東京大学
情報理工学系研究科 創造情報学専攻
修士論文

BlueScript 向けのマイコン上での ZIP 解凍による SRAM 消費量を削減する手法

Reducing SRAM Consumption Used in ZIP Decompression on Microcontrollers for BlueScript

渡邉 純一
Junichi Watanabe

指導教員　千葉 滋 教授

2025 年 2 月

概要

BlueScript はマイコン向けに開発された新しいアーキテクチャを採用した言語処理系であり、非常に高い対話性を持っている．BlueScript はアプリケーションを実行する際、ホストマシンでコンパイルしたネイティブコードを BLE によってマイコンへ転送する．しかし、ライブラリや比較的大きなアプリケーションを実行する際、コードサイズが増加してしまうため、低速な BLE がボトルネックとなり対話性が著しく損なわれてしまう．そのために ZIP を利用して転送サイズを減らしたいが、 ZIP 解凍はマイコンの限られた SRAM を多く消費してしまう．そこで本論文では ZIP 解凍による SRAM 消費量を削減しつつ、圧縮率を向上させ対話性を改善するために 3 つの手法を提案する．実験を通して、提案手法によって SRAM 消費量を既存手法から 99.70 % 削減できたと同時に、圧縮率が向上したことにより、対話性をより改善できたことが確認できた．

Abstract

BlueScript is a language developed with a new architecture for microcontroller units (MCUs), offering exceptionally high interactivity. When executing an application in BlueScript, it compiles native code on the host machine and transfers it to the MCU via BLE. However, when running libraries or relatively large applications, the code size increases, making the slower BLE a bottleneck that significantly degrades its interactivity. Using ZIP compression for reducing the size seems to solve this issue. However, ZIP decompression consumes a significant amount of the MCU's limited SRAM. This paper proposes three methods to reduce SRAM consumption during ZIP decompression, while simultaneously improving compression efficiency to enhance interactivity. Through experiments, it was confirmed that the proposed methods reduced SRAM consumption by 99.70 % compared to existing methods. At the same time, the improved compression efficiency further enhanced interactivity.

目次

第 1 章

はじめに

BlueScript [1] はマイコン向けに開発された新しいアーキテクチャを採用した言語処理系であり、 Jupyter Notebook のような開発環境と TypeScript のような文法を提供する．また、通常の利用ではユーザがアプリケーションを実行してから結果が返ってくるまでの時間が短いため、この高い対話性が BlueScript の重要な性質の一つとなる．

BlueScript は Bluetooth Low Energy (BLE) によってホストマシン上でコンパイルしたアプリケーションのネイティブコードをマイコンへと転送し実行するが、ここで使用する BLE は電力消費が低い代わりに帯域幅が小さいという特徴を持つ．つまり、ユーザが比較的大きなアプリケーションやライブラリのコードを転送する際に、BLE の通信時間がボトルネックとなることで対話性が損なわれてしまう可能性がある．

そのため、転送サイズを減らすために ZIP [2] や ZStandard [3] を使用することが考えられる．マイコンには高速だが小さい SRAM と低速だが大きいフラッシュメモリがあるが、フラッシュメモリは書き込み速度が SRAM の 379.6 倍遅いだけでなく、書き込み回数に制限があるため、解凍のために SRAM を消費する必要がある．しかし、BlueScript ではネイティブコードは SRAM 上に配置されているが、解凍による SRAM の消費量が大きいため、これらの手法を利用することで対話性は向上するものの、BlueScript で実行可能なアプリケーションのコードサイズに大きな制約を与えてしまうことになる．

そこで、本論文では ZIP 解凍によるマイコン上の SRAM 消費量を削減しつつ、圧縮率を向上させ対話性をより改善するために 3 つの手法を提案する．まず、SRAM 上に配置されたネイティブコードを LZ77 辞書としてみなすことで、LZ77 辞書による SRAM 消費量を削減する。次に、解凍のたびにハフマンテーブルを送る代わりに、フラッシュメモリ上に事前作成されたテーブルを使用することで、動的に構築する際の SRAM 消費量を削減する。最後に、ソフトリセット時の古いネイティブコードをフラッシュメモリにコピーし、LZ77 の辞書として再利用することで、SRAM を消費せずに圧縮率を向上させる。

実験により、本提案手法は既存手法よりも大幅に SRAM 消費量を削減できたと同時に、対話性の改善も既存手法の水準程度以上達成できたことを確認した．

本論文の構成は次のようになっている．第 2 章では BlueScript のアーキテクチャ、BLE 通信の速度と懸念点、一般的な可逆圧縮、BlueScript で ZIP を利用する際の問題点、その他

のマイコン向けコード転送高速化の技術について説明する．第 3 章では 2 章で述べた問題点に対する解決策として、先ほど述べた 3 つの提案案手法について解説する．第 4 章では 3 章で紹介した提案手法の有効性を示すために行なった実験の結果と、それに対する考察を載せる．第 5 章ではこれまでの内容を踏まえた本論文の結論を述べる。

第 2 章

背景

2.1 BlueScript

BlueScript はマイコン向けに開発された新しいアーキテクチャを採用した対話的な言語処理系である．ホストマシン上での効率的なマイコンのアプリケーション開発を目的としており、図 2.1 で示されている通り、 Jupyter Notebook のような開発環境と TypeScript に似た文法を提供している．また、ユーザが BlueScript でマイコンのアプリケーションを開発する流れは次のとおりである：

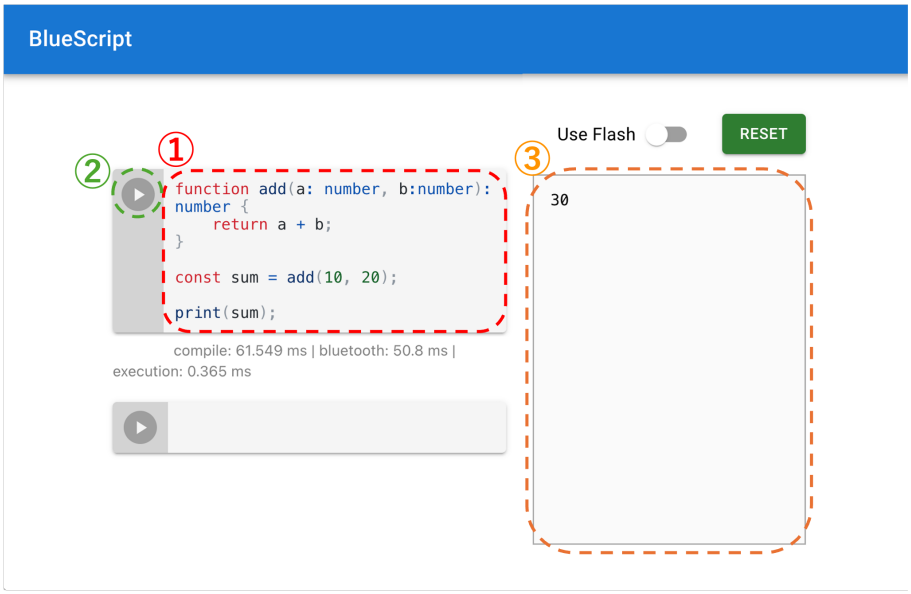

図 2.1. BlueScript における開発画面と開発の流れ

1. 赤枠で囲われたセルと呼ばれるプログラミング可能なスペースに、マイコン上で実行したいアプリケーションプログラムを BlueScript で記述する．
2. 緑枠で囲われた実行ボタンを押し、ホストマシンからマイコンへアプリケーションを転

送する．

3. マイコン上にアプリケーションを展開し、実行が完了すると、実行結果がマイコンからホストマシンへと送信され、黄枠で囲われたログに表示される．

BlueScript ユーザはこれらを開発の 1 サイクルとして、アプリケーション開発の間繰り返していく．図 2.1 のような通常の開発であれば、実行ボタンを押してから結果が返ってくるまでの時間は 1 秒未満である．このような高い対話性が BlueScript における重要な性質の一つになる．

次に BlueScript のアーキテクチャについて説明する．図 2.2 は実行ボタンが押されてからマイコン上でアプリケーションが実行されるまでの間の、バックエンド側で行われる処理の流れを示している．

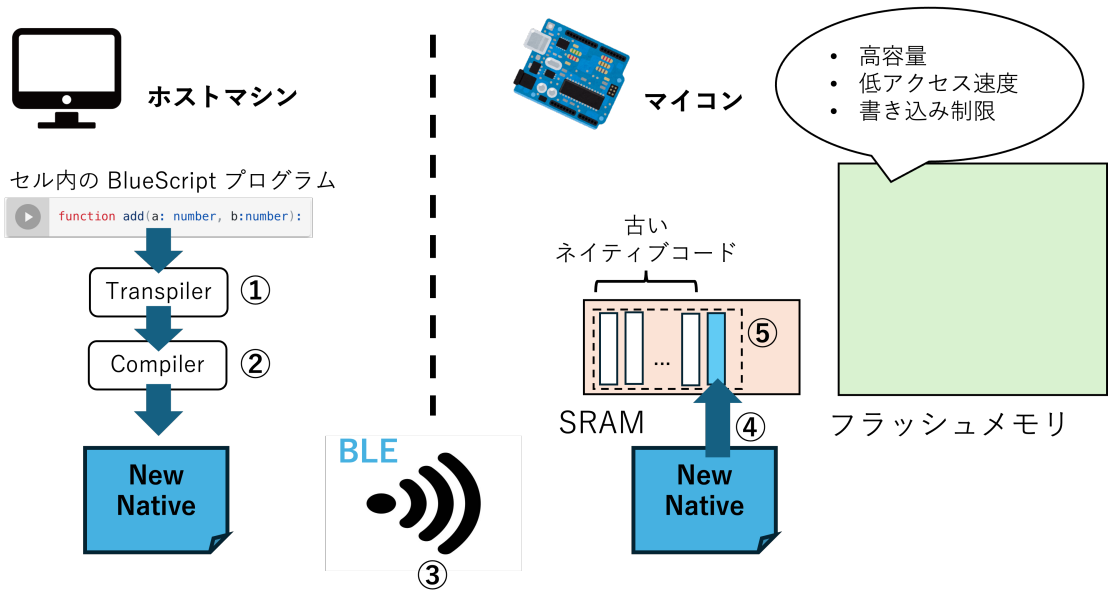

図 2.2. BlueScript でアプリケーションがマイコンで実行されるまでの流れ

1. セルに書かれた BlueScript プログラムを C 言語のプログラムへトランスパイルする．
2. C 言語のプログラムを xtensa-gcc によってネイティブコードへコンパイルする．
3. ネイティブコードを BLE を利用してホストマシンからマイコンへ転送する．
4. 転送されたネイティブコードを SRAM へロードする．
5. SRAM 上にロードされたネイティブコードを実行する．

BlueScript は高い対話性のために様々な工夫が施されている．インタプリタで各行を解釈し実行する代わりに、コンパイラによって生成されたネイティブコードを実行するため、MicroPython[4] のような他の対話的なマイコン開発環境よりもアプリケーションの実行速度が高速である．また、コンパイル機能をマイコン上ではなくホストマシン上にオフロードする

表 2.1. ESP32 における SRAM とフラッシュメモリのアクセス速度

	SRAM (A)	フラッシュメモリ (B)	(B) の (A) に対する倍率
読み出し速度	3,275 μs	18,305 μs	5.6
書き込み速度	2,886 μs	1,095,414 μs	379.6
実行速度	7,177 μs	51,710 μs	7.2

ことによって、より潤沢な計算資源とメモリ資源を駆使し、最適化されたネイティブコードを高速に生成することができる．

そして、高速なメモリデバイスを選択することにより、アプリケーションの高速実行を可能としている．一般的に製造コストの問題上、マイコンには高速だがサイズが小さい SRAM と、サイズが大きいが低速なフラッシュメモリという二種類のメモリデバイスを持つことが多い．例えば、BlueScript が利用可能な ESP32-D0WDQ6-V3[5] というマイコンでは、520 KB の SRAM と 16 MB のフラッシュメモリを持っている．ESP32-D0WDQ6-V3 の両メモリデバイスにおける実際のアクセススピードを調べるため、SRAM またはフラッシュメモリ上の配列・関数に対してそれぞれ 30,000 回読み出し・書き込み・実行にかかった時間を測定し、その結果を表 2.1 に示す．ここで測定用に用意した関数は何も実行しないですぐに戻るものである．ここで留意すべきは ESP32-D0WDQ6-V3 では 64 KB のキャッシュメモリのフラッシュメモリはを持っているということである．キャッシュメモリによってフラッシュメモリアクセスが高速化され、実際のフラッシュメモリの速度が計測されないことを防ぐために、配列や関数をキャッシュメモリサイズである 64 KB 分飛び飛びでアクセスするようにした．

表 2.1 からわかる通り、フラッシュメモリは SRAM よりも大幅にアクセス速度が遅いことがわかる．書き込み速度に関しては 379.6 倍遅く、BlueScript では実行ボタンが押されるたびにマイコンにネイティブコードを書き込む必要があるため、フラッシュメモリ上に書き込むと対話性が損なってしまうことがわかる．一方で、読み出し速度と実行速度はそれぞれ 5.6 倍と 7.2 倍遅い．マイコンでは SRAM は希少であり、大容量なフラッシュメモリを使用することで SRAM の節約が可能となるため、フラッシュメモリをアクセス頻度が低く、静的なデータやコードを保存する領域として利用することができることがわかる．

ここで特筆すべきことは、多くのマイコンがハーバードアーキテクチャ型の CPU を持つということである．ハーバードアーキテクチャは、データメモリとプログラムメモリという二種類のメモリがそれぞれデータバスとインストラクションバスと呼ばれる独立したバスで接続されている CPU アーキテクチャである．マイコンでは通常 CPU の性能が低いため、データと命令コードを並列にアクセスすることでスループットの向上とメモリ競合の削減による恩恵が大きい．また、処理時間の短縮はそのまま消費電力の低下へとつながり、このアーキテクチャではマイコンで貴重な電源を節約することができる．そして、データと命令コードの特徴に合わせてメモリデバイスを適切に選択することで、コストを最適化し、マイコンの単価を下げることができる．例えば、 ESP-D0WDQ6-V3 においてはデータバスに 200 KB の SRAM

