(MaxPooling1)	(None, 16, 64)	0
conv1d_2(Conv1D)	(None, 16, 128)	24704
dropout_2 (Dropout)	(None, 16, 128)	0
max_pooling1d_2(MaxPooling1)	(None, 8,128)	0
conv1d_3(Conv1D)	(None, 8, 256)	98560
dropout_3 (Dropout)	(None, 8, 256)	0
max_pooling1d_3(MaxPooling1)	(None, 4, 256)	0
conv1d_4 (Conv1D)	(None, 4, 512)	393728
dropout_4 (Dropout)	(None, 4, 512)	0
max_pooling1d_4 (MaxPooling1)	(None, 2, 512)	0
conv1d_5 (Conv1D)	(None, 2, 1,024)	1573888
dropout_5 (Dropout)	(None, 2, 1,024)	0
max_pooling1d_5 (MaxPooling1)	(None, 1, 1,024)	0
global_max_pooling1d_1 (GlobalMaxPooling1)	(None, 1,024)	0
dense_1 (Dense)	(None, 1,024)	1049600
dense_2 (Dense)	(None, 103)	105575

表 4.4: 簡易自動楽曲ファイルジェネレータのアーキテクチャ[1]

4.4 chunk 領域抽出の実装形式の決定法の評価

4.4.1 決定法における評価の概要

第 3 章で示した通り，chunk 領域抽出をどのタイミングで行うかによって 2 パターンの実装が考えられる．第 3 章で示した提案手法と，その他の実装方法について実験を行い，提案手法で述べた chunk 領域抽出の実装形式の決定法について評価する．

4.4.2 決定法における評価軸

chunk 領域抽出の実装形式の決定法を評価するにあたっては，実装形式ごとに生成される FPID 自体に差異はないので，実行処理速度を基準にする．

よって，この決定では，前述した評価軸のうち,「実時間における FPID 生成処理の実行時間」を用いる．

以下の条件について FPID 生成の時間がどれ程かかっているかの計測を行い，3 つの実装形式においてその実時間における FPID 生成処理の実行時間を比較した．

- ソフトウェア（C++）のみの HiFP2.1 の実装
- ソフトウェアドライバ上でサンプル抽出を行う HiFP2.1 の FPGA 実装
- FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装

4.4.3 実行時間の検証方法

実行時間の計測方法としては，C++の標準ライブラリ内のシステムコール関数である clock_gettime() をソフトウェア実装およびドライバ上で使用して 1ns の精度でウォールクロックタイムを計測する．

この実験で使用する楽曲データは 30 秒 (5,292,588byte) の PCM の wav データである．これは，ヘッダの 44byte を除くと 5,292,544byte のデータ部を持つ．この wav データから FPID 生成時間を計測する．

また，処理時間を計測する処理工程の順序はバージョンごとに以下の通りになる．また，それぞれバージョンにおいて全ての chunk 数を個別で計測する．

- ソフトウェア（C++）のみの HiFP2.1 の実装
    1. 合計処理時間（whole time）
- ソフトウェアドライバ上でサンプル抽出を行う HiFP2.1 の FPGA 実装
    1. ドライバ上での楽曲データの展開時間（file time）
    2. ドライバ上での chunk 領域の抽出時間（extract time）
    3. hostPC から FPGA への書き込み時間（write time）

4. FPGA から hostPC への読み込み時間（read time）

- FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装
  1. ドライバ上での楽曲データの展開時間（file time）
  2. hostPC から FPGA への書き込み時間（write time）
  3. FPGA から hostPC への読み込み時間（read time）

この実験結果はグラフ 4.3 の通りとなった.

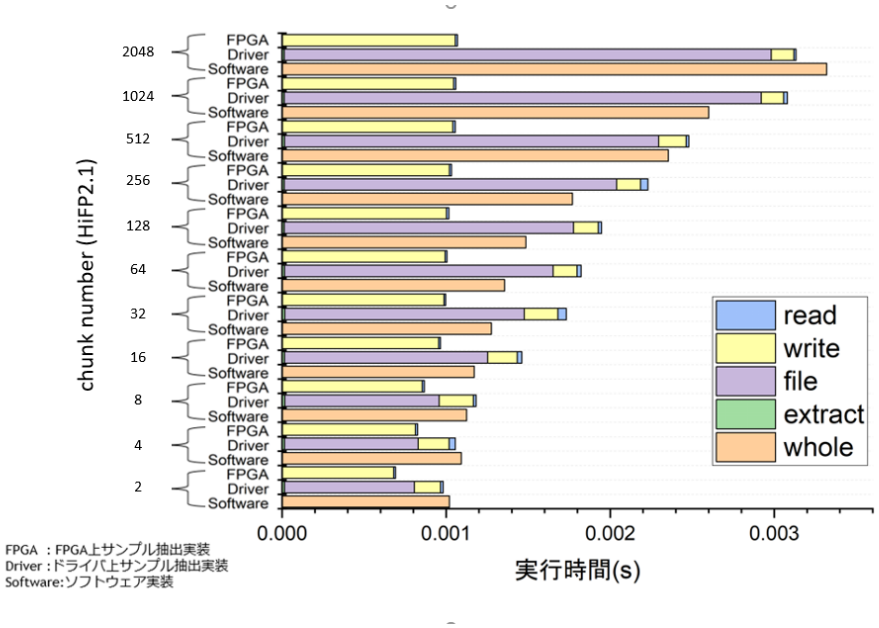


図 4.3: 各種実装形式での実時間における実行時間のグラフ

実験結果としては,「ソフトウェア（C++）のみの HiFP2.1 の実装」および「ソフトウェアドライバ上でサンプル抽出を行う HiFP2.1 の FPGA 実装」よりも「FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装」の実行処理時間が全体として速いことが分かった.

4.4.4 実験結果と提案手法の評価

ソフトウェア（C++）のみの HiFP2.1 の実装については，指数関数的に処理実行時間が増加している．これは，第 3 章で示した通り，ソフトウェアで chunk 領域サンプル抽出を行う場合，chunk 領域数が増加するごとに chunk 領域の分布がより散発的となる．よって，楽曲データからの chunk 領域を抽出する処理として

楽曲データが展開されたメモリへのアクセス回数が増加するためであると考えられる．

ソフトウェアドライバ上でサンプル抽出を行う HiFP2.1 の FPGA 実装においては，ソフトウェアのみでの実装と同様に，chunk 領域数の増加に伴うドライバ上でのメモリアクセスの回数増加およびアクセス位置の散発化によって処理実行時間，特に chunk 領域抽出時間（extract time）が指数関数的増加を見せている．一方で，chunk 領域抽出後の FPGA 側への書き込み処理時間（write time），FPGA 内部での AFP 生成処理時間（wait time）および FPGA からの読み出し処理時間（read time）では全てのバージョンで一定量のサンプルと一定量の FPID を取り扱うので，差異がほとんど見られなかった．

FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装では，chunk 領域数の増加に伴う送信するサンプルデータ量の増加によってホスト PC から FPGA への書き込み時間（write time）の増加が顕著である．しかし，この実装ではホスト PC から FPGA に楽曲データを送信するにあたって，メモリ上に展開された楽曲データをシーケンシャルアクセスでそのまま Xillybus の Write File に書き込むため，他の実装と比較して write time の増加は相対的に小さく収まっている．

これらの結果から,「ソフトウェア実装」および「ソフトウェアドライバ上でサンプル抽出を行う HiFP2.1 の FPGA 実装」よりも,「FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装」の方が実験環境においては実行処理時間が抑えられることが分かる．

また，chunk 領域数が 16 以上になると chunk 領域数の増加に対する処理時間全体の増加率が緩やかになり，差に最大でも 0.1ms 程度の違いしか現れない．