がデータメモリ、インストラクションバスに 320 KB の SRAM がプログラムメモリとしてそれぞれ独立に接続されている．特に ESP-D0WDQ6-V3 においてデータメモリは DRAM 、 プログラムメモリは IRAM と呼ばれる．ハーバードアーキテクチャにおいてインストラクションバスは 4 バイトアラインメントでアクセスされる必要があり、そこに接続されている 320 KB の IRAM は汎用コンピュータのようにバイトアクセスすることができない．つまり、マイコン上で SRAM を扱う場合は両者における差を認識しながらプログラムを書く必要がある．

これまで BlueScript の開発の流れ、重要な性質である対話性、そして対話性を支える技術的工夫について説明したが、対話性を維持する上で注意しなければならない要素がもう一つ存在する．それは BLE による通信時間である．先述した通り、 図 2.1 のセルに書かれたような比較的小さなプログラムフラグメントを実行する場合における待ち時間は 1 秒未満である．しかし、ユーザが比較的大きなプログラムをセル内に書き込む可能性は十分に存在し、また BlueScript にライブラリのインポート機能が登場することを考え、大規模なネイティブコードを BLE で転送する場合の時間について考慮していく必要がある．

2.2　BLE の通信速度と消費エネルギー

BLE における通信速度を測定するために、ホストマシンから ESP32-D0WDQ6-V3 へ 5 KB から 100 KB のデータを転送する時間を測定し、その結果を図 2.3 の (A) に示す．この結果により、 BLE の平均通信速度は 10 [KBps] 程度であることがわかり、例えば 20 KB のネイティブコードをホストマシンからマイコンへ転送する場合、 BlueScript は転送だけで 2 秒間待つ必要があり、またデータサイズが増加していくと線型的に待ち時間も増加し、BlueScript の対話性という大きな性質を損なってしまうことがわかる．

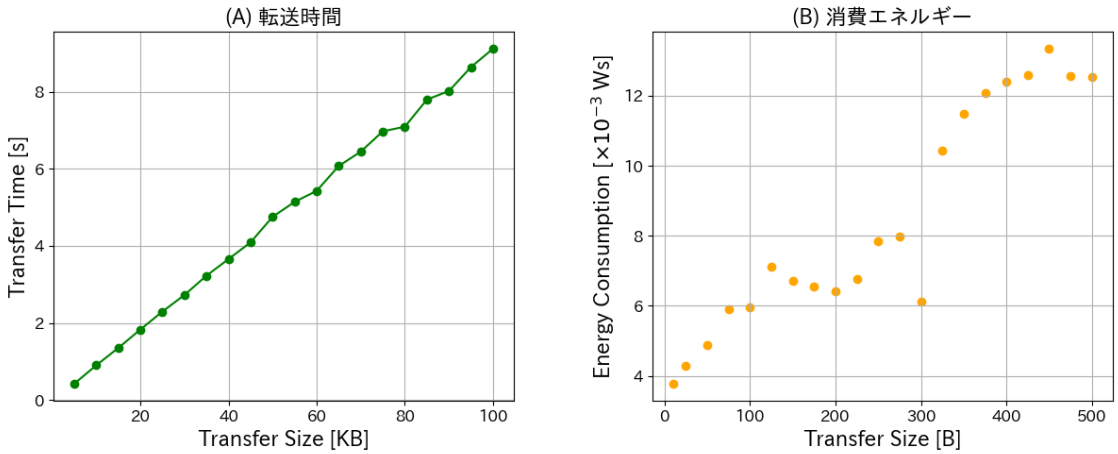

図 2.3. BLE 通信において異なるサイズのデータを転送した際の転送時間と消費エネルギー

また、大きいネイティブコードを転送するために BLE の通信回数が増加することによっ

て、マイコンの稼働時間が短くなるという問題が発生する．一般的にマイコンは実地に展開されると電源にアクセスすることはできなくなるため、バッテリーを取り付けてそれにより駆動する．つまり、エネルギー消費の高い処理が続くと、それだけバッテリーを消耗し、結果的にマイコンの稼働時間を短縮させてしまう．図 2.3 の (B) に 20 B から 500 B のホストマシンからマイコンへ BLE で転送した際の消費エネルギーを計測した結果を示している．計測の最大値を 500 B に設定したのは BLE のパケットサイズの最大値は 500 B であり、それ以上の場合は単純にパケット数に 500 B 転送する場合の消費エネルギーを掛け合わせることで算出できるためである．これにより、パケットサイズが増加する際、確かにエネルギー消費も線型的に増加したことが確認できる．

2.3 可逆圧縮アルゴリズム

このような帯域幅が小さい通信路でのデータ転送を高速化する手法として、可逆圧縮アルゴリズムが考えられる．可逆圧縮アルゴリズムとは、圧縮データを圧縮前のデータへ一切の情報の損失なく再構築することができる圧縮アルゴリズムであり、圧縮前データを再構築する処理を解凍と呼ぶ．本論文では可逆圧縮アルゴリズムの中からハフマン符号化、Lemple-Ziv アルゴリズム、ZIP や ZStandard を紹介する．

ハフマン符号化 [6] は、入力データの各文字を、出現頻度に基づき割り当てられた符号に置き換える可逆圧縮アルゴリズムである．圧縮をする際に、エンコーダはまず入力データを文字と呼ばれる単位に分割する．次に、入力文字列内の各文字の出現頻度を頻度表に記録していく．そして、頻度表からハフマン木と呼ばれる木構造を生成する．ハフマン木における葉は各文字に対応しており、根から葉への道が符号、高さが符号長という特徴を持っている．ハフマン木によって記される符号は文字列での出現頻度が高い文字には短い符号、高い文字には長い符号が割り当てられるようになっている．これによって、文字列の大部分をより短い符号へと変換し、全体のサイズを削減することが可能となる．最後に、エンコーダはハフマン木をハフマンテーブルと呼ばれる文字と符号の対応表に変換し、文字列の各文字に対して逐次的に符号への変換を行うことで、圧縮された符号列を出力する．一方で、解凍をする際は、デコーダはエンコーダからハフマンテーブルを共有されている必要があり、それを元に符号列の各符号を文字へと変換していくことで、圧縮前の文字列を取得することができる．ハフマン符号化の特徴としては、文字の出現頻度が偏っている入力に対しては高い性能を出せる一方で、分布が均等に近しい場合はあまり高い性能を出すことができない．

Lemple-Ziv アルゴリズムは入力データ内で繰り返される冗長なパターンをより短い表現に置き換えるアルゴリズムであり、 LZ77 [7] や LZ78 [8] のようにさまざまなバージョンが公開されてきた．まず、 LZ77 について説明する．圧縮をする際、エンコーダは入力データを文字列へと変換し、過去に同じパターンが出現したかどうかを探索しながらカレント・オフセットを移動させる．直近の過去データは辞書と呼ばれるメモリ領域に保持されており、カレント・オフセットから始まる部分文字列との最長の一致文字列を探索する．見つかった最長一致文字列は一致距離と一致長から構成される参照で表す．一致距離は最長一致文字列の開始オフセッ

トとカレント・オフセットの差であり、一致長はその長さである．そしてカレント・オフセットから一致長分先の文字を不一致文字とする．最後にエンコーダは (一致距離、一致長、不一致文字) を出力し、一致文字列と不一致文字を辞書に保存し、またカレント・オフセットを (一致長 + 1) 分進めることで、冗長なパターンを参照という効率的な表現に置き換えることができる．一方で、解凍する場合は、参照による最長一致文字列の出力、不一致文字の出力、辞書のアップデートを繰り返していくことで、圧縮前の文字列を取得することができる．LZ77 はパターンが明確な構造化されたデータに対しては高い性能を示すことができる一方で、ランダム性の高いデータに対してはデータサイズが返って増加する可能性がある．また、圧縮率の性能は辞書のサイズによって左右されるが、サイズが大きすぎると、メモリ消費量と探索時間の増加というトレードオフが発生してしまう．

次に LZ78 について説明する．圧縮をする際、 LZ77 と同じように冗長なパターンを探索するのだが、使用する辞書の構造が異なる．LZ77 では辞書は直近の過去データを辞書サイズ分保持していたが、 LZ78 では今まで出現してきたパターンのテーブルとなっている．エンコーダは入力文字列の部分文字列に対して一致する最長のパターンをテーブル参照によって探索する．そして、冗長なパターンを参照の代わりに一致するパターンの辞書インデックスに置き換えることでデータを圧縮する．この時、一致パターンを見つけることができなかった場合にインデックス 0 が出力され、辞書に新しいパターンとして登録される．解凍する際は、辞書を構築しながら、辞書インデックスかのパターンを出力と不一致文字を出力を繰り返すことで、圧縮前の文字列を取得することができる．LZ78 は LZ77 と違い、辞書は入力全体のパターンが網羅的に探索される一方で、より細かな探索が行われないため、 LZ77 よりも圧縮率で劣る場合がある．更に、辞書は動的に構築されていくため、短い実行ではメモリ消費量は LZ77 よりも優れるが、長い時間実行されるとメモリ消費量が上回る可能性がある．

ZIP は高い圧縮率と適度な実行速度を達成するため、 LZ77 符号化とハフマン符号化を組み合わせた可逆圧縮アルゴリズムである．ZIP エンコーダは入力データをブロックと呼ばれる単位に分割し、各ブロックに対して独立して圧縮を行う．ZIP エンコーダは LZ77 エンコーダとハフマンエンコーダがこの順で直列で接続されており、ブロックデータは最初に LZ77 エンコーダによって符号化される．この時、 LZ77 エンコーダは 32 KB の固定サイズの辞書を有している．LZ77 符号列がハフマンエンコーダへと渡され、符号化されるが、ここで ZIP では LZ77 符号を **一致長・不一致文字** 、 **一致距離** という二種類のハフマン入力文字グループに分け、それぞれのグループにおいて独立してハフマン符号化を行なっていく．

ZIP ではハフマン符号化は最適ハフマン符号化と静的ハフマン符号化の二種類存在している．最適ハフマン符号化はブロックが更新されるたびに、今までのハフマンテーブルを破棄して、新しい LZ77 符号列からハフマンテーブルを構築する手法である．静的ハフマン符号化は ZIP 圧縮・解凍の状況にかかわらず、 ZIP の仕様書上に定義されているハフマンテーブルを利用する手法である．最適ハフマン符号化は符号化するブロックの LZ77 符号列における各文字の出現頻度を反映させるため、圧縮率がより高く、 ZIP においては通常はこれを利用する．しかし、入力データサイズが小さい場合では、文字間における出現頻度のばらつきが小さくなるため、結果的に圧縮率の観点から最適ハフマン符号化の静的ハフマン符号化に対する優

位性が薄れてしまう．また、最適ハフマン符号化は静的ハフマン符号化とは異なり、エンコーダはデコーダにハフマンテーブルの情報を共有する必要があるため、ハフマンテーブルをヘッダに含める位必要がある．以上の観点から、入力データサイズが小さい場合において、最適ハフマン符号化の代わりに静的ハフマン符号化や、 LZ77・ハフマン符号化を行わない生データの直接出力が選択される場合がある．最後にブロックの終わりを示す終端記号を追加することによってそのブロックにおける ZIP 圧縮データが出力される．

ZIP デコーダはエンコーダとは逆にハフマンデコーダ・ LZ77 デコーダの順に直列接続されており、 ZIP 圧縮データを読み込み始めると、まずヘッダを読み込む．ヘッダからそのブロックが最適ハフマン符号化を利用しているのか、静的ハフマン符号化を利用しているのか、生データをそのまま出力しているのかを判断する．もし生データブロックであるのなら、ブロックの終端記号に到達するまでハフマンデコーダ・ LZ77 デコーダを通さずに直接出力し続ける．もし静的ハフマン符号化であるのなら、 ZIP の仕様書によって定義されている静的ハフマンテーブルをロードする．もし最適ハフマン符号化であるのなら、ヘッダから先述した二つのハフマンテーブルをヘッダからリゾルブする．次に、ハフマンデコーダが ZIP 圧縮データを LZ77 符号列へと復号し、終端符号に到達するまで LZ77 デコーダへと渡し続ける．そして LZ77 デコーダは LZ77 符号列から元データにおける部分データを複合する．この時、LZ77 デコーダはエンコーダ同様 32 KB の固定サイズ辞書を持っており、部分データを復号すると同時にこの辞書へと書き込み次の復号で利用できるようにする．LZ77 デコーダは終端符号に到達するまでこの処理を繰り返すことによって圧縮データを元データに解凍する．

ZStandard は Facebook によって開発された ZIP よりも高い圧縮・解凍速度と圧縮率を目指して開発された可逆圧縮アルゴリズムである．ZStandard エンコーダでは LZ77 エンコーダの後ろに ハフマンエンコーダと Finite State Entropy (FSE) エンコーダが並列接続されている．FSE 符号化はハフマン符号かと同じエントロピー符号化の一種であるが、各入力文字の出現頻度に加え、符号化するごとにエンコーダを状態遷移して追加の情報を利用することによって、ハフマン符号化よりも高い圧縮率が達成できる．ZStandard エンコーダは ZIP エンコーダ同様、ブロックデータをまず LZ77 エンコーダで LZ77 符号列に符号化する．次に ZIP エンコーダとは異なり、 LZ77 符号列を **一致距離・不一致文字** 、 **一致長** の二つのハフマン文字グループに分類し、前者をハフマンエンコーダ、後者を FSE エンコーダへと渡す．ハフマンエンコーダは ZIP 同様、最適ハフマン符号化・静的ハフマン符号化から選択される．最後にハフマンエンコーダ・ FSE エンコーダの出力結果に基づいて、データサイズが悪化するのであれば、無圧縮データを出力し、そうでなければ、ハフマン・ FSE テーブル、ハフマン・ FSE 符号列、終端記号を出力することで ZStandard 圧縮データを出力する．