よって，実時間における AFP 生成処理の実行時間の観点から見て提案手法である「FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装」が最も HiFP2.1 に適した実装であると言える．

4.5 最適な chunk 領域数の決定法の評価

4.5.1 決定法における評価の概要

第 3 章で示した通り，実装する HiFP2.1 においては，AFP 生成に使用するサンプルを抽出する chunk 領域数を任意に設定できる．第 3 章で示した提案手法と，その他の chunk 領域数の実装について実験を行い，提案手法で述べた最適な chunk 領域数の決定法について評価する．

4.5.2 決定法における評価軸

最適な chunk 領域数の決定法を評価するにあたっては,「FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装」においては chunk 領域数は実行処理時間および

検索精度に影響を与える．

よって，この決定では，前述した評価軸のうち,「実時間における FPID 生成処理の実行時間」「FPID による検索精度」を用いる．

前者については既に行った実験の結果を用いる．また，後者については実装した簡易自動楽曲ファイルジェネレータから生成した 10,000 曲の楽曲データベースを用い，BER を基準とした AFP による楽曲の検索精度測定の実験を行った．

これら 2 つの実験結果から，提案手法における最適な chunk 領域数の決定法の評価を行う．

- ソフトウェア（C++）のみの HiFP2.1 の実装
- ソフトウェアドライバ上でサンプル抽出を行う HiFP2.1 の FPGA 実装
- FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装

4.5.3 HiFP2.1 アルゴリズムの検索精度の検証方法

今回は AFP による楽曲の識別の方法として，BER を用いる．BER はあるオリジナルの楽曲の FPID とそのオリジナル楽曲を元に各種圧縮フォーマットへと変換済みの楽曲のそれぞれの FPID の間のハミング距離を計算し，全 bit に対するハミング距離の値の割合を算出したものを指す．BER は同一楽曲から生成された AFP 同士の場合は 0 に近く，異なる楽曲同士から生成した FPID 同士の場合には 0.5 に近くなる性質を持つ．ここで，図 4.6 は，用意した楽曲データベースにおける圧縮フォーマット MP3 の ABR8kbps における BER の正規分布であるこの図は，データベース内の 10,000 曲のオリジナル楽曲と MP3 ABR8kbps に変換した後に PCM に再変換した楽曲 10,000 曲それぞれから生成した AFP を，前者 10,000 個と後者 10,000 個の全ての組み合わせ全 100,000,000 通りについて BER を算出し，その分布を正規分布として示したものである．示されている正規分布は，同一楽曲から生成された FPID 同士の BER 合計 10,000 個の分布と，異なる楽曲同士から生成した FPID 同士の BER 合計 99,990,000 個の分布の 2 種類である．前者が 25%付近に偏って分布しているのに対して，後者は 50%付近に分布しており，前述した BER の性質を示していることが分かる．

BER を用いた楽曲の識別方法としては，インターネットトラフィック上に流れる保存形式の不明な未知の楽曲を識別するため，この同一楽曲同士の BER と異なる楽曲同士の BER の分布の違いを利用し，2 つの分布の交差点に BER の閾値を設定する．ある未知の楽曲クエリが渡されたときに，データベース内の FPID との BER を求め，閾値以下の組み合わせのみを検索対象の候補として示す．

であるので，今回の検証では，ある一定の閾値を設定し，BER を用いた識別方法の信頼性について検証する．

具体的には，図 4.4 と図 4.5 で示されているように行う．

まず，10,000 曲のオリジナルの PCM 楽曲と，各種圧縮フォーマットに一度変換した後に PCM に再変換した楽曲を生成する．

そして，HiFP2.0 および HiFP2.1 の各チャンク数の 11 バージョンそれぞれにおいて「オリジナル楽曲 10,000 曲」と「オリジナル楽曲を元に各種圧縮フォーマットへと変換済みの楽曲（以下，変換済み楽曲）10,000 曲」のそれぞれの楽曲群の FPID を，前者 10,000 個と後者 10,000 個の全ての組み合わせ全 100,000,000 通りについて BER を算出する．

識別の信頼性については，図 4.5 で示されているように，識別を行った場合の偽陰性，偽陽性の識別失敗数を元に判断する．つまり，ある閾値 α を設定した場合を考える．この時，ある楽曲同士から生成された AFP の BER の値が α を越えたにも関わらず，その組み合わせが同一楽曲同士であった場合は偽陰性の識別失敗とし，逆に α を下回ったにも関わらず同一楽曲同士ではなかった場合にはそれを偽陽性の識別失敗とする．前者が存在する場合は正しい楽曲が検索後方に含まれない状況が発生する可能性があり，後者が存在する場合は検索候補が絞り込めないということになる．

特に HiFP2.1 においては，楽曲全体からサンプルを周期的に取り出す箇所である chunk 数の各バージョンそれぞれの場合での一致率および BER を算出し，各 chunk 領域のサイズと識別信頼性の関連性について検証し，最適な chunk 領域数の決定法の評価に用いる．

本実験で使用する変換形式は 4.5 に示す通りである．MP3 以外の各変換形式においては，オリジナルの楽曲データの形式であるコーデック Wave Format PCM のサンプリング周波数 44,100Hz，サンプルビット 16bit，Stereo 音源と比較して音質的に大きく劣化しない設定を選択した．また，MP3 では変換後の音質の異なった複数のビットレートおよびそれに伴うサンプリング周波数に変換したファイルを用意し，その音質と識別成功率および BER との関係性を調査する．

変換形式	品質
MP3	ABR256kbps (sampling rate:44,100Hz)
	ABR64kbps (sampling rate:22,050Hz)
	ABR8kbps (sampling rate:11,025Hz)
OGG	Quality Level 5
WMA	128kbps CBR
FLAC	Compression Level 5

表 4.5: 実験に使用する変換形式

また，各種保存形式への変換には AV Audio Converter 2.0.5 を使用した．

今回，閾値としては音質が大幅に劣化すると考えられる変換形式 MP3 ABR8kbps の正規分布を用いて設定する．その値は 41.1587 である．

4.5.4 検索精度の検証用ソフトウェアの実装

検索精度検証実験では，専用の BER 大規模処理用ソフトウェアを実装し，使用した．その構成図が図 4.7 である．

このソフトウェアに実装されている HiFP アルゴリズムは”ソフトウェア（C++）のみの HiFP2.1 の実装”，”ソフトウェア（C++）のみの HiFP2.0 の実装”のものと完全に同一である．一方で，このソフトウェアと他の実装との差異は，図 4.7 のようになっている．つまり，FPID 間のハミング距離計算およびそれによる BER 計算処理とそれらの実行結果を txt ファイルとして出力する仕組みが追加され，単体で多数の楽曲に対して処理を行うことが出来るように改良したものである．

通常の HiFP のソフトウェア実装は，入力として単一の楽曲データファイルを，出力としてその楽曲データに対応する FPID を想定する．ソフトウェアがコマンドラインにおいて実行されるとコマンドライン引数から楽曲データのファイル名を取得する．(”コマンドライン引数受け取り”）そして，その引数を元に HiFP 関数である getfp() が呼び出される．getfp() は指定された楽曲データから FPID を生成して txt ファイルとして出力する．FPID は HiFP2.0 アルゴリズムによるものと，HiFP2.1 アルゴリズムの chunk 領域数 2 から 2,048 までの 11 パターンの合計 12 パターン全てについてそれぞれ別のファイルに生成される．(”HiFP アルゴリズム（2.0 or 2.1）”および”FPID”）