ZStandard デコーダでは ZStandard エンコーダとは逆順に各符号化のデコーダを接続している．ZIP デコーダ同様、まずヘッダからハフマンテーブルと FSE テーブルを再構築し、各デコーダに渡す．次に入力データからハフマン符号列・ FSE 符号列の順に読み込み、各デコーダへと入力する．そして終端記号に到達するまで、両デコーダから LZ77 エンコーダへ LZ77 符号列を入力し続けることで、元データへと解凍することができる．

ZStandard は ZIP とは異なるいくかの特徴が存在している．ZStandard は圧縮レベルを調

表 2.2. MINIZ と ZStandard における圧縮と解凍時に必要なデータサイズ

	MINIZ	ZStandard
圧縮に必要なデータサイズ	319.352 KB	40.920 KB
解凍に必要なデータサイズ	41.168 KB	95.968 KB

節することで、 ZIP よりも高速な圧縮・解凍処理、または高い圧縮率を達成することができる．ZStandard が処理時間と圧縮率のトレードオフを利用している部分として LZ77 符号化時の最長一致文字列探索が挙げられる．また、ZStandard では並列処理環境において、圧縮・解凍処理速度の最適化を行うことができる．例えば、ハフマン・ FSE 符号化・復号化の際に入力データをいくつかのデータ列に分割し、それぞれ独立して符号化・復号化を行うことで、マルチコア環境において各コアに処理を分散することができる．この最適化は符号化テーブルの数が増加するため、ヘッダが増大し、圧縮データサイズでオーバヘッドが発生するが、入力データが大きくなるにつれて、ヘッダの全体に対する割合が小さくなるため、影響が薄るれていく．最後に、通常圧縮元のデータのサイズが小さい場合、 LZ77 符号化において、有効なパターンが辞書内に累積されていないため、高い圧縮率を出しにくいが、 ZStandard では類似したデータ群に対してブロックの外部辞書を構築することで、有効なパターンを増やし、結果的に小さい入力データの圧縮率を向上することができる．

2.4　BlueScript の対話性を改善するために ZIP・ZStandard を利用する際の問題点

先述したように BlueScript では比較的大きなアプリケーションを実行する場合、BLE の転送速度が低いことにより通信時間が増加し、結果的に対話性を損なってしまうという問題を解決するために、 ZIP や ZStandard を使用して転送サイズを削減したいが、解凍時にマイコン上の SRAM を多く消費してしまうと言う障害が発生してしまう．

表 2.2 に MINIZ[9] と ZStandard それぞれにおける圧縮・解凍に必要なメモリ量を示している．MINIZ は本論文で使用するマイコン、 ESP32-D0WDQ6-V3 を開発するためのフレームワークである ESP-IDF[10] で利用可能な ZIP アルゴリズムの実装である．メモリ消費量の内訳としては、 MINIZ においては圧縮・解凍時のコンテキストを保持する *tdefl_compressor* と *tinfl_decompressor* という構造体、 ZStandard においては *ZSTD_CCtx* と *ZSTD_DCtx* という構造体のサイズをそれぞれ載せている．

表 2.2 より解凍時にはメモリを 41.168 KB 消費することがわかる．このように広いメモリ領域が必要なのは、先述した LZ77 辞書とハフマンテーブル以外に、一致文字列探索を高速化させるハッシュテーブルや、デコードを高速化させるルックアップテーブルなどの様々なデータ構造が含まれているためである．さらに、これらのデータ構造は解凍が開始してから完了するまでの間、頻繁にアクセスされ続けるため、低速で書き込み回数に制限のあるフ

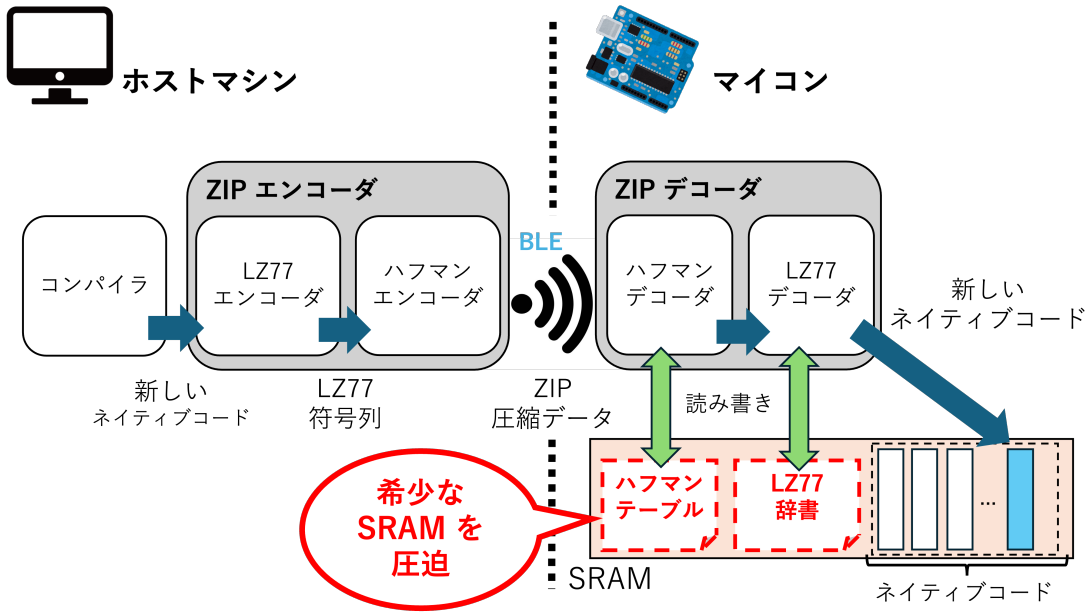

図 2.4. マイコン上に ZIP 解凍を行なった場合の問題点

ラッシュメモリではなく、 SRAM 上で確保される必要がある．しかしながら、先述した通り ESP32-D0WDQ6-V3 は 520 KB の SRAM しか持っておらず、さらに BlueScript のネイティブコードは SRAM 上に配置されているため、 BlueScript が実行可能なアプリケーションのサイズを大幅に狭めてしまう．図 2.4 はマイコン上で ZIP 解凍する際の流れを示しているが、解凍に必要なデータ構造によってネイティブコードの配置先が逼迫されていることがわかる．

ここで両手法における SRAM 上に確保するメモリサイズを単に減らせば良いと考えるかもしれないが、それでは対話性を改善することができない．図 2.5 は MINIZ と ZStandard で異なる LZ77 辞書のサイズで圧縮した際の圧縮率を示している．ここでの圧縮対象は BlueScript によってコンパイルされた合計 133 KB のネイティブコードを結合したものである．また、圧縮率は式 2.1 によって計算されており、圧縮率がより高ければ性能がより高いことを示している．

$$(\text{圧縮率}) = \frac{(\text{圧縮元データサイズ}) - (\text{圧縮データサイズ})}{(\text{圧縮元データサイズ})} \tag{2.1}$$

図 2.5 からわかる通り、 LZ77 辞書のサイズと圧縮率には正の相関が存在することがわかる．つまり、 SRAM を節約するために確保する領域サイズを小さくすると、圧縮率が下がり、転送データサイズの減少幅が小さくなり、結果的に対話性改善への寄与も小さくなってしまう．既存手法では SRAM 消費量と対話性の改善率がトレードオフ関係となってしまっていることがわかる．

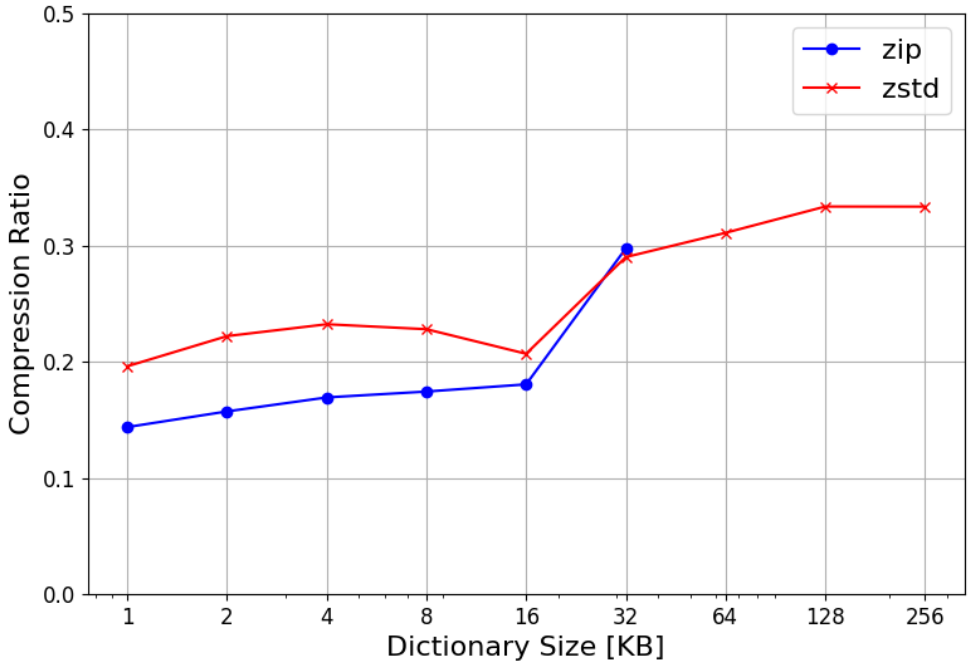

図 2.5. LZ77 辞書のサイズを変化させたときの ZIP と ZStandard の圧縮率

2.5　マイコンでデータ転送量を削減するためのその他の研究

近年では実地にある大量のマイコンによって構築される Internet of Things (IoT) という技術の発展し、それとともマイコンを利用する場面が増えている．また、無線通信モジュールを搭載したマイコンを安価に大量に製造することが可能となったため、 IoT を無線センサネットワーク上に構築することが一般的である．これにより、物理的な制約が減少し、スマートシティ、医療、そして産業オートメーションなど、さまざまな場面での利用が想定され、開発が進められている．マイコンは、 IoT 上での目的のアプリケーションによって、適切なファームウェアをフラッシュメモリ上にロードすることで、高度な機能を提供することができる．マイコンの開発者は、実地に展開された後でも、新しい機能の追加、ソフトウェアのバグやセキュリティ上の脆弱性の解消、アプリケーションの方向性の変化などによってファームウェアをアップデートする場面が多く存在している．先述した通り、マイコンは物理的なアクセスが難しい場所に展開されている場合もあるため、無線通信を通してファームウェアをアップデートする Over-The-Air Programming (OTAP) という機能が登場した．しかし、先述した通り、マイコンの無線通信帯域幅は低く、また充電できない状況で長時間稼働する必要があり、さらには CPU 資源とメモリ資源が限られているため、軽量でよりファームウェアに対して高い圧縮率を達成できるアルゴリズムやプロトコルが考案されてきた．

Contiki OS [11] は IoT 向けに開発された軽量な OS であり、初期バージョンでは動的リンカが導入された．これによってファームウェアのバージョンアップデートによって変更されたデータのみをホストマシンからマイコンへ転送し、マイコン上で動的リンクを行うことでファームウェアのアップデートを行えるようにした．この手法では直接的には圧縮は行われていないが、バージョンアップによって変化した部分のみを送ることで、転送データサイズを減らすことを達成している．しかしながら、これらの機能を利用するためにはアドレス解決可能である高度なリンカをマイコン上で実装する必要があり、またシンボルテーブルや再配置テーブルも転送データに含まれ、メモリ上で展開する必要があるため、計算リソース・メモリリソースを大きく消費すると同時に、差分データ以外のデータも転送する必要があるため、マイコン環境に最適化されたとは言えない．

Zephyr[12] は、無線センサネットワーク向けの効率的なインクリメンタル OTAP プロトコルであり、既存のファームウェアと新しいファームウェアとの差分を効率的に計算し転送することで、転送時間の短縮とエネルギー消費量の削減を行う．通常ファームウェアの異なるバージョン間では、バイナリの構造が類似しているため、差分を計算するだけで十分にデータサイズを圧縮することができる．Zephyr ではより効率的な差分計算のために、最適化された Rsync アルゴリズムを使用して差分スクリプトを生成する．差分スクリプトは圧縮前データを再構築するためにデコーダが実行する一連のコマンドを記述したものであり、マイコンは差分スクリプトを受け取ると既存のファームウェアから共通部分をコピー、または差分スクリプト上のバイトコードをそのまま出力することで新規のファームウェアを再構築する．そのため、ホストマシンは差分スクリプトを転送するだけでファームウェアのバージョンアップを行うことができるため、 Contiki OS とは異なり、シンボルテーブルや再配置テーブルなどの追加のデータを転送する必要がない．Zephyr はそれに加え、ファームウェアのバージョン間における構造の類似性を最大化するために、関数呼び出しの間接参照などを使用している．新しいファームウェアを生成する際、コードが挿入または削除されることにより、後続する関数の配置先アドレスが変化する関数シフトという現象が発生する．この関数シフトが発生してしまうと、その関数を呼び出す命令コード内のアドレスを全て書き換える必要があり、結果的に差分のサイズを増大させてしまう．しかし、間接参照を利用することで、すべての関数呼び出し命令のオペランドは対応す間接参照テーブルのエントリのアドレスに固定されるため、結果的に関数シフトが発生しても差分のデータサイズを抑えることができる．

これに加え、 Zephyr はマルチホップな無線センサネットワークへ対応し、より大規模なネットワークでの利用を可能とした．このような設計のおかげで、 Zephyr は Deluge[13] などの既存の OTAP プロトコルよりも、再プログラミングにおいて、最大で 1987 倍の性能向上を達成した．また、マイコン上で高度なリンカが不要となるため、 Contiki OS と比べ比較的軽量でファームウェアの再構築を行うことがわかる．

R3[14] は、 Zephyr におけるファームウェアの再構築処理によるオーバヘッドを削減することでより高い性能を目指したインクリメンタル OTAP プロトコルである．R3 では差分の生成手順を、ファームウェア間の類似性を高める前処理である R3sim と、効率的な差分スクリプト生成アルゴリズムである R3diff に分割できる．Zephyr では間接参照テーブルを利用