一方で，BER 大規模処理用ソフトウェアは，入力として”song[曲番号]．wav”と名付けられた”song00000.wav”から始まって最大”song99999.wav”までの連番の楽曲データファイル群であり，これらについて”Original wav”ファイル（図 4.4 における”オリジナル wav ファイル”）としてのものと，それをもとに生成された”Converted wav”ファイル（図 4.4 における変換済み wav ファイル）としてのものでそれぞれ同一曲数分用意したものを想定し，出力として入力から得られた全ての組み合わせの BER の txt データを想定する．ソフトウェアがコマンドラインで実行されるとコマンドライン引数から処理する曲数と”Converted wav”ファイルの変換形式を取得する．(”コマンドライン引数受け取り”）そして，指定され楽曲データ全てについて，それぞれで getfp() が呼び出され FPID を生成，txt ファイルに出力する．ここで生成される FPID は通常の HiFP のソフトウェア実装と同じ 12 パータンで別々の txt ファイルに格納される．その上で，”Original wav”ファイルのものと”Converted wav”ファイルのもので合計 24 個の txt ファイルが生成される．そして，出力された FPID データの txt ファイル群を用いて 12 パターンそれぞれにおいて Original wav”ファイルのものと”Converted wav”ファイルのものとの全ての組み合わせのハミング距離計算を行う．(”ハミング距離計算”）そして，その結果をもとに，BER を算出し，txt ファイルとして出力する．(”BER 算出”）

BER 大規模処理用ソフトウェアのこのような仕様上，使用する楽曲のファイル名は必ず”song00000.wav”から始まって最大”song99999.wav”までの連番で用意する必要がある．その上で，”Original wav”ファイルと Converted wav”ファイルで

同一の曲数の楽曲データを用意する必要がある．

ここで，getfp() の引数である ifp は処理する楽曲ファイルの FILE 型構造体のファイルポインタ，fname は出力ファイルに使用する楽曲ファイル名の string 型変数，number_chunk は chunk 領域数の int 型変数である．

4.5.5 各変換形式における検索精度の検証の評価

4.5.5.1 様々な変換形式における実験

HiFP2.0 および各 chunk 領域数の HiFP2.1 における様々な変換形式での識別性能の違いについて調査するため，楽曲データを複数の変換形式に変換し，そこから生成される FPID による検索精度を調査する実験を行った．この実験によって，変換形式の違いに対する HiFP2.1 の検索精度のロバスト性および chunk 領域数増加による改善度に違いが存在するかを検証する．

実験結果をグラフ 4.8 に示す．実験結果としては，偽陽性の識別失敗は chunk 領域数が低いものほど多く起こっていることが分かった．また，偽陽性の識別失敗については，発生しなかった．

4.5.5.2 様々な変換形式における実験の評価と考察

HiFP2.1 では，chunk 領域数が大きくなるごとに楽曲全体から取り出す chunk の位置の分散が大きくなる．そのため，異なる楽曲同士において楽曲のある局所的に似通ってしまっている場合でも，chunk 領域数が大きな方が楽曲全体の特徴から識別を行うことが出来るために，似通った部分が識別の判断材料となる確率が減少し，偽陽性の識別失敗数を低く抑えることができるものと思われる．

また，グラフ 4.8 からわかる通り，複数の変換形式においても，その偽陽性失敗数が著しく悪化するようなことはなかった．その上で，chunk 領域数を増加させることで基本的には検索精度が向上している．

であるので，HiFP2.1 は同じ程度の音質であれば，変換形式の違いによって検索精度に違いが起きるようなことはないと考えられる．

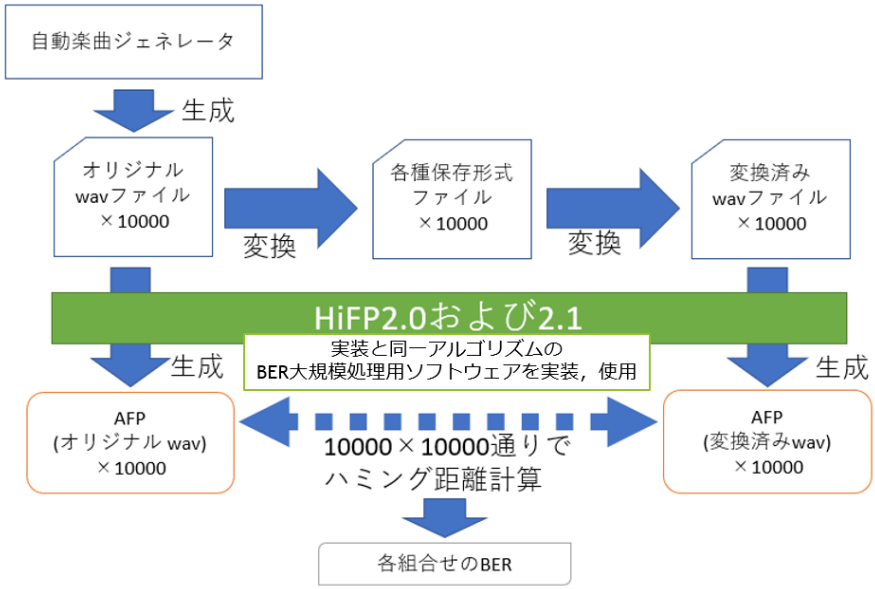
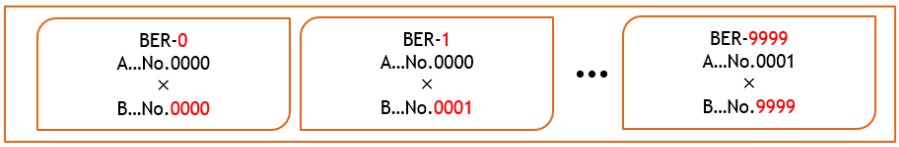