することで、関数シフトによる影響を緩和し、結果的にファームウェア間の類似性を高めたが、いくつか懸念点がある．まず、関数呼び出し命令しか対応していないため、関数シフトによる *mov* や *br* などの他の命令への影響は緩和できていない．次に、間接参照テーブルをメモリ上に確保する必要があるが、そのサイズは定義された関数の数に依存するため、開発が進むにつれてメモリ消費量が増大する．そして、同時に、間接参照テーブルによるメモリへのアクセス回数が増加することによって、アプリケーションの実行性能が低下してしまう．最後に、グローバル変数のアドレスが変化した際に発生するデータシフトという問題には全く対処していない．そこで、 R3sim ではまず異なるバージョンのファームウェアのバイナリを解析し、テキスト領域やデータ領域におけるシンボルのサイズ、アドレス、参照関係を特定する．次に、 R3sim は二つのバージョン間で配置先アドレスが変化しないように、関数やデータを適切に再配置し、さらにはそれらへの参照を新しいアドレスに直接書き換えることで、Zephyr と比べ、より多様の参照命令に対応でき、間接参照テーブルによる実行速度とメモリ消費量のオーバヘッドを回避し、データシフトによる影響も緩和できるようになり、資源の節約と同時により高い圧縮率を達成できている．

R3diff は二つのファームウェア間をバイト単位で比較し、一致バイト列と不一致バイト列を特定する．そして、一致するバイト列は一致位置と一致長からなるタプルを記録し、不一致バイト列はリテラルとして記録することで、効率的な圧縮を目指している．これは新しいファームウェアを入力データ、古いファームウェアを静的な辞書とした LZ77 圧縮と同じ処理の流れである．また、 R3diff は実行中には常に再帰的に最小の差分スクリプトサイズを計算することによって Zephyr に比べより効率的に圧縮される．これらの処理によって生成されたはデータはマイコンに転送され、元のファームウェアへと再構築するが、その処理に関しては Zephyr と共通している．

これらの OTAP で利用されている転送の高速化技術は差分アルゴリズムを利用していることが一般的であることがわかる．確かに対象がファームウェアであるとき、ファームウェアは厳密に構造化された比較的に大きなデータであるため、ファームウェア間の構造の類似性を高めることで圧縮率が高くなり、既存研究でもそのためにに色々と工夫されていることがわかる．一方で、 BlueScript はユーザが書く任意のアプリケーションプログラムをコンパイルした比較的小さなネイティブコードを圧縮対象であり、ネイティブコード間の構造上の差異がファームウェア間に比べ大きいため、差分アルゴリズムではなくより汎用的な ZIP や ZStandard などの方が適していることがわかる．

一部の研究では、コードを圧縮する代わりに、コードコンパクションを行うことで、コードサイズを減少し、転送時間を短縮することを目指している．例えば、Cooper と McIntosh の研究 [15] では繰り返しのコードフラグメントを特定し、それらを固定された一つのフラグメントへの制御転送命令に置き換えることで生成するネーティブコードのサイズを削減する手法である．この研究では繰り返されるコードフラグメントの中から、他全てから参照される参照インスタンスを決定する．そしてその参照インスタンスをプロシージャ抽象化することで、ジャンプ命令へと置き換え、全体のコード量を削減する．この時、繰り返し部分はサフィックスツリーによって構築される．この研究によって、従来の最適化済みコードよりも平均 5 %、

最大 15 % のコードサイズ削減が達成できている．

Squeeze[16] は、アリゾナ大学で開発されたリンク後コード最適化ツール alto の一部である．コンパイラはコンパイル対象のソースコードファイル単位で最適化する一方で、コンパイル後のオブジェクトファイル群に対しては最適化を行わない．一方で、 Squeeze はリンク済みの実行可能ファイル単位で追加の最適化を行い、不要なコードを削除することで、コードコンパクションを実行している．Squeeze は Cooper と McIntosh の研究をベースとしており、冗長コードや到達不能コードの削除や、より優れたプロシージャ抽象化を実装することで全体のコード量をより削減している．

これらのコードコンパクションは本論文で使用する可逆圧縮とは異なり解凍を行う必要がないため、それに伴う SRAM 消費が発生しない．しかし、圧縮率に関してはコードコンパクションではあくまで最適化可能な箇所を見つけ出すだけであるので、可逆圧縮よりも小さい．また、冗長なコードブロックを関数呼び出しで書き換えることはオーバヘッドが発生するので、コードコンパクションでの実行速度は必ずしも可逆圧縮よりも優れているとは言えない．さらに、対話性のボトルネックは実行速度ではなく BLE による転送速度であるため、より圧縮率が高い可逆圧縮の方が対話性向上の観点から適していると言える．

第 3 章

提案手法

本研究は第 2 章で説明した BLE の通信時間を短縮するために BlueScript 内で ZIP 圧縮を使用したいが、マイコン上での ZIP 解凍処理によるメモリ消費量が大きすぎるという問題を受けて、 BlueScript 向けの低メモリ消費量で効率的な ZIP 解凍処理を提案する．本提案は次の 3 つの手法からなる．まず、解凍された SRAM 上のネイティブコードを LZ77 辞書として再利用し、既存手法と比べ 32 KB の SRAM 消費量を削減する．次に、最適ハフマン符号化の代わりに、 BlueScript に最適化された静的ハフマン符号化を利用し、既存手法と比べ 8 KB の SRAM 消費量を削減する．最後に、ソフトリセット時に SRAM 内のネイティブコードを全てフラッシュメモリに移動し、次のセッションにおける LZ77 辞書として利用し、SRAM 消費量のオーバヘッドなく圧縮率を向上させる．

3.1 解凍された SRAM 上のネイティブコードを LZ77 辞書として利用

まず最初の提案として、 ZIP 解凍の際に LZ77 デコーダが今まで解凍されて SRAM 上に配置されたネイティブコードを LZ77 辞書として利用することを紹介する．これによって、ZIP における圧縮率を一切損なうことなく、 LZ77 辞書のために確保されていた 32 KB のメモリ領域を削減することができる．その流れを図 3.1 に示す．

この提案手法では、まずホストマシン上でネイティブコードを ZIP 圧縮したデータを BLE でマイコンへと転送する．次にハフマンデコーダが圧縮データを復号し、 LZ77 符号列を LZ77 デコーダへと渡す．そして、 LZ77 デコーダは 32 KB の SRAM の辞書用領域の代わりに、SRAM 上に配置された今までのネイティブコード群を LZ77 辞書として読み込む．その後、 LZ77 デコーダは圧縮データを解凍し、ネイティブコードの配置先にそのまま解凍されたネイティブコードを配置する．最後に、新しく配置されたネイティブコードを実行する．

この提案手法は LZ77 辞書と BlueScript のネイティブコードの配置先におけるデータ構造の共通性を利用したものである．LZ77 辞書においては、復号化時に、データは解凍されるたびに逐次的に辞書の末尾に追加される．これを BlueScript の BLE の高速化のために ZIP 圧縮を利用した場合で考えると、 ZIP における LZ77 デコーダはホストマシンから圧縮データ

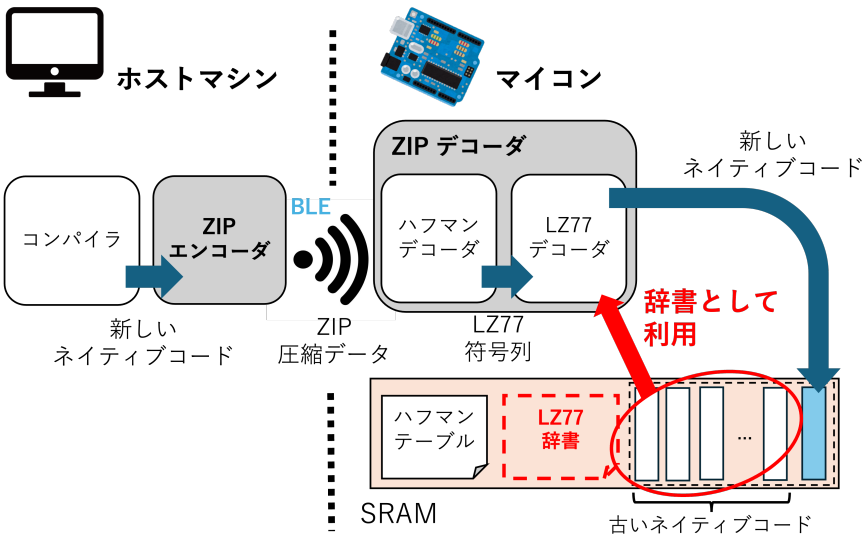

図 3.1. 提案手法 1

が送られるたびに、ネイティブコードへと解凍し、それを逐次的に LZ77 辞書の末尾へと追加する．一方で、 BlueScript のネイティブコードの配置先においては、ホストマシンからネイティブコードが BLE で転送されるたびに、それを逐次的に配置先領域に詰めて配置する．このようにして、 LZ77 辞書の構築の手順とネイティブコードの配置先の構築の手順が等しく、またデータが等しいことがわかる．本提案手法では、この共通性を利用して、 LZ77 デコーダが LZ77 辞書のために専用に確保された 32 KB のメモリ領域ではなく、ネイティブコードの配置先をそのまま利用するようにすることで、解凍のアルゴリズムを一切変更することなく、32 KB のメモリ領域を削減することができる．

また、この手法は ZStandard における Stable Buffer を MINIZ 上に再実装したものである．Stable Buffer は ZStandard のデコーダで利用可能なフラグであり、このフラグが立っていると、 LZ77 デコーダは辞書のためのメモリ領域を確保する代わりに、ZStandard の出力バッファである *ZSTD_outBuffer* を LZ77 辞書としてみなし、そこへ直接解凍データを展開するようにする．ただし、解凍が終了するまでの間、デコーダは常に同じ出力バッファを渡される必要があり、またこの出力バッファは解凍データ全体を保持するのに十分な大きさを持っている必要がある．これは先述した通り、 LZ77 デコーダは連続したメモリ領域である LZ77 辞書上で過去の解凍データを参照し圧縮データを解凍するためである．これゆえに、 Stable Buffer ではメモリ消費が減る代わりに、 ZStandard の利用者が LZ77 辞書となる出力バッファを管理する責任が生じてしまい、 ZStandard はこれをあくまで実験用のパラメータとして用意している．

本提案手法は MINIZ 上に実装されており、これから実装の詳細について説明する．MINIZ では、 ZIP 解凍を開始する関数として、 *mz_inflate* が用意されている．*mz_inflate* の主な目的としては、入力バッファと LZ77 辞書のポインタを ZIP 解凍を実際に解凍を行う

tinfl_decompress という関数に渡し、解凍結果を出力バッファにコピーするということである．MINIZ においては、圧縮・解凍のコンテキストを保持する構造体である *mz_stream* が入力バッファと出力バッファのポインタを保持しており、また ZIP デコーダの状態を保持する構造体である *inflate_state* が LZ77 辞書のメモリ領域 *m_dict* を保持している．本手法における主な変更点としては、1) *inflate_state* 内に *m_dict* が確保される代わりに、 BlueScript のネイティブコードの配置先の先頭ポインタを *section* で保持するようにした；2) *mz_stream* から出力バッファを削除した、ことである．

また、それに伴い MINIZ の解凍関連の関数を書き直した．実装した関数は以下の通りである．

sd_inflateInit
: ZIP デコーダの状態 *inflate_state* を初期化する関数である．新たにネイティブコードの配置先の先頭ポインタを受け取り、それを *section* にセットする．

sd_inflate
: ZIP 解凍を開始する関数である． *m_dict* の代わりに *section* を LZ77 辞書として *tinfl_decompress* を呼び出す．また、 *tinfl_decompress* による解凍結果は LZ77 復号時にすでに *section* に書き込まれているため、出力バッファへのコピーをしない．指定したバイト分、作業バイトバッファから IRAM のデータを読み取る．

sd_inflateRelease
: ZIP デコーダの状態 *inflate_state* を削除する関数である．また、デコーダが初期化されてから解凍したデータサイズの合計値を出力する．

このようにして、 MINIZ 上に ZStandard の Stable Buffer と同様に、任意のメモリ領域を LZ77 領域として利用する機能を実装した．次に、 BlueScript でこの機能をどう利用するかを説明する．BlueScript ではネイティブコードはテキスト領域とデータ領域から構成されている．また、先述した通り ESP32-D0WDQ6-V3 はハーバードアーキテクチャであり、SRAM は命令コードを配置する IRAM とデータを配置する DRAM に分割されている．そこで、 BlueScript はこのアーキテクチャに基づいて、テキスト領域の配置先領域は IRAM 、データ領域の配置先領域は DRAM 上にそれぞれの確保されるようにしている．本実装では、この配置先領域が分離しているという特徴を利用し、テキスト領域とデータ領域をそれぞれ独立して圧縮・解凍するようにしている．

ここで留意すべきは、 LZ77 辞書の読み書きにおけるアクセスと BlueScript のネイティブコードのテキスト領域の配置先である IRAM へのアクセスの単位が異なるという点である．LZ77 辞書は先述した通り、バイト単位で最長一致部分文字列または不一致文字がデコーダの現在位置へと挿入されていく．しかしながら、 IRAM はハーバードアーキテクチャによって、専用の命令バスによって CPU から接続されており、あらゆる読み書きは 4 バイト単位で実行される必要がある．つまり、 IRAM 上のメモリ領域をそのまま LZ77 辞書として利用した場合、 *LoadStoreError* や *LoadStoreAlignmentError* などの実行時エラーが発生してしまう．

しかし、メモリ資源が乏しいマイコンでは、仮に IRAM に命令コードを格納した後

に、未使用の領域が出た場合、それをデータを格納する領域として活用したいという需要が存在する．そこで ESP-IDF では IRAM へのバイトアクセスを可能とする *CONFIG_ESP32_IRAM_AS_8BIT_ACCESSIBLE_MEMORY* というプロジェクト・コンフィグレーションが存在する．ESP-IDF はこのコンフィグレーションが設定されていると、コンパイル時に IRAM 上のアドレスへのアクセスを検出し、それを IRAM をバイトアクセスするため命令コードブロックに置き換えることで、 IRAM 上のアドレスへバイトアクセスしてもエラーが発生することがなくなる．この設定では確かにデコーダは IRAM を LZ77 辞書としてバイト単位で読み書きすることができるようになるが、実行速度が問題となる．

ESP-IDF のウェブサイトによると、このコンフィグレーションを設定した場合、一つの IRAM 上のアドレスすへのアクセスにつき、 167 CPU サイクルがペナルティとして発生する．これは、 LZ77 復号化の際の処理時間のオーバヘッドが発生するだけでなく、ネイティブコードを解凍しロードし終わった後のアプリケーションの実行時間にオーバヘッドが発生することを意味する．これによって、 BlueScript の対話性を損なうだけでなく、 MicroPython などの開発環境に比べアプリケーションの高速な実行が可能である、という大きな性質を損なってしまう．

そこで、本論文では IRAM へのバイトアクセスを可能とする関数を実装した．これにより IRAM 上の LZ77 辞書で LZ77 復号する際のみ呼び出し、4 バイトアラインメントアクセスからバイトアクセスへ命令コードを変換することによる実行速度の悪化を最小限にした．

LZ77 デコーダは復号化する際に、 1) 一致文字列の *memcpy* ； 2) 不一致文字の書き込み、という二種類のアクセスを行う．よって、 *bs_aligned_memcpy* と *bs_aligend_write_byte* という二種類の IRAM 用バイトアクセス関数を定義した．*bs_aligned_memcpy* は IRAM 上のソースアドレスから IRAM 上のディスティネーションアドレスへ指定したバイト分コピーする関数である．*bs_aligend_write_byte* は IRAM 上のアドレスへバイト値を書き込む関数である．これらの関数を実装するために、以下のマクロをまず定義した．

BS_READ_BYTES_ALIGNED
: IRAM 上のアドレスを 4 バイトアライメントで最大 4 バイト読み出す．

BS_FULFILL
: 作業バイトバッファが満杯になるまで IRAM からデータを読み出すマクロであり、内部では BS_READ_BYTES_ALIGNED を呼び出す．

BS_GET_BYTES
: 指定したバイト分、作業バイトバッファから IRAM のデータを読み取る．

BS_WRITE_BYTES_ALIGNED
: IRAM 上のアドレスに 4 バイトアライメントで最大 4 バイト書き出す．

BS_FLUSH
: 作業バイトバッファの内容を IRAM 上のアドレスに書き出すマクロであり、内部では BS_WRITE_BYTES_ALIGNED を呼び出す．

そして、これらのマクロを組み合わせることで *bs_aligned_memcpy* と *bs_aligned_write_byte*

は実装される．*bs_aligned_memcpy* では残りのコピーバイト数が 0 になるまで、コピー元から作業バイトバッファへのコピーを行う BS_FULFILL と、作業バイトバッファからコピー先への書き込みを行う BS_FLUSH を繰り返すことによって、 4 バイトアラインメントの制約を乗り越えている．一方で、 *bs_aligend_wirte_byte* は作業バイトバッファに書き込みたい値をまずセットし、次に作業バイトバッファ上の先頭 1 バイトを指定したアドレスへと書き込む BS_WRITE_BYTES_ALIGNED を利用することによって制約を乗り越えている．

最後にこの一連の処理を BlueScript におけるホストマシンとマイコンの BLE 通信のプロトコルに組み込む．BlueScript では GATT クライアントと GATT サーバをそれぞれに立てて、 GATT パケット内にネイティブコードを転送するためのコマンである BS_CMD_LOAD やネイティブコードの転送が完了したことを示す BS_CMD_JUMP などを含めることで、双方の通信が成り立つ．それに倣い、本手法では以下の三つのコマンドを新しく定義した．ここで注目すべきは BS_CMD_DEC_LOAD である．通常、 GATT のパケットの最大値は 500 B であり、 BS_CMD_DEC_LOAD は最大 500 B 弱のデータしかマイコンに渡すことができない．しかし、不完全なデータでも ZIP デコード関数である *sd_inflate* が呼び出されているのは、 ZIP がストリーミング圧縮に対応しているためである．ZIP デコーダはステートマシンとなっており、データが入力されるたびに状態遷移しながら解凍データを出力していく．入力データが不完全でも、途中の状態さえ保持できれば、また入力が再開されたら処理を続けることができる．このようにすることで、マイコン上で圧縮データ全体を一時保存するためのメモリ領域を確保せずに解凍を行うことができる．

BS_CMD_DEC_START
: ホストマシンが ZIP 圧縮を利用してこれからネイティブコードを転送することを伝えるコマンドであり、*sd_inflateInit* を呼び出す．

BS_CMD_DEC_LOAD
: ホストマシンが ZIP 圧縮データのパケットを転送するためのコマンドであり、ペイロードに圧縮データが含まれている．BlueScript はこれを受け取ると *sd_inflate* を呼び出す．

BS_CMD_DEC_END
: ホストマシンが ZIP 圧縮データの転送が完了したことを伝えるコマンドであり、*sd_inflateRelease* を呼び出す．指定したバイト分、作業バイトバッファから IRAM のデータを読み取る．

3.2　圧縮・解凍時に幾つかの事前作成されたハフマンテーブルを使用

次に二つ目の提案として、 ZIP 圧縮・解凍の際に最適ハフマン符号・復号化する代わりに幾つかの事前作成したハフマンテーブルを使用してハフマン符号・復号化することを紹介する．

それによって最適ハフマンテーブルを圧縮データが転送されるたびにマイコン上で構築する必要がなくなり、また SRAM 上にそのためのワークスペースを確保する必要がなくなり、結果的に SRAM 消費量が減少する．

ZIP で最適ハフマン符号化を利用する場合は、エンコーダは圧縮のたびに入力データをブロックに分割し、ブロックごとに異なるハフマンテーブルを作成してそれをヘッダに含めて圧縮データを出力する．そのような圧縮データが ZIP のデコーダ側に到着すると、まずは圧縮データを圧縮ブロックに分割し、そして新しい圧縮ブロックが読み込まれるたびにヘッダからハフマンテーブル構築に必要な情報を読み込み、新しいハフマンテーブルを作成する．例えば MINIZ では、ハフマンテーブルを構築するために必要な情報としてハフマン符号サイズ表をヘッダから読み出し、そこからハフマン木とルックアップテーブルが構築される．ハフマン木はハフマン符号から LZ77 符号へデコードする際に必要となるハフマンテーブルの実装の一つであり、ルックアップテーブルはデコードをさらに高速化するために使用される補助的なデータ構造である．MINIZ はそれぞれ構造体 *tinfl_decompressor* の一部として *m_code_size* 、 *m_tree* 、 *m_look_up* が定義されており、これらは再構築される際に頻繁にバイトアクセスされるため、合計 8 KB 以上の SRAM のメモリ領域が消費されてしまう．

一方で、ZIP で利用可能なハフマン符号化として、静的ハフマン符号化も存在する．これは新しいブロックが開始しても、 ZIP の仕様書上ですでに定義されているハフマンテーブルを利用するため、エンコーダとデコーダ間で事前にハフマンテーブルの情報を共有し、エンコーダからデコーダへ送る必要がない．これによって、デコーダは圧縮データを受け取っても、ヘッダからハフマン符号サイズ表を読み出す必要がなく、またハフマンテーブルの再構築を行う必要がないため、 SRAM の代わりに頻繁に書き込みができないフラッシュメモリ上に配置することができる．しかし、 MINIZ の実装上の問題として、圧縮ブロックの解凍を開始した際に、静的ハフマンテーブルをハフマン符号サイズ表からハフマン木とルックアップテーブルを再構築しており、そのまま利用する場合には最適ハフマンブロックの解凍と同じ SRAM の消費量が発生してしまう．これは MINIZ 内における最適ハフマンブロックと静的ハフマンブロックの解凍における処理を共有するためだと考えられるが、本研究の背景であるマイコン上での解凍を考慮すると、実装上の SRAM 消費に関するオーバヘッドが大きいと言える．

さらに、 ZIP の静的ハフマン符号化は、最適ハフマン符号化に比べ圧縮率が低いという懸念点も存在する．最適ハフマン符号化では入力データ内における文字の出現頻度などの統計データを計算し、出現頻度が高い文字に短い符号、出現頻度が低い文字に長い符号、というように効果的に符号を割り当てることで高い圧縮率を達成することができている．しかし、静的ハフマン符号化ではあらかじめ仕様書で決められた文字に対する決め打ち的な出現頻度表を使用して各文字に符号を割り当てている．これは入力データにおける各文字の出現頻度を無視しており、最適ハフマン符号化よりも低い圧縮率となってしまうことは明白である．そもそも、ZIP において静的ハフマン符号化が利用されているのは、最適ハフマン符号化ではうまく圧縮できないデータを代わりに圧縮するためである．しかし、そういった入力データは数十 KB とデータサイズが小さい場合が多く、圧縮・解凍のオーバヘッドを考慮して無圧縮となる場合もあり、また一般的な BlueScript のネイティブコードよりも小さいため、本研究の背景では

利用されることは想定されない.

そこで、本研究では BlueScript がコンパイルしたネイティブコードからいくつかの静的ハフマンテーブルを事前作成し、マイコン上フラッシュメモリ上にそれらを配置することで、既存の静的ハフマン符号化よりも最適ハフマン符号化からの圧縮率の低下を抑えつつ、 SRAM の消費量を大幅に削減する方法を提案する．ハフマンテーブルを事前作成することで、静的ハフマン符号化を行えるため、ハフマンテーブルをフラッシュメモリ上に配置することでメモリ消費量を削減でき、また本研究背景における ZIP の入力データは BlueScript がコンパイルしたネイティブコードであるため、ネイティブコード間の構造上の類似性を利用できると考えた．先述した通り、 BlueScript は BlueScript プログラムを C 言語へトランスパイルしてからコンパイルするため、それぞれの処理で共通のパターンが出力され、ネイティブコード間でより高い類似性を有していると考えられる．本提案手法の処理の流れを 3.2 に示す.

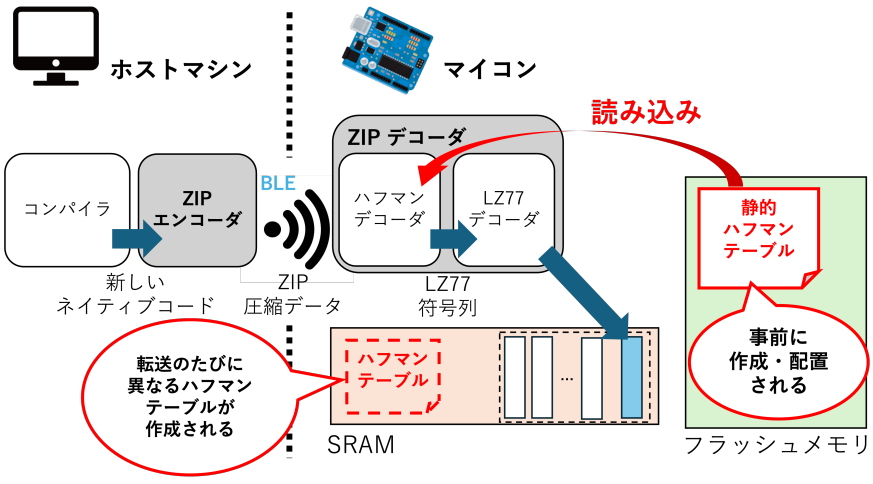

図 3.2. 提案手法 2：圧縮・解凍時に幾つかの事前作成されたハフマンテーブルを使用

まず BlueScript からネイティブコードが生成されると、 ZIP エンコーダへと入力されるが、ZIP エンコーダはホストマシンに保存されている事前作成された静的ハフマンテーブルを読み込み、それを使用してネイティブコードを圧縮する．次に、圧縮データが BLE でマイコンへと転送され、 ZIP デコーダへと入力される．そこで ZIP デコーダは SRAM 上で最適ハフマンテーブルを構築する代わりに、フラッシュメモリ内の静的ハフマンテーブルを読み出し、それを使用してハフマン符号列を LZ77 符号列へと復号する．最後に、提案手法 1 で述べたように LZ77 デコーダが復号処理を行うことで、ネイティブコードが配置先にそのままロードさている状態となり、実行したのちにホストマシンへ実行結果を返す.