図 4.4: 実験２についての図解-BER 算出について

例：No.0000の識別
A:オリジナル楽曲/B:変換済み楽曲



	BER <= α （同一楽曲候補）	BER > α （同一楽曲候補でない）
A=B	正しく判定	偽陰性
A≠B	偽陽性	正しく判定

全ての組み合わせで，偽陰性，偽陽性の個数を調べる

図 4.5: 実験２についての図解-識別成功率算出について

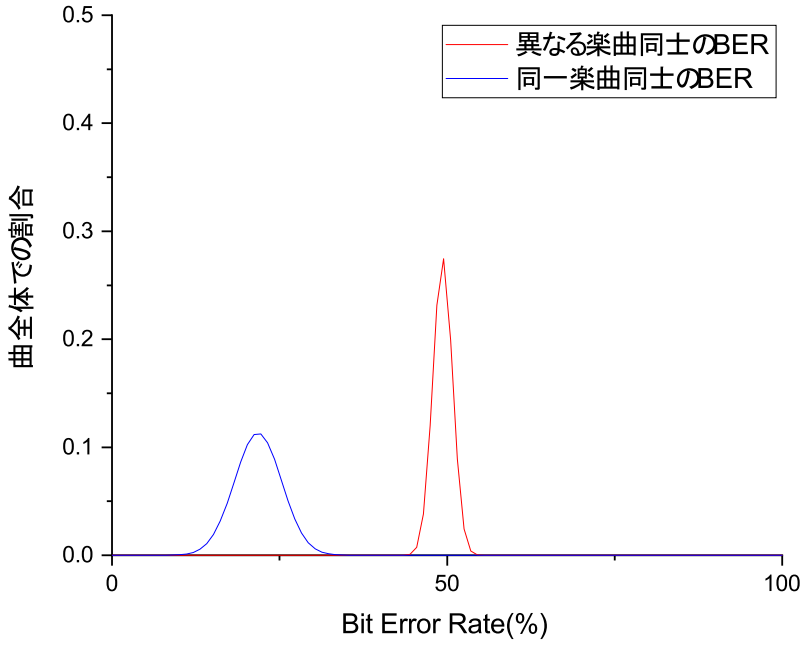


図 4.6: MP3 ABR8kbps における BER の正規分布

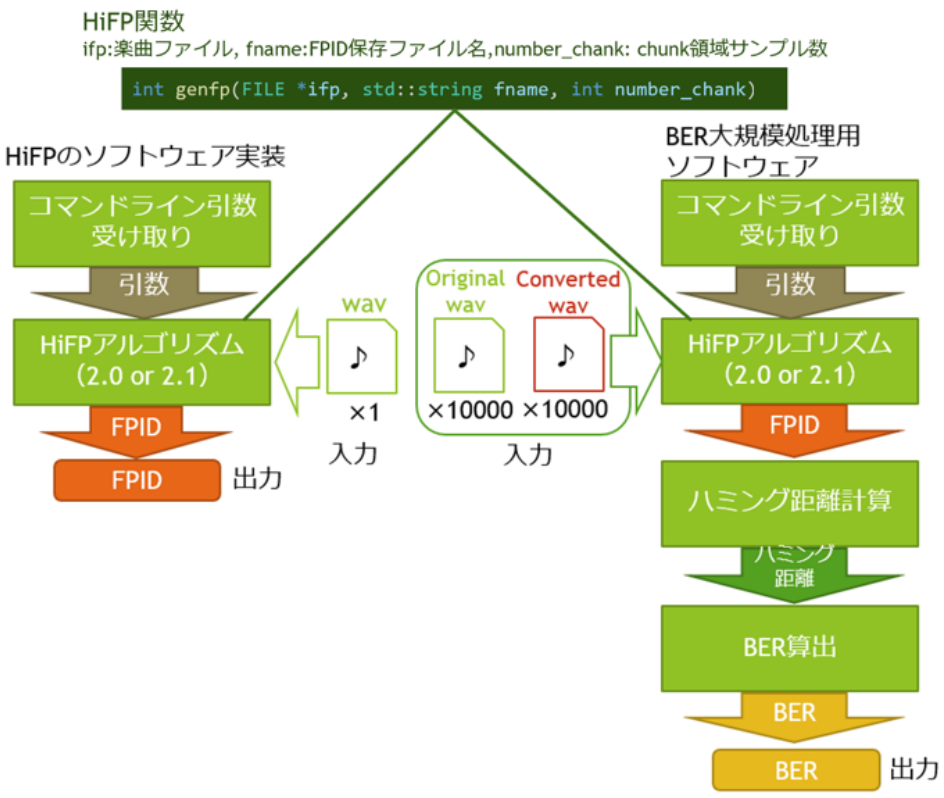


図 4.7: HiFP のソフトウェア実装と BER 大規模処理用ソフトウェアの違い

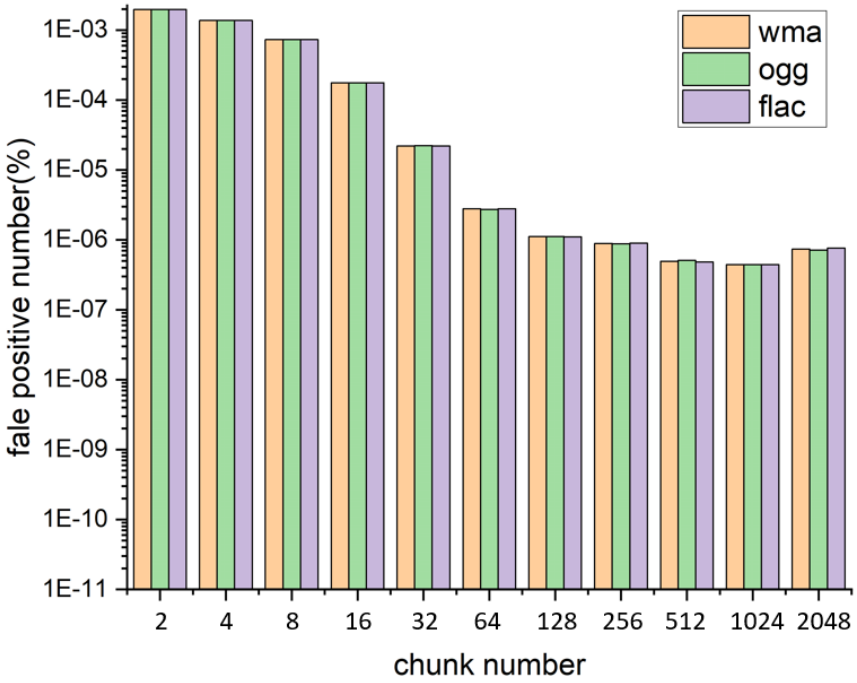


図 4.8: 各変換形式における偽陽性の識別失敗数のグラフ

4.5.5.3　複数のビットレートの MP3 それぞれにおける実験

HiFP2.0 および各チャンク数の HiFP2.1 において，変換の際の異なる圧縮率による劣化度と識別性能との関係性について調査するため，変換形式 MP3 において複数のビットレートへの変換し，そこから生成される AFP による検索精度を調査する実験を行った．ここでは，変換する MP3 のビットレートとそのサンプリングレートが低いものほど音質が劣化するものとする．これによって，HiFP2.1 の音質の劣化に対してのロバスト性および chunk 数増加による検索精度の違いが表れるかを検証する．実験結果を表 4.9 とグラフ 4.10 に示す．

この実験においては，グラフ 4.9 およびグラフ 4.10 から分かる通り MP3 の 256kbps および 64kbps に変換した場合においては HiFP2.0 およびどの chunk 領域数の HiFP2.1 においても偽陰性の失敗は発生しなかった．その一方で音質を十分に劣化させた 8kbps においては，HiFP2.0 およびより少ない chunk 領域数の HiFP2.1 ほど多くの楽曲で偽陰性の識別失敗が発生した．

4.5.5.4　複数のビットレートの MP3 それぞれにおける実験の評価と考察

グラフ 4.9 およびグラフ 4.10 から分かるように chunk 領域数の増加に従って偽陰性および偽陽性の識別失敗数が減少している．また，より音質の劣化した mp3-8 の方が偽陽性の検索失敗率がわずかに低くなっているが，これは閾値の設定にこの音質の mp3 から得たデータを使用したためである．それでも，偽陰性の検索失敗は mp3-8 のみで起こっている．つまり，HiFP2.1 においても音質の劣化した楽曲データをクエリなどにした検索では，そうでないものの場合よりも検索精度は悪化するということである．

また，全体として chunk 領域数が増加するに伴って検索失敗数も大幅に減少している．これは，大きく音質が劣化した楽曲データであっても chunk 領域を細分化し分散配置することで音質劣化部分を確率的に回避する手法は有効であることを示している．

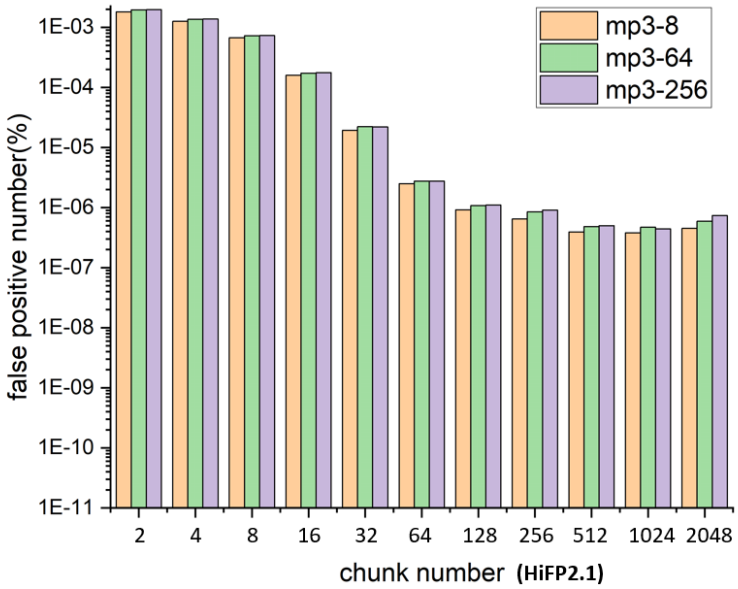


図 4.9: MP3 における偽陽性の識別失敗数のグラフ

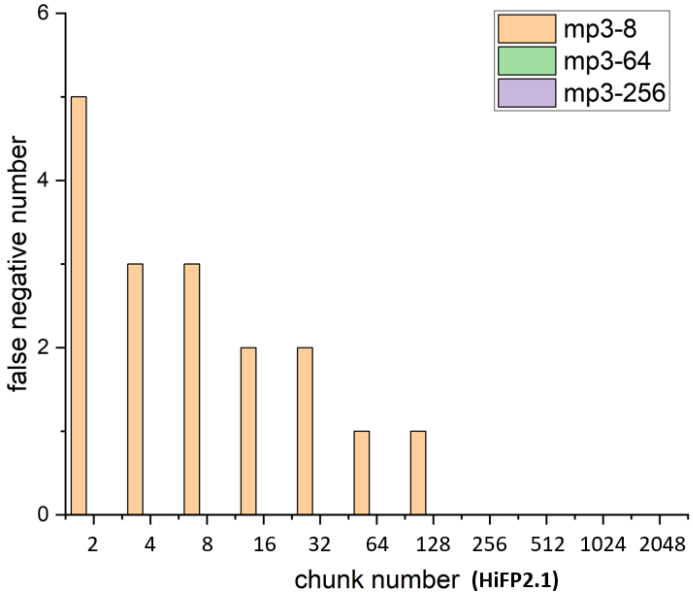


図 4.10: MP3 における偽陰性の識別失敗数のグラフ

4.5.5.5　AFP による検索精度の実験全体での提案手法の評価

この実験において特に偽陰性の検索失敗数全ての結果をまとめたグラフは図 4.11 のようになる．

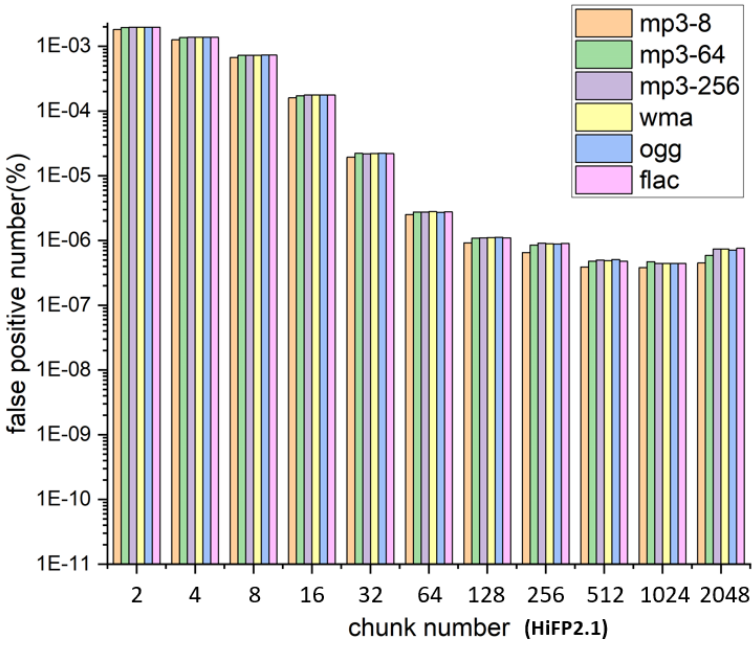


図 4.11: 実験全体における偽陰性の識別失敗数のグラフ

擬陽性の検索失敗数については，mp3 の 8kbps のみで発生することは既に 4.9 で示した．

この実験全体において，楽曲データを変換する場合，どういった形式においても十分な音質が保証された設定を行えばそれらの楽曲に対して HiFP2.0 および各チャンク数の HiFP2.1 は十分な識別性能を示すことが分かった．

その一方で，ビットレートやサンプリング周波数などを変化させて音質を十分に劣化させた場合，それらの楽曲に対して HiFP2.0 および chunk 領域数の少ない HiFP2.1 においては，その中の一定数を識別することが出来なかった．一方，chunk 領域数を十分にとった HiFP2.1 は前者の識別できなかった楽曲をも識別可能となることが分かった．

また chunk 領域数 2,048 の場合は，chunk 領域数が 2,048 より少ない 1,024 の場合に比べて偽陽性の失敗率が増加している．

4.5.5.6　2 つの実験の結果と提案手法の評価

「実時間における FPID 生成処理の実行時間」「AFP による検索精度」の 2 つの評価軸で提案手法を評価する．提案手法で用いられている FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装における実時間における FPID 生成処理の実行時間はすでに行った実験よりグラフ 4.12 のようになっている．

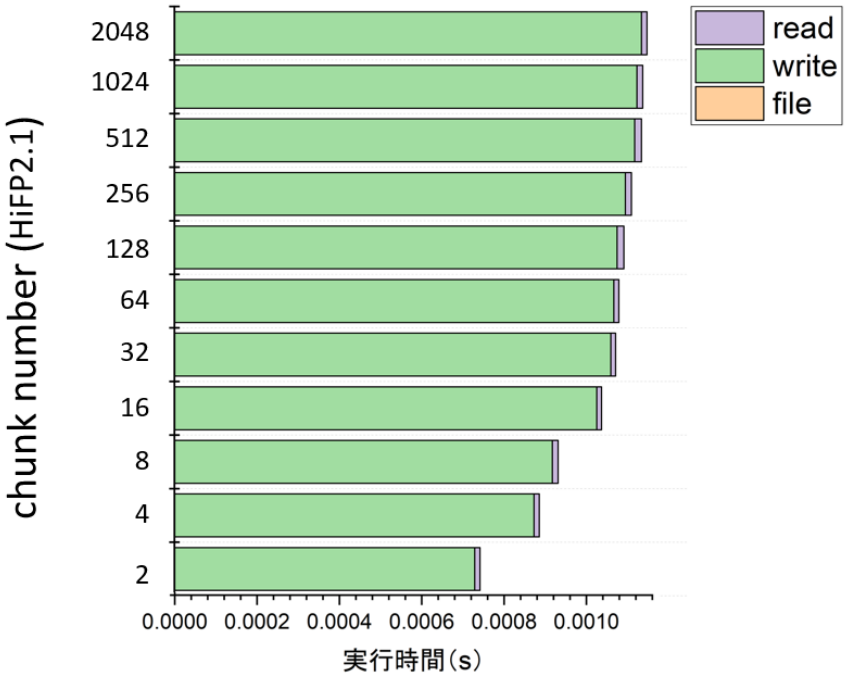


図 4.12: ソフトウェアのみで実装した HiFP2.0 および HiFP2.1 の計測結果

実行時間の観点から見ると，実行時間の増加量が chunk 領域数の増加に対して対数関数的であることが分かった．このことから，一定以上 chunk 領域数を大きくとる場合は実行時間の増加については無視できる．そして，検索精度の観点から見ると，大きく劣化した楽曲データ群においては chunk 領域数を大きくとることで指数関数的に検索精度が向上することが分かった．

一般的に楽曲検索システムを使用する場合には十分な検索精度が保証されている必要がある．また，提案手法において，実行時間の増加率に対して検索精度の向上率が顕著であるといえる結果が得られた．であるため，本研究において最適な chunk 領域数を決定する場合には検索精度を優先する．