次に、本手法における実装を説明する．まず、静的ハフマンテーブルを作成する関数として *bs_create_static_compression_table* と *bs_create_static_decompression_table* を用意している．それぞれ ZIP エンコーダ向けと ZIP デコーダ向けに静的ハフマンテーブルを作成する関数であるが、このようにエンコーダとデコーダにわざわざ分割するのは、 MINIZ ではそれぞ

れにおいて利用されるハフマンテーブルのフォーマットが異なるためである．

MINIZ の ZIP エンコーダでは文字と符号の変換表がハフマンテーブルとして使用される．ハフマン符号化の際に、入力データはバイト単位で文字に分割され、各文字を変換表を通してハフマン符号へと一対一で変換されていくことで圧縮されていく．一方で、MINIZ の ZIP デコーダでは先述した通りハフマン木とルックアップテーブルがハフマンテーブルとして使用される．このようにエンコーダとデコーダが異なるデータ構造を使用して符号化・復号化を行うのは、復号の際に入力されるハフマン符号が可変長であるためである．デコーダは入力される圧縮データをビット単位でハフマン木の根から枝を辿っていき、最終的に葉に到達することで文字へと変換する．しかし、デコードのたびにこのような木の探索を行なっていくと、 ZIP 解凍処理の処理時間に大きなオーバヘッドが発生してしまう．そこで、ルックアップテーブルが処理時間の短縮に利用されている．ルックアップテーブルの各インデックスは入力データの先頭のビット列に対応しており、エントリを参照することで復号先の文字と符号長のペアが出力される．具体的にはルックアップテーブルの各インデックスにおける Least Significant Bit から Most Significant Bit へビット単位で読んでいった時に現れる符号の、符号長と出力先文字が 7 ビットと 9 ビット結合されている状態でそのエントリに格納されている．しかし、エントリ数は 1024 と 10 ビット分の入力データまでしかデコードすることができなく、 ZIP のハフマン符号化は最大 16 ビットまでの符号を出力するため、全ての符号がルックアップテーブルでデコードされるわけではないが、ハフマン符号化では短い符号の出現頻度が高いという性質を考慮すると、十分な高い性能が達成できることがわかる．このようにして ZIP デコーダではエンコーダでは使用されないデータ構造を持つため、それぞれ異なる静的ハフマンテーブルを作成する必要がある．

bs_create_static_compression_table においては 1) 圧縮するデータへの LZ77 符号化、 2) 生成された LZ77 符号列から文字の頻度表の作成、 3) 頻度表からハフマンテーブルの作成、と処理していくことで圧縮時に使用する静的ハフマンテーブルを生成することができる．まず、圧縮するデータに対する LZ77 符号化は MINIZ の関数をそのまま利用している．次に、頻度表を作成するための関数として *bs_create_count_up_table* を実装した．MINIZ では頻度表の作成の際に、ブロックの上限である 32 KB 分の圧縮データが LZ77 符号化されると自動的に頻度表のカウントを停止し、 LZ77 符号化が再開されると今までの頻度表の情報を全て初期化してしまう．これは最適ハフマン符号化において、データの空間局所性を考慮して 32 KB という範囲で LZ77 符号化が自動的に切り上げられてしまい、またハフマン符号化は LZ77 符号化に付随するため頻度表を頻繁に初期化する必要があった．しかし、本提案手法では静的ハフマンテーブルは特定のブロックに紐付いていないため、頻度表を頻繁に更新する必要がない．そこで、本提案手法では頻度表のいずれかのエントリが *uint16_t* の最大値である 65535 に到達するまで各文字の頻度をカウントし続けるようにしている．

このようにしてして頻度表が作成されたのちに、頻度表の各エントリ内の値をインクリメントする．これは頻度表は LZ77 符号列における各文字の出現頻度を記録しており、出現しない文字のエントリは値が 0 となってしまうためである．最適ハフマン符号化では、出現しない文字には符号を割り当てないことで、符号の割り当て数を節約し、平均的な符号長を短くす

る．一方で、本手法ではハフマンテーブルは汎用的に使用され、作成時に登場しない文字も符号化する可能性があるため、全ての文字に符号を割り当てる必要がある．そこで、本手法では *bs_create_count_up_table* によって作成された頻度表の各エントリに 1 を足すことで、各文字の出現頻度に大きな変化をもたらすことなく、全ての文字に符号を割り当てることがきる．最後に頻度表を *tdefl_optimize_huffmam_table* へ渡すことで、ハフマン符号サイズ表とハフマン符号表を生成する．ここで留意すべきは ZIP においては不一致文字・一致長と一致距離という二つのグループが独立してハフマン符号化が行われることであり、頻度表から二つのグループに分け、その関数を実行している．そして、ハフマン符号サイズ表とハフマン符号表をファイル出力することで圧縮用の静的ハフマンテーブルを作成することができる．

bs_create_static_decompression_table は 1) ハフマン符号サイズ表の読み込み、 2) 文字へのハフマン符号の割り当て、 3) ルックアップテーブルの構築、 4) ハフマン木の構築、と処理を分けることができる．*bs_create_static_decompression_table* ではまず、圧縮用の静的ハフマンテーブルを作成した際に得られるハフマン符号サイズ表をロードする．次に、 *next_code* というデータ構造を初期化するが、これはある符号長における次に割り当てる符号を保持する配列であり、ハフマン符号長の上限である 16 個のエントリを持っている．そして、文字を昇順にハフマン符号サイズ表を通して符号長へと変換し、符号長を *next_code* を通してその文字に割り当てる符号へと変換する．このときに参照された *next_code* のエントリの値をインクリメントし、次にその符号長に割り当てる符号を用意する．その後、その符号の長さが 10 ビット以上なら、ルックアップテーブルのエントリを作成せず、以下なら、符号が LSB にくるように複数のインデックスを設定し、符号長とデコード先の文字を結合した値をそれらのエントリに書き込む．最後に、各符号のビット列に沿ってハフマン木の枝と葉を作成していくことで、解凍時に使用する静的ハフマン復号化関連のデータ構造が *tinfl_decompressor* 内に作成される．

しかし、 *tinfl_decompressor* にはアクセスパターンの異なる二種類のメンバ変数群を持っており、それぞれを異なるメモリ上に配置する必要がある．先述した通り、*m_code_size*・*m_tree*・*m_look_up* は静的ハフマン復号化においては読み取り専用でアクセスできる一方で、*m_bit_buf* 、 *m_state* 、 *m_raw_header* などはブロックが更新されたり、デコード処理がループしたりする場合に新しい値を書き込む必要がある．そこで、本手法は *tinfl_decompressor* を SRAM に配置する *bs_tinfl_decompressor* と、フラッシュメモリに配置する *bs_tinfl_decompressor_flash* の二つに分割し、それぞれデコーダの状態関連の 84 B の構造体と、静的ハフマンテーブル関連の約 8 KB の構造体となっている．最後に、後者を解凍用の静的ハフマンテーブルとしてファイル出力することで処理が完了する．

ファイル出力された解凍用静的ハフマンテーブルはマイコン上のフラッシュメモリに書き込む必要があるが、 ESP-D0WDQ6-V3 においては以下のコマンドが用意されている．このコマンドのアドレス部にフラッシュメモリ上の書き込み先のアドレスを設定し、ファイル名に静的ハフマンテーブルのものを設定すれば良い．

```
esptool.py --chip <chip> --port <port> --baud <speed>
                write_flash <address> <file>
```

次に圧縮･解凍時に静的ハフマンテーブルを読み出す関数として *bs_load_static_compression_table* と *bs_load_static_decompression_table* を定義した．*bs_load_static_compression_table* はホストマシン上において LZ77 符号列をハフマン符号化する前に呼び出す関数であり、先述したファイルから圧縮用静的ハフマンテーブルを読み出す．同様に、*bs_load_static_decompression_table* はマイコン上においてハフマン符号列を復号化する前に呼び出す関数であり、*bs_mmap_huff_ptr* を呼び出しフラッシュメモリからハフマンテーブルをアドレス空間上にマッピングしてから、フラッシュメモリから解凍用静的ハフマンテーブルを読み出す．このフラッシュメモリ上の物理アドレスをアドレス空間上の仮想アドレスにマッピングする機構として、*esp_partitioin* という ESP-IDF 内のライブラリの *esp_parition_mmap* を使用した．このライブラリではフラッシュメモリをパーティションと呼ばれる単位で分割・管理され、本手法ではフラッシュメモリに新しく静的ハフマンテーブルを保存するための *huff* というパーティションを追加している．

このようにして、 ZIP エンコーダ・デコーダは静的ハフマンテーブルを事前作成し、それらを読み込むことができるになったが、デコーダは送られてくる圧縮データが最適ハフマンブロックなのか、既存の静的ハフマンブロックなのか、本研究での静的ハフマンブロックなのか見分けがつかない．そこで、本研究は ZIP の reserved と呼ばれるブロックタイプ番号を利用することとした．ZIP にはブロックの種類を識別するために、ブロックヘッダにブロックタイプ番号が含まれている．一般的には、無圧縮のブロックは 0 、静的ハフマンブロックは 1 、最適ハフマンブロックは 2 、 reserved ブロックは 3 となっている．この reserved ブロックは現在は使用されておらず、 MINIZ では reserved ブロックのタイプ番号である 3 がヘッダから読み出されるとエラーが発生するようになっている．本手法によって圧縮されたブロックを新たに STATIC ブロックと命名しタイプ番号 3 を割り当てると同時に、最適ブロックと既存の静的ブロックをそれぞれ DYNAMIC ブロックと FIXED ブロックと命名し直すことで、区分している．

以上のようにして、事前作成された静的ハフマンテーブルを使用して ZIP 圧縮・解凍し、メモリ消費量を削減できるようになったが、圧縮率は最適ハフマン符号化から低下している．そこで、圧縮率を向上するために、単一の静的ハフマンテーブルで圧縮・解凍するのではなく、複数のテーブルを利用することにした．図 3.3 は第 4 章で紹介するベンチマークをコンパイルした BlueScript のネイティブコードから生成された静的ハフマンテーブルのうち、 lzss と crc というベンチマークのものを使用して各ネイティブコードを ZIP 圧縮した際の圧縮率の棒グラフである．この図において、青色の棒は lzss 、橙色の棒は crc の圧縮率を表している．ここから、同じ静的ハフマンテーブルでも入力ネイティブコードによって圧縮率が大きく変化し、また同じ入力ネイティブコードでも静的ハフマンテーブルによって圧縮率が大きく変化するということである．例えば、 lzss から生成したテーブルにおいて、 fft というネイティブコードを圧縮した際の圧縮率は 7.76 であり、queens の場合は 31.41 である．一方で crc から生成したテーブルにおいては、 fft では圧縮率は 13.54 であり、 queens では 21.37 である．さらに、 crc のネイティブコードを圧縮するに至っては、 lzss は 0.15 であり、 crc は 18.00 ととても大きな差が存在している．よって、幅広いネイティブコードに対して安定して高い圧

縮率を維持するために、複数の静的ハフマンテーブルを利用して、圧縮時に最も高い圧縮率のものを選ぶように拡張した.

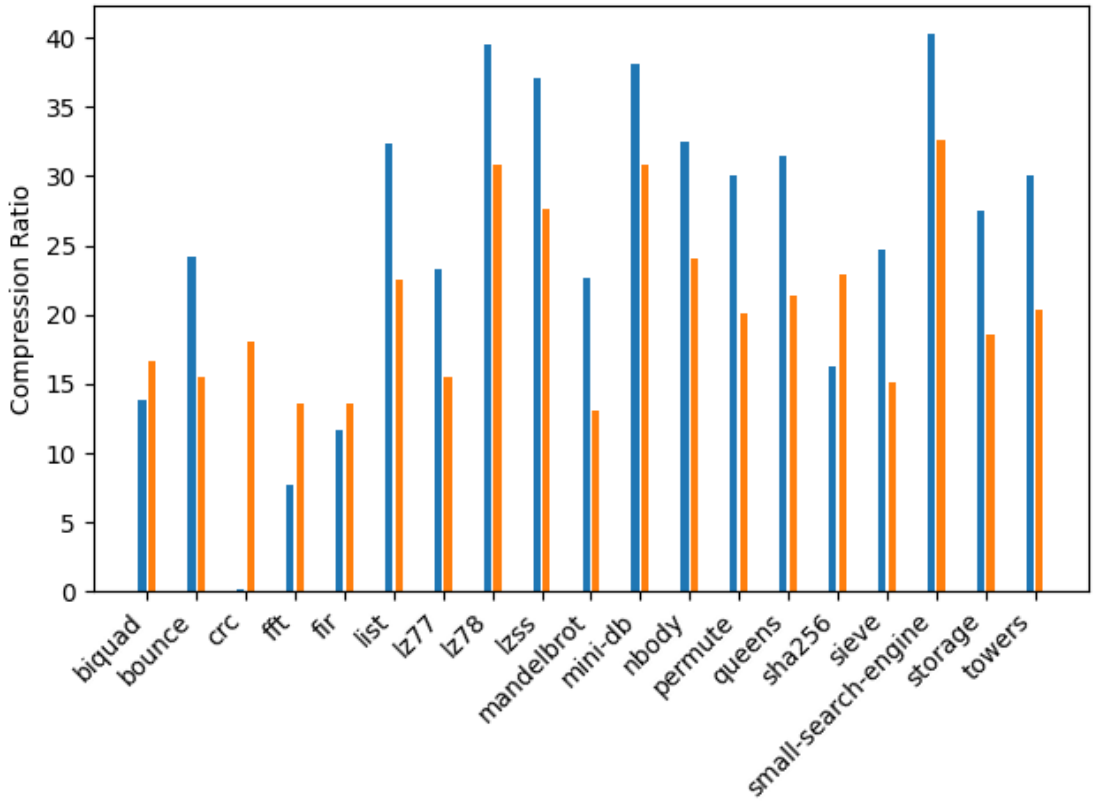

図 3.3. lzss (青色) と crc (橙色) から生成した静的ハフマンテーブルによる圧縮率の違い

エンコーダは最適な静的ハフマンテーブルを選択するために、静的ハフマン符号化が開始する前に、候補である複数の静的ハフマンテーブルを、 *bs_get_huff_result_length* という関数に渡す. この関数は、渡された静的ハフマンテーブルをファイル内のハフマン符号長表をロードし、 LZ77 符号列における各文字から符号の長さに変換してその合計を出力するものである. エンコーダはその結果の中から ZIP 圧縮データが最短となる静的ハフマンテーブルを選び、ハフマン符号化を行っていく. この一連の処理は単に LZ77 符号列の各文字を一回変換するだけでよく、候補のテーブル数と LZ77 符号列の長さの積に比例した実行時間のオーバヘッドで完了する.