また，隣接した特徴量から生成される FPID ビットの割合が全体の 50%しかない chunk 領域数 2,048 の場合は，より割合の多い 1,024 の場合に比べてわずかに偽陽性の失敗率が増加している．これは，第 3 章で述べた通り，離れた特徴量同士から生成される FPID の割合が高い chunk 領域数 2,048 の実装では対応できない劣化を見せた楽曲データが存在したことが原因であると考えられる．

よって，これらのことから，提案手法である”chunk 領域数=1,024”が最も HiFP2.1 に適した実装であると言える．

4.6 精度および処理時間における HiFP2.0 との比較と評価

この実験では，提案手法と，HiFP2.0 アルゴリズムの FPGA 上でサンプル抽出を行う FPGA 実装を,「実時間における AFP 生成処理の実行時間」「AFP による検索精度」の評価軸で比較し，提案手法について評価する．

4.6.1 HiFP2.0 と比較した実時間における AFP 生成処理の実行時間の評価

ブロードキャストモニタリングなどのインターネット上での楽曲データ検索の利用においては，FPID の生成時間は重要である．HiFP2.1 では楽曲データ全体を扱っている性質上，データの冒頭から固定長で FPID を生成する HiFP2.0 に比べて処理時間がかかってしまう．この研究では，ホスト PC から送信されてくるデータをストリーム処理しているので，HiFP2.1 の処理時間の大幅な改善を考える場合は，通信部分の改善を行わざるを得ないと考えられる．

また，この実験では，提案手法の実行速度が HiFP2.0 のものと比較してどの程度悪化するのかを調査する．また，悪化した場合のボトルネックとなっている処理について考察する．

この実験においても使用する楽曲データは 30 秒 (5,292,588byte) の PCM データである．これは，ヘッダの 44byte を除くと 5,292,544byte のデータ部を持つ．この PCM データから FPID 生成を計測する．

- ソフトウェア（C++）のみで実装した HiFP2.0
- ソフトウェア（C++）のみで実装した HiFP2.1
    1. 合計処理時間（whole time）
- FPGA 上に実装した HiFP2.0
- FPGA 上に FPGA 抽出実装した HiFP2.1
    1. ドライバ上での楽曲データの展開時間（file time）
    2. hostPC から FPGA への書き込み時間（write time）
    3. FPGA からホスト PC への読み込み時間（read time）

計測方法としては，C++のシステムコール関数である clock_gettime() をソフトウェア実装およびドライバ上で使用して 1ns の精度でウォールクロックタイムを計測する．

この実験結果はグラフ 4.13 の通りとなった．

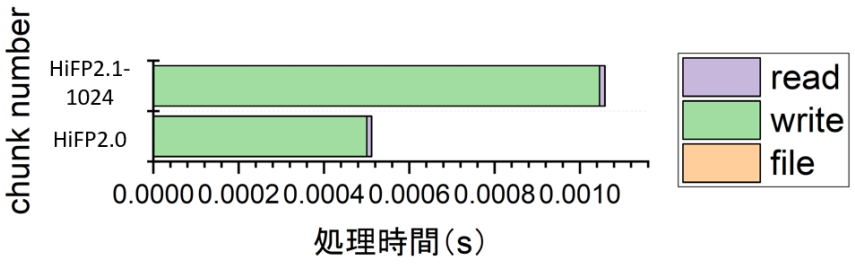


図 4.13: HiFP2.0 アルゴリズムの FPGA 上でサンプル抽出を行う FPGA 実装および提案手法の実行時間

実験結果としては，30 秒の楽曲データに対して提案手法の実装では，同実装の HiFP2.0 の処理時間と比べて処理時間は 2.069 倍となることが分かった．

4.6.1.1 評価と考察

また，提案手法の実装では，同実装の HiFP2.0 の処理時間と比べて処理時間は 2.069 倍となる．この実験では 30 秒の楽曲データを使用しているが，この提案手法では最大 4 分程度の楽曲データを想定している．write time が処理時間全体の大部分を占める本実装においては書き込みサンプル量が実行時間に大きく影響するため，HiFP2.0 実装の処理時間と HiFP2.1 実装の処理時間での比は最大約 16 倍になると考えられる．

HiFP2.0 は FPID の生成時間の高速性を特徴とした FPID 生成アルゴリズムであるため，処理時間の悪化は避けたい事象である．

今後，楽曲データ全体に対するサンプル抽出範囲の増加率と検索精度の関係性を調査し，ある一定の増加率以上では検索精度に高止まりの傾向が見られる場合などに，サンプル抽出範囲を限定する機能を加えることで，FPID 生成時間を抑えることが出来るのではないかと考える．

4.6.2 HiFP2.0 と比較した FPID による検索精度の評価

インターネット上での楽曲転送などの用途においてはファイルのデータ量が重要となってくるため各種圧縮フォーマットが使用されるが，それにより楽曲に劣化が生じることになる．そこで，各種圧縮フォーマットへの変換による音質の劣化が起こっても楽曲を正しく識別できるロバスト性が必要となってくる．よって，この実験では，提案手法において各種圧縮フォーマットにおける楽曲の識別性能について，HiFP2.0 と比較し検証を行う．

今回，閾値としては用いる全ての変換フォーマットにおいて最も音質が劣化すると考えられるものであり，前述した MP3 ABR8kbps の正規分布を用いて設定する．その値は 41.1587 とする．

この実験結果は，グラフ 4.15 およびグラフ 4.15 の通りになった．

実験結果としては，偽陰性の検索失敗は楽曲データの音質が大きく劣化した場合の HiFP2.0 のみで発生し，偽陽性の検索失敗は HiFP2.0 に比べ提案手法の方が大きく改善した．

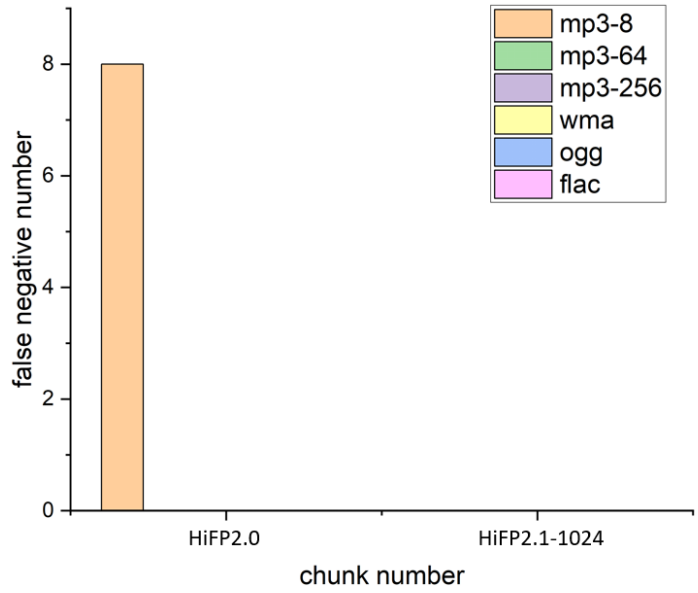


図 4.14: HiFP2.0 アルゴリズムの FPGA 上でサンプル抽出を行う FPGA 実装および提案手法の偽陰性の検索失敗数のグラフ

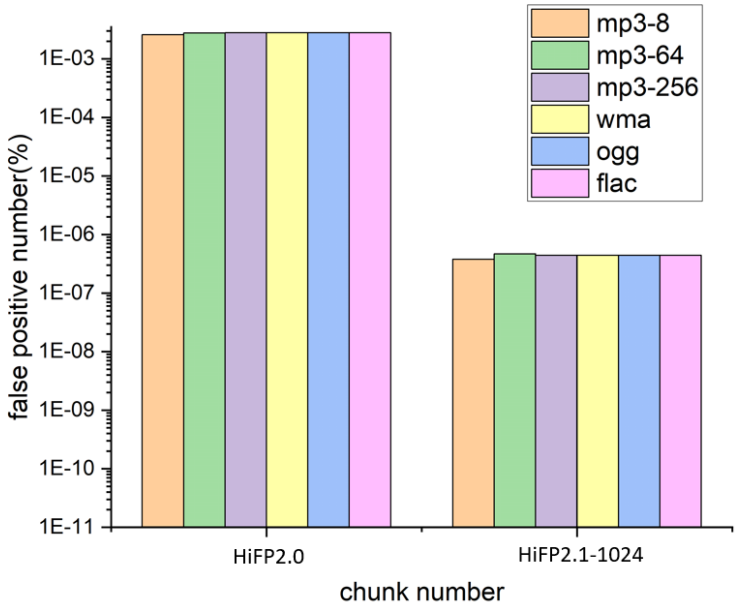


図 4.15: HiFP2.0 アルゴリズムの FPGA 上でサンプル抽出を行う FPGA 実装および提案手法の偽陽性の検索失敗数のグラフ

4.6.2.1　評価と考察

偽陰性の検索失敗では，HiFP2.0 および提案手法において殆どの変換形式において 0 となったが，HiFP2.0 ではこの実験において最も音質が劣化している mp3-8 の場合のみ偽陰性の失敗が発生した．

また，偽陽性の失敗においては，どの変換形式においても HiFP2.0 に比べて提案手法の方が大きく失敗数が減少している．その減少は mp3-8 の場合で最も大きく，HiFP2.0 に比べ提案手法はその 0.000175 倍となっている．

よって，検索精度においては提案手法によって HiFP2.0 から大幅な改善が達成されたと言える．

4.7　リソース使用量および電力消費量

各実装方法におけるリソース使用量を調査するため，Vivado2019 におけるシミュレーションを用いて各アルゴリズムの実装形式のリソース使用量について調査した．

HiFP2.0 に対する HiFP2.1 の回路におけるリソース使用量については，LUT 及び FF が僅かに増加しているのみである．また，HiFP2.1 において全てのチャンク数のバージョンにおいて使用リソース量に違いはなかった．

4.7.1　評価と考察

HiFP2.0 と HiFP2.1 の実装上の差異は図で示されている通り Chunk や Gap の数や距離，現在処理しているデータの波形データ全体の中での位置の計算処理のみであり，これらの要素を実装するために LUT および BRAM のリソースが使用されたと考えられる．一方で，PCIe 関連の回路に差異はないので，Giga bit transceiver (GT) ，Mixed mode clock maneger(MMCM)，IO 等の要素には変化はない．

また，使用電力量においても，両者間に大きな差異はなかった．

これらのことを踏まえると，HiFP2.0 および HiFP2.1 の各チャンク数のバージョン間で使用リソース量および消費電力量上での差異は，アルゴリズムの選択時に考慮する必要はないと考えられる．

4.8　まとめ

この章では，第三章で示された提案手法である HiFP2.1，その chunk 抽出方法に基づく実装形式，そして最適な chunk 数の決定方法をもとに実際に実装した「FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装」かつ「chunk 数 1 ,024」の HiFP2.1 と，その他の実装の実装をもちいて主に検索精度と実行速度をもとにした実験と検証を行った．

その結果，提案手法である「FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装」が chunk 数増加に伴う実行時間の増加率が最も緩やかであり，かつ最も実行時間を抑えられることが分かった．

また，提案手法である「chunk 数 1,024」は，chunk 数の増加に伴ってどの変換形式や音質の楽曲データにおいても検索失敗率が下がることと，chunk 数 2,048 はそれが生成する FPIDbit の中で離れた特徴量同士で比較して生成したものの割合が 50%に達しており，chunk 数 1,024 の場合と比較して若干検索失敗率が悪化するということを踏まえて，最適な chunk 数であると言えることが分かった．

また，提案手法を HiFP2.0 と比較し，その性能の改善や悪化について考察した．検索失敗率は提案手法の HiFP2.1 は同実装の HiFP2.0 と比較して最大 0.000175 倍となり大幅に改善することが分かった．一方で実行時間は 30 秒の楽曲データに対して提案手法の実装では，同実装の HiFP2.0 の処理時間と比べて処理時間は 2.069 倍となることが分かった．検索精度の向上が著しいため，実行時間の悪化は許容できると考えられる．しかし，望ましくないことは事実であるので，サンプルデータの抽出範囲を限定する手法などについて調査したい．

次章では，今まで述べた全ての章のまとめと今後の課題について述べる．

第5章　結論

5.1　まとめ

本研究では，ハードウェア上で離散ウェーブレット変換を用いたフィンガープリント生成を行う HiFP2.0 の問題点を改良した HiFP2.1 および，その chunk 抽出方式の決定法や最適 chunk 数の決定法に関する提案を行った．HiFP2.0 の楽曲データの冒頭 2.97 秒間から行う局所的なサンプル抽出を，楽曲全体において chunk 領域を周期的に散在させることでサンプルデータを楽曲全体から抽出し，確率的に問題発生個所を回避する手法に拡張したことによって，処理時間は原理的に楽曲の大きさに合わせて増加するようになったが，歪みに対するロバスト性が上昇した．chunk 領域抽出における実装手法及び chunk 領域数の決定方法としては，提案手法の「FPGA 上でサンプル抽出を行う HiFP2.1 の FPGA 実装」かつ「chunk 数 1024」の HiFP2.1 が最適であることが分かった．具体的には，検索失敗率は提案手法の HiFP2.1 は同実装の HiFP2.0 やその他の HiFP2.1 実装と比較して最大 0.000175 倍となり大幅に改善することが分かった．一方で実行時間は 30 秒の楽曲データに対して提案手法の実装では，同実装の HiFP2.0 の処理時間と比べて処理時間は 2.069 倍となることが分かった．これによって，HiFP2.1 アルゴリズムによってサンプル抽出領域を楽曲の広範囲に持つことで HiFP2.0 と比較して検索精度が大幅に向上することが分かったと同時に，実行時間が悪化することが分かった．

5.2　今後の課題

現状では，提案手法 HiFP2.1 は楽曲データ全体を使用する関係上，HiFP2.0 よりも実行時間は長くなる．ただ，HiFP2.0 からの検索精度の向上が著しいため，実行時間の悪化はある程度許容できると考えられる．しかし，望ましくないことは事実である．そこで，楽曲データ全体に対するサンプル抽出範囲と検索精度の関係性を調査し，サンプル抽出範囲の増加に対する検索精度の高止まりなどの事象を発見できれば，それを元にサンプル抽出範囲を楽曲全体でなく一定の範囲に限定できれば，処理速度の改善が見込めるものと考えられる．

また，HiFP2.1 アルゴリズムは，FPID を生成するための 2.97 秒間のサンプル抽出領域を chunk 領域として細分化して楽曲データ全体に分散させる仕様上，楽曲データの時間的なずれの発生した場合，わずかなずれであっても chunk 領域が

配置される地点が全く異なってしまう．本来，Shazam のように楽曲データ全体に対して部分的に切り抜かれたようなクエリデータであっても識別できるアルゴリズムの方が汎用性が高く望ましい．このように HiFP アルゴリズムの用途を拡張することを考えた場合，この脆弱性は解決されなければならない．よって，HiFP で使われている離散ウェーブレット変換のサブバンド分解のランドマーク型への応用と FPGA を用いた実装に取り組みたい．

また，HiFP2.1 の回路を FPGA 上で複数生成し，並列化させることで高速化を図っていきたい．

本研究に関する発表論文

[1] 山名 友也, 井口　寧, ”ハードウェアにおける高速なオーディオフィンガープリントを用いた楽曲全体のマッチング手法” 令和元年度北陸地区学生による研究究発表会, 2020.