実際には、エンコーダは符号化を開始する前にデコーダとテーブルの情報を共有するために、ブロックヘッダに候補のテーブル群をリストとして際のテーブルの番号を含める. デコーダはエンコーダと同様の静的ハフマンテーブルのリストをフラッシュメモリ上に有しており、ブロックの解凍が開始すると、テーブルの番号を読み出し、それを拡張された *bs_load_static_decompression_table* へと渡すことで適切な静的ハフマンテーブルが利用可能な状態となる. この関数はフラッシュメモリの huff パーティションを

bs_tinfl_decompressor_flash の配列として仮想アドレスにマッピングし、テーブル番号を辿るように拡張されている．

最後に、 BlueScript のネイティブコードをテキスト領域とデータ領域に分割してそれぞれ独立した静的ハフマンテーブルを作成することとした．提案手法 1 の実装を開設する際に、BlueScript はネイティブコードのテキスト領域とデータ領域をそれぞれテキスト領域とデータ領域はそれぞれ命令コードとデータというふうに、パターンに差異が存在する．LZ77 符号化においては過去に出現したパターンを参照するため、そのようにパターンの異なるデータが混在していても探索さえされれば圧縮率に大きな変化はないとわかる．しかし、ハフマン符号化は各文字の出現頻度に左右され、またテキスト領域とデータ領域においては文字の出現頻度が大きく異なることが予想できるので、分割した方が圧縮率が良くなると予測できる．

3.3 ソフトリセット時に古いコードをフラッシュにコピーし、LZ77 辞書として利用

最後に三つ目の提案として、ソフトリセットする際に今まで SRAM 上に配置されてきたネイティブコードをフラッシュメモリ上に移動させ、 LZ77 辞書として利用する方法を紹介する．これによって、追加の SRAM 消費量が発生することなく、高い圧縮効率を達成することができるようになる．

LZ77 符号化は先述通り、入力データにおいて繰り返されるパターンを検出し、それを参照と呼ばれるコンパクトな表現によって圧縮する．LZ77 エンコーダは入力データを処理した順に辞書に登録していき、現在の処理位置から始まる文字列との最長一致文字列を辞書から探索することで繰り返しを見つけ出す．つまり、いかに辞書にこれから符号化する入力データと同じパターンが存在しているかによって LZ77 符号化は圧縮性能が異なる．BlueScript のネイティブコードは先述したように、 C 言語へのトランスパイルとネイティブコードへのコンパイルを経て生成されるため、ネイティブコード間における構造の類似性は高い．そこで BlueScript のネイティブコードを追加で辞書に含めることで、より多くの有効的なパターンが存在し、圧縮率を向上することができると考えた．

しかし、単にネイティブコードを辞書に含めれば良いというわけではない．図 3.3 でもあるように、ネイティブコードはどのようなアプリケーションからコンパイルされたかによって出現するパターンが異なる．特に、 crc のように数値計算をするアプリケーションでは当然データの割合が増加するため、比較的命令コードの割合が高い lzss とは異なるデータパターンを持つ．そこで、本手法では BlueScript がソフトリセットされた際に、今までの SRAM 上で配置されてきたネイティブコードを LZ77 として利用することとした．それは、ノートブックライクな開発環境において、ユーザはソフトリセットした前後でも似たようなプログラムを書く可能性が高いと考えたためである．第 2 章で紹介したように、マイコン開発者はデバッグや機能の修正のために頻繁にファームウェアを更新するため、 OTAP というマイコンにおける効率的な転送の需要が高まってきている．つまり、 BlueScript ユーザはマイコン上のアプリ

ケーションのデバッグや機能の修正などのため、ソフトリセット前後のプログラムの差異は小さいといえる．結果的に、ソフトリセット前後のネイティブコードの構造上の類似性は高く、辞書に含めることで効率的に圧縮することができる．

この際に追加される辞書は、デコーダから参照のみされるため、書き込む必要がない．よって、フラッシュメモリ上で保持することができ、結果的に SRAM を一切消費することなく圧縮率を大幅に向上できる．また、 BlueScript においてはホストマシンはシャドウメモリと呼ばれるマイコン上のメモリ空間の写しを持っている．よって、ソフトリセット時にホストマシンはその時点でのシャドウメモリをファイル出力し、マイコンはネイティブコードをフラッシュメモリに移動することで追加の辞書を得ることができる．本提案手法の処理の流れを図 3.4 に示す．

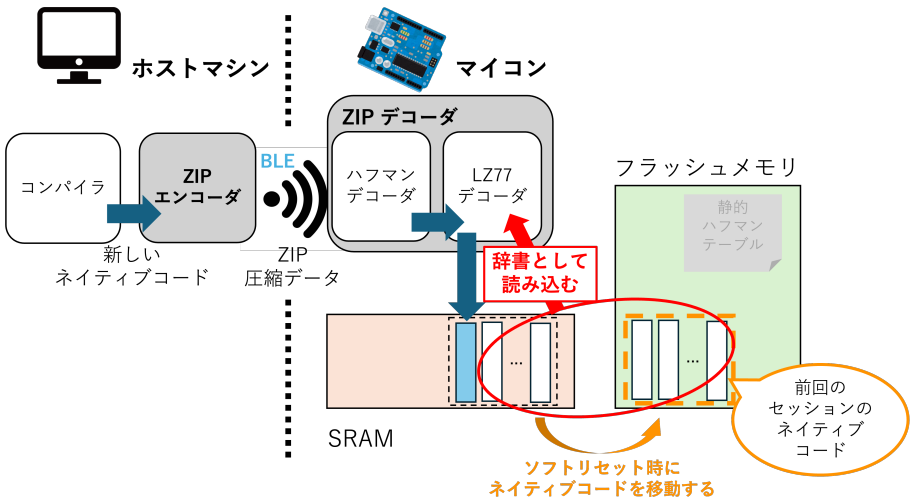

図 3.4. 提案手法 3

BlueScript ユーザがソフトリセットを実行すると、ホストマシンにおいてはシャドウメモリから今までのネイティブコードをファイルに出力し、マイコンにおいては SRAM 上のネイティブコードをフラッシュメモリに移動する．ソフトリセット後に BlueScript ユーザがプログラムを実行すると、ネイティブコードへとコンパイルされ、 ZIP エンコーダに渡される．ZIP エンコーダは前回のセッションのネイティブコードを LZ77 辞書に挿入してから LZ77 エンコーダを呼び出し、 LZ77 符号化したのちにハフマン符号化を行い、 ZIP 圧縮データを生成する．生成された ZIP 圧縮データは BLE によってマイコンへと転送され、ハフマンデコーダによって LZ77 符号列へと復号され、 LZ77 デコーダへと渡される．LZ77 デコーダは適切に SRAM または フラッシュメモリへの参照を切り替えながら、復号することで、ネイティブコードへと解凍することができる．

次にソフトリセット時にシャドウメモリからネイティブコードをファイルへ退避する際の流れについて説明する．BlueScript は図 2.1 からわかるように緑色のリセットボタンがあり、それが押されるとソフトリセットが実行される．ソフトリセットの際、 IDE はまずコンパイ

ラサーバとマイコンへそれぞれソフトリセットコマンドを送る．コンパイラサーバがソフトリセットコマンドを受け取ると、シャドウメモリにおけるテキスト領域とデータ領域のネイティブコードをこの順で結合し、ローカル上のファイルに出力する．一方で、マイコンでは *bs_copy_code_to_flash* が実行される．この関数は IRAM 上に配置されたテキスト領域の合計サイズと DRAM 上に配置されたデータ領域の合計サイズの和から全ネイティブコードの合計サイズを計算し、ネイティブコードのサイズ、テキスト領域、データ領域の順に指定されたフラッシュメモリ上のアドレスへと書き込まれる．実装の簡潔化のために、フラッシュメモリ上に dict と呼ばれるパーティションを定義し、 *esp_partition_write* を使用してパーティションの API で辞書を構築する．ここで留意しなければならないのはデータを書き込む前に *esp_partition_erase_range* を呼び出しフラッシュメモリのパーティションを初期化する必要がある点である．フラッシュメモリのデータはビット単位で保持されており、値を 1 から 0 へと書き込みによって変更することができるが、 0 から 1 へとは書き込みによって変更することができないという性質を持っている．そのため、フラッシュメモリ上で再度データを書き込む際には、一度フラッシュメモリをブロック単位で電圧を加え、全てのデータを 1 にする初期化処理である **ERASE** を行う必要がある．

こうしてホストマシンとマイコンの両方においてネイティブコードを適切に退避することができる．次に実際の圧縮・解凍処理における実装について説明する．エンコーダでは ZIP 圧縮を開始する前には *bs_update_dict* という関数を呼び出す．この関数は、ローカル上に前回のセッションのネイティブコードを保存したファイルが存在するかどうかを確認し、もし存在すればそれを辞書に含め、存在しなければそのまま戻る関数である．*bs_update_dict* がコードを辞書に含める際、先述した *m_dict* にコピーすると同時に、ハッシュテーブルを構築しなければならない．ZIP 圧縮において、ハッシュテーブルは LZ77 符号化の高速化に使用される．ハッシュテーブルをしない場合であると、 LZ77 エンコーダは毎回辞書と現在の処理位置で一文字ずつ比較する必要があり、辞書のサイズに比例して処理時間が増加してしまう．そこで、辞書にバイトデータをロードするたびに辞書の末尾から先頭に向かう 3 バイトをハッシュ関数に渡しハッシュ値を計算し、それをハッシュテーブルのインデックスとし、またバイトデータを新しくロードした辞書の位置をハッシュテーブルの値とする．このハッシュテーブルを文字列探索時に使用することで、最初から 3 バイトマッチしている辞書上の候補を参照することができ、エンコーダはその中から一致文字列の長さが最大のものを選び LZ77 符号化する．一般的に、辞書内において 3 バイトマッチしている部分文字列を探索する時間は全探索に比べ圧倒的に速いため、多くの ZIP の実装で採用されている．

このようにして辞書とハッシュテーブルを構築した後、 LZ77 エンコーダは通常通りに符号化を行う．この時に構築された辞書上の領域を本論文では便宜上 Extra Dictionary (ED) と呼ぶ．LZ77 符号化の際にもし ED 内から参照を探索してきたら、その参照における一致距離は本来の辞書サイズ以上のオフセットを示すこととなる、という性質がある．このようにして圧縮されたデータは BLE によってマイコンへと転送され、通常通りにハフマンデコーダによって LZ77 符号列へと復号される．しかし、そのままでは一致距離が大きいため LZ77 デコーダは SRAM 上の LZ77 辞書外のメモリを参照してしまう．

そのため、本研究では 3 種類の参照を定義し、それぞれにおいて適切な処理を実装した．それぞれ 1) SRAM のみを参照する場合、2) フラッシュメモリのみを参照する場合、3) SRAM とフラッシュメモリの両方を参照する場合、となっている．まず、一致距離における SRAM 上の辞書とフラッシュメモリ上の辞書の境界を、 SRAM 上の辞書サイズとした．一致距離がこの境界よりも小さい場合は 1) 、この境界よりも大きく、さらに一致文字列が境界を跨がない場合は 2) 、跨ぐ場合は 3) となる．1) と 2) においては参照先をそれぞれのメモリデバイスがマッピングされた仮想アドレスにセットすることで、今まで通りのデコード処理復号できる．一方で、 3) は参照先を二つに分割し、それぞれのメモリデバイスの仮想アドレスから参照をコピーしてくることで、 LZ77 復号処理が行われる．

このように圧縮・解凍されることで、 SRAM のメモリ消費に関するオーバヘッドが発生することなく、圧縮率を向上することがでいる．

第 4 章

実験

4.1 使用するベンチマークに関する説明

本研究における提案手法を計測・評価する前に、まず計測するために使用したベンチマークについて説明する．まず、ランダムに選んだ lz77[*1]、 lz78[*2]、 lzss[*3]、 mini-db[*4]、 small-search-engine[*5] の 5 つの Github 上で公開されているリポジトリを BlueScript に書き換えたものをベンチマーク・グループ A として利用する．グループ A の各ベンチマークの名前、説明、コンパイル後に生成されるネイティブコードサイズの情報を表 4.1 にまとめている．これらのベンチマークはサイズが比較的大きく、 BlueScript ユーザによって開発された大規模なアプリケーションやライブラリに対する本提案手法の性能を考察するのに適している．

表 4.1. ベンチマーク・グループ A

ベンチマーク名	説明	ネイティブコードサイズ [B]
lz77	LZ77 符号化・復号化の実装	1431
lz78	LZ78 符号化・復号化の実装	3738
lzss	LZSS 符号化・復号化の実装	9692
mini-db	単純なデータベースエンジンの実装	6546
small-search-engine	生徒情報の管理用システムの実装	4868

次に、 *are we fast yet?* [17] におけるマイクロベンチマークを BlueScript に書き換えたものをベンチマーク・グループ B として利用する．ベンチマーク名とネイティブコードサイズを表 4.2 の左側にまとめている．これらのベンチマークはサイズが比較的小さく、BlueScript ユーザによって開発された小規模アプリケーションや第 2 章で述べたようなコードフラグメ

[*1] https://github.com/yourtion/LearningMasteringAlgorithms-C/blob/master/source/lz77.c
[*2] https://github.com/SteCicero/lz78
[*3] https://github.com/MichaelDipperstein/lzss
[*4] https://github.com/anishbasu/mini-db
[*5] https://github.com/lord-charite/Small-Search-Engine

ントに対する本提案手法の性能を考察するのに適している．最後に、C や MicroPython などのような ESP32 マイコンで利用可能なプログラミング言語の実行速度を測るために開発されたベンチマーク [18] を BlueScript に書き換えたものをベンチマーク・グループ C として利用し、同様に表 4.2 の右側に詳細をまとめている．これらのベンチマークは CRC チェックサムや高速フーリエ変換などの数値計算が中心のベンチマークであり、プログラムの中に計算用のデータが多く含まれている．よってデータを多く含むアプリケーションに対する本提案手法の性能を考察するのに適している．

表 4.2. ベンチマークグループ B (左) と ベンチマークグループ C (右)

ベンチマーク名	ネイティブコード サイズ [B]
bounce	1118
list	1106
mandelbrot	753
nbody	2490
permute	557
queens	717
sieve	433
storage	750
towers	1180

ベンチマーク名	ネイティブコード サイズ [B]
crc	5813
sha	9120
fft	17228
fir	25385
iir	22124

そして実験 4.5 と実験 4.6 でソフトリセット後に BlueScript プログラムに少し変化を加えるために、各ベンチマークにソースコード 4.1 の関数を挿入する．これによって、ソフリセット後のネイティブコードがソフトリセット前から、約 160 バイトほど増加する．

Listing 4.1. 実験 4.5 と 4.6 で使用する関数

```
function foo() {
    const arr = new Array<integer>(5);
    for (let i = 0; i < arr.length; i++)
        arr[i] = i * 2;
    let sum: integer = 0;
    for (let i = 0; i < arr.length; i++)
        sum += arr[i];
    let max: integer = 0;
    for (let i = 0; i < arr.length; i++)
        if (arr[i] > max)
            max = arr[i];
}
```