謝 辞

本研究を行うにあたりまして，様々な御助言や実装，実験環境の機器調達および手解きなど多くの手厚い御指導を賜りました北陸先端科学技術大学院大学の情報社会基盤センター井口 寧教授へ深い感謝とともに御礼を申し上げます.

また，ゼミなどにおいて多くの御助言を頂きました，河野 隆太助教授に深く感謝致します.

また，中間審査会などで様々な御助言を頂きました，金子　峰雄教授，田中 清史教授に心から感謝致します.

また，研究室の現所属学生，卒業生あるいは元所属学生であり，私の研究に御協力頂いた河村 知記，Nguyen Mau Toan，Kien Chi Vu，NGUYEN, Minh Tien，稲葉 貴大，多田 大希，齊藤 正章，大塚 達史，齋藤 卓磨，岩田 拓也，根田 巧，横山 政巨，Faiz Al Faisal の皆様にも心から感謝致します.

また，副テーマ研究において適切な御指導を頂きました篠田 陽一教授にも心から感謝致します.

最後に，ここまで私を支えてくださった家族の皆様に心から感謝致します.

参考文献

[1] [Online]. Available: https://www.tensorflow.org/?hl=ja

[2] 高田高志 and 吉澤千和子, "調査研究ノート　フィンガープリント導入への道程," **放送研究と調査**, vol. 68, no. 2, pp. 74–83, 2018. [Online]. Available: 10.24634/bunken.68.2_74

[3] P. Cano, E. Batlle, H. Mayer, and H. Neuschmied, "Robust sound modeling for song detection in broadcast audio," 2002.

[4] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system with an efficient search strategy," *Journal of New Music Research*, vol. 32, no. 2, pp. 211–221, 2003. [Online]. Available: 10.1076/jnmr.32.2.211.16746

[5] J. Haitsma and T. Kalker, "Speed-change resistant audio fingerprinting using auto-correlation," in *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).*, vol. 4, 2003, pp. IV–728.

[6] Y. Liu, H. S. Yun, and N. S. Kim, "Audio fingerprinting based on multiple hashing in dct domain," *IEEE Signal Processing Letters*, vol. 16, no. 6, pp. 525–528, 2009.

[7] M. D. Kamaladas and M. M. Dialin, "Fingerprint extraction of audio signal using wavelet transform," in *2013 International Conference on Signal Processing , Image Processing Pattern Recognition*, 2013, pp. 308–312.

[8] S. Kim and C. D. Yoo, "Boosted binary audio fingerprint based on spectral subband moments," in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, vol. 1, 2007, pp. I–241–I–244.

[9] I. Schmädecke and H. Blume, "Hardware-accelerator design for energy-efficient acoustic feature extraction," pp. 135–139, Oct 2013. [Online]. Available: 10.1109/GCCE.2013.6664775

[10] H. Schreiber, P. Grosche, and M. Müller, "A re-ordering strategy for accelerating index-based audio fingerprinting," 2011. [Online]. Available: 10.5281/zenodo.1417607

[11] A. C. Ibarrola and E. Chavez, "A robust entropy-based audio-fingerprint," in *2006 IEEE International Conference on Multimedia and Expo*, 2006, pp. 1729–1732.

[12] 荒木光一, 佐藤幸紀, V. Jain, and 井口寧, "ハードウェアにおける高速なオーディオフィンガープリント生成システムの性能評価," **先進的計算基盤システムシンポジウム***: SACSIS 2010* **論文集**, vol. 2010, no. 5, pp. 295–302, may 2010. [Online]. Available: http://hdl.handle.net/10119/9551

[13] A. Wang, "An industrial strength audio search algorithm." 01 2003.

[14] S. Fenet, G. Richard, and Y. Grenier, "A scalable audio fingerprint method with robustness to pitch-shifting," 10 2011, pp. 121–126.

[15] X. Anguera, A. Garzon, and T. Adamek, "Mask: Robust local features for audio fingerprinting," in *2012 IEEE International Conference on Multimedia and Expo*, 2012, pp. 455–460.

[16] M. Jia, T. Li, and J. Wang, "Audio fingerprint extraction based on locally linear embedding for audio retrieval system," *Electronics*, vol. 9, no. 9, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/9/1483

[17] J. George and A. Jhunjhunwala, "Scalable and robust audio fingerprinting method tolerable to time-stretching," pp. 436–440, July 2015. [Online]. Available: 10.1109/ICDSP.2015.7251909

[18] T. Jie, L. Gang, and G. Jun, "Improved algorithms of music information retrieval based on audio fingerprint," in *2009 Third International Symposium on Intelligent Information Technology Application Workshops*, 2009, pp. 367–371.

[19] C. V. Cotton and D. P. W. Ellis, "Audio fingerprinting to identify multiple videos of an event," in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2010, pp. 2386–2389.

[20] Yan Ke, D. Hoiem, and R. Sukthankar, "Computer vision for music identification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, pp. 597–604 vol. 1.

[21] S. Baluja and M. Covell, "Audio fingerprinting: Combining computer vision data stream processing," in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, vol. 2, 2007, pp. II–213–II–216.

[22] B. Zhu, W. Li, Z. Wang, and X. Xue, "A novel audio fingerprinting method robust to time scale modification and pitch shifting," in *Proceedings of the 18th ACM International Conference on Multimedia*, ser. MM '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 987–990. [Online]. Available: https://doi.org/10.1145/1873951.1874130

[23] M. Malekesmaeili and R. K. Ward, "A novel local audio fingerprinting algorithm," in *2012 IEEE 14th International Workshop on Multimedia Signal Processing (MMSP)*, 2012, pp. 136–140.

[24] K. Kashino, A. Kimura, H. Nagano, and T. Kurozumi, "Robust search methods for music signals based on simple representation," in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, vol. 4, 2007, pp. IV–1421–IV–1424.

[25] H. Nagano, R. Mukai, T. Kurozumi, and K. Kashino, "A fast audio search method based on skipping irrelevant signals by similarity upper-bound calculation," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 2324–2328.

[26] C. Saravanos, D. Ampeliotis, and K. Berberidis, "Audio-fingerprinting via dictionary learning," in *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*, 2020, pp. 1–7. [Online]. Available: 10.1109/MMSP48831.2020.9287073

[27] H. Khemiri, D. Petrovska-Delacrétaz, and G. Chollet, "A generic audio identification system for radio broadcast monitoring based on data-driven segmentation," in *2012 IEEE International Symposium on Multimedia*, 2012, pp. 427–432.

[28] A. Ramalingam and S. Krishnan, "Gaussian mixture modeling of short-time fourier transform features for audio fingerprinting," *IEEE Transactions on Information Forensics and Security*, vol. 1, no. 4, pp. 457–463, 2006.

[29] J. S. Seo, Minho Jin, Sunil Lee, Dalwon Jang, Seungjae Lee, and C. D. Yoo, "Audio fingerprinting based on normalized spectral subband moments," *IEEE Signal Processing Letters*, vol. 13, no. 4, pp. 209–212, 2006.

[30] [Online]. Available: http://xillybus.com/

[31] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR*, vol. abs/1609.03499, 2016. [Online]. Available: http://arxiv.org/abs/1609.03499

[32] [Online]. Available: https://www.analyticsvidhya.com/blog/2020/01/how-to-perform-automatic-music-generation/

[33] [Online]. Available: http://www.piano-midi.de/midicoll.html