4.2 提案手法と既存手法と ZStandard における解凍時の SRAM 消費量の比較

提案手法と既存手法 (MINIZ) と、ZStandard における解凍時の SRAM 消費量とその内訳を計測し、図 4.1 に示す．提案手法と既存手法では *inflate_state* と *tinfl_decompressor* のサイズの合計、ZStandard では *ZSTD_DCtx* のサイズを解凍時の SRAM 消費量全体としている．また、既存手法では *inflate_state* の *m_dict* のサイズを LZ77 辞書、*tinfl_decompressor* の *m_look_up* 等のサイズの合計をハフマンテーブルによる SRAM 消費量としている．ZStandard では *ZSTD_DCtx* の *workspace* のサイズを LZ77 辞書、*litExtraBuffer* のサイズをハフマンテーブルによる SRAM 消費量としている．そして、全体から LZ77 辞書とハフマンテーブルのサイズを引いた残りをその他としている．

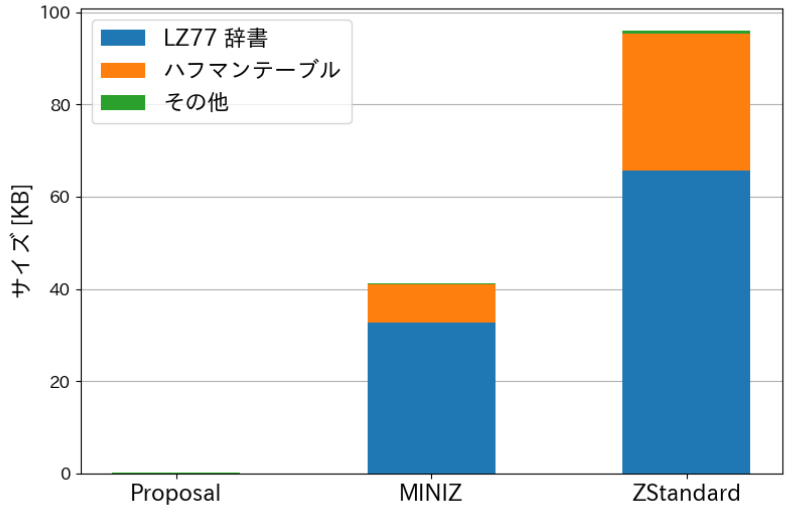

図 4.1. 提案手法 (左)、MINIZ(中)、ZStandard(右) における解凍時の SRAM 消費量とその内訳

図 4.1 から、既存手法と ZStandard による SRAM 消費量のほとんどは LZ77 辞書とハフマンテーブルによるものであることがわかる．提案手法の LZ77 辞書とハフマンテーブルは SRAM を消費しないため、既存手法から SRAM 消費量を 99.70 % 削減可能である．また、ZStandard の消費量と比較した際、提案手法は 99.87 % 低くなっている．

これにより、本提案手法は SRAM 消費量の大きい既存研究とは異なり、たった 124 バイトのオーバヘッドだけで ZIP 圧縮・解凍が利用可能であることがわかる．

4.3　MINIZ と ZStandard における性能の比較

提案手法と既存手法を性能比較する前に、既存手法が BLE 転送を高速化し、 BlueScript の対話性を向上させるのに有効であることを示す．それにあたり各ベンチマークのネイティブコードを 1) 無圧縮で送った場合、2) MINIZ + 最適ハフマン符号化で圧縮して送った場合 (既存手法)、3) MINIZ + 静的ハフマン符号化で圧縮して送った場合、4) ZStandard で圧縮して送った場合における圧縮率と BlueScript ユーザの待機時間を測定した．圧縮率は無圧縮の場合以外で計測し、図 4.2 に載せている．BlueScript ユーザの待機時間は実行ボタンが押されてから、マイコンからホストマシンへネイティブコードのロードが完了したことを伝えるメッセージが届くまでの時間としており、これによって対話性を測る．図 4.3 に各ベンチマークをそれぞれ実行した際の待機時間を 5 回測定し、それらの平均を載せている．

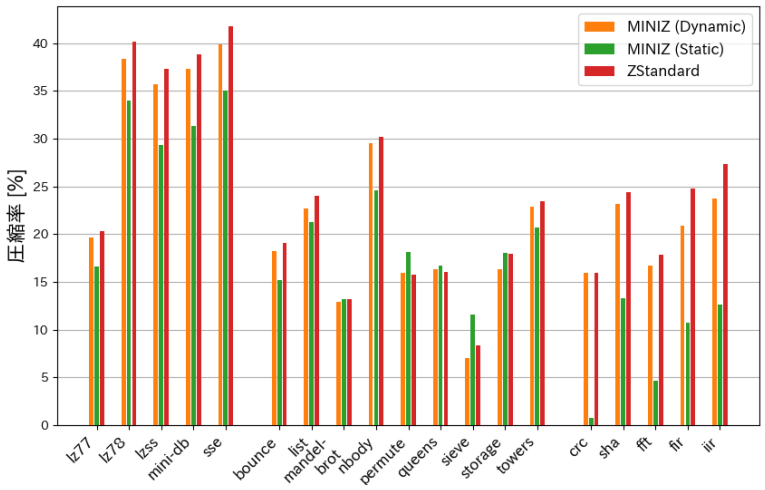

図 4.2. MINIZ と ZStandard における圧縮率の比較

まず、ベンチマーク全体における圧縮率の平均は、 MINIZ + 最適ハフマン符号化で 22.81 %、 + 静的ハフマン符号化で 18.31 %、ZStandard で 24.03 % となっている．これによって、ZStandard は MINIZ よりも、また最適ハフマン符号化は静的ハフマン符号化よりも圧縮率が優れていることがわかる．一方で、無圧縮の場合の待機時間からの減少率の平均はそれぞれ 14.37 %、4.86 %、9.19 % となっている．ZStandard は既存手法よりも圧縮率が高いにも関わらず待機時間が伸びているのは、解凍時間が伸びているためであると考えられる．また、静的ハフマン符号化は圧縮率・待機時間の減少率が共に性能が最も低いことがわかる．

ベンチマーク・グループ別で見ると、グループ A における圧縮率の平均はそれぞれ 34.21 %、29.28 %、35.67 % となっており、待機時間の減少率の平均はそれぞれ 29.05 %、23.06 %、28.50 % となっている．全ての場合で全体平均よりも良い性能が出たことがわかるが、これはグループ A のネイティブコードが比較的に圧縮しやすい命令コードを多く含むためであ

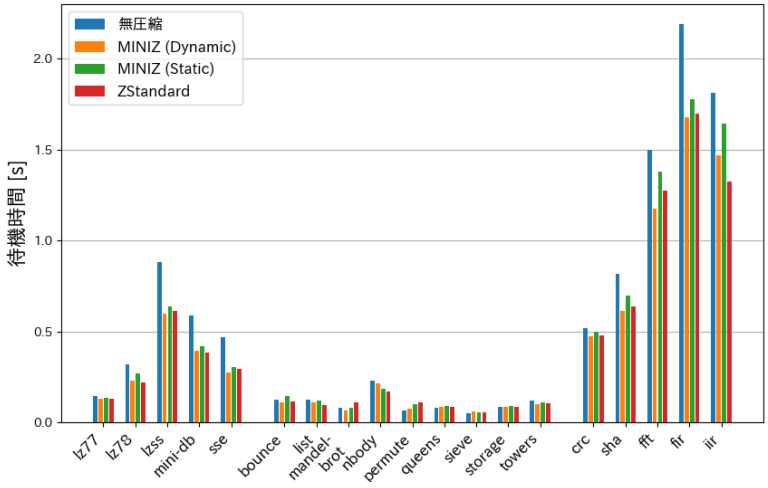

図 4.3. 無圧縮、 MINIZ 、ZStandard における実行時の待機時間の比較

る．グループ B における圧縮率の平均はそれぞれ 17.98 %、17.70 %、18.66 % となっており、待機時間の減少率の平均はそれぞれ 3.26 %、-8.65 %、-6.96 % となっている．このように圧縮率が全体平均を下回っているのは、グループ B のサイズが小さく、文字の出現頻度に十分な偏りがないためである．また、一部待機時間がむしろ増加してしまっているのは、転送サイズが小さく、 BLE 転送時間が小さいため、全体における解凍時間の割合が増加したためであると考えられる．しかし、実際にはグループ B のネイティブコードを転送するために、平均で 0.1 秒以下の待機時間しか発生しないため、待機時間が増加しても対話性の観点から問題とはならない．最後にグループ C における圧縮率の平均はそれぞれ 20.10 %、8.42 %、22.05 % となっており、待機時間の減少率の平均はそれぞれ 19.68 %、11.02 %、18.96 % となっている．グループ C はランダムなデータを多く含んでいるために、圧縮率が下がり結果的に待機時間の減少率も下がっているが、その中でも特に静的ハフマン符号化の圧縮率が低いことがわかる．

以上の結果により、比較した 3 つの圧縮手法の中で、特に既存手法である MINIZ + 最適ハフマン符号化が待機時間の減少率の観点から最も優れていることがわかった．グループ B の場合は待機時間がほとんど変化しないため、高い対話性を維持し、グループ A とグループ C においては待機時間を大幅に減少させ、 BlueScript 対話性を大きく改善していることがわかる．よって、既存手法は対話性の改善に有効であるが、一方で解凍のために SRAM を 41.17 [KB] 消費するため、メモリ資源が限られているマイコンでは問題がある．

4.4　既存手法と提案手法 2 における性能の比較

次に既存手法と提案手法 2 における圧縮率と待機時間の性能を比較する．各ベンチマークを既存手法と提案手法 2 で圧縮して送った場合の圧縮率を図 4.4、待機時間を 5 回測定した

平均を図 4.4 に載せている．ここで、lz77・bounce・crc それぞれから生成した静的ハフマンテーブルをフラッシュメモリ上に保存することで、圧縮・解凍時に複数のテーブルの中から圧縮率が最大のものを選択できるようにした．また、それにあたり、圧縮性能における公平性を考慮して以上の 3 つベンチマークを測定対象から外した．ちなみに、この実験 4.3 を含め、以降のすべての提案手法の計測において、提案手法 1 は有効となっている．

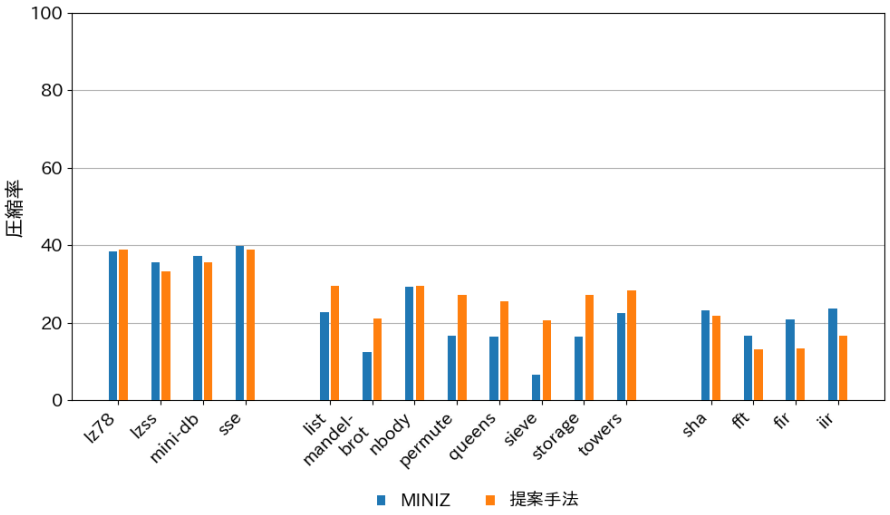

図 4.4. 既存手法と提案手法 2 における圧縮率の比較

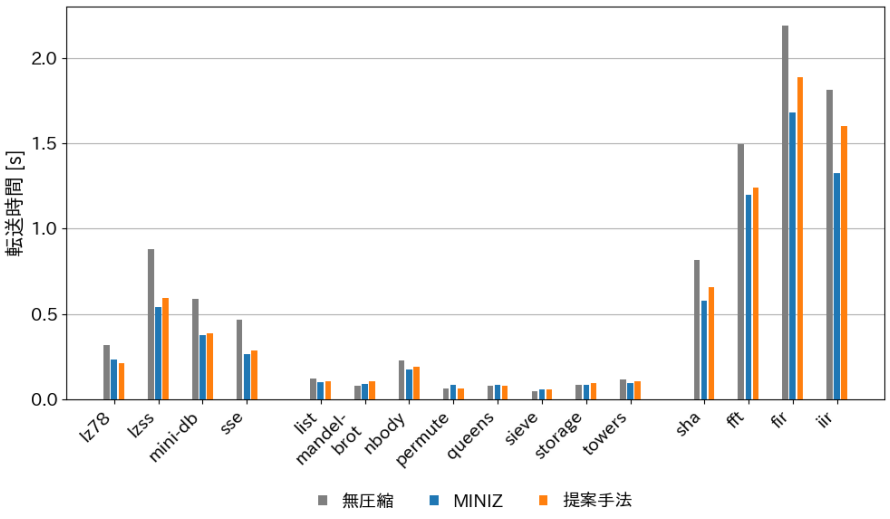

図 4.5. 既存手法と提案手法 2 における実行時の待機時間の比較

まず、提案手法 2 のベンチマーク全体における圧縮率の平均は既存手法の 23.72 % よりも 2.60 ポイント高い 26.32 % となっており、既存手法の待機時間からの減少率の平均は -6.54 % となっている．ここでまず注目すべきは、提案手法 2 では静的ハフマン符号化を使用して

いるにも関わらず、最適ハフマン符号化を使用している既存手法よりも圧縮率が高くなっている点である．これは提案手法 2 の ZIP ヘッダにはハフマンテーブルを再構築するためのデータを必要とせず、ヘッダサイズの減少量が静的ハフマン符号化による圧縮データサイズの増加量を上回ったためである．実験 4.3 における MINIZ + 静的ハフマン符号化でも同様にヘッダサイズは減少しているが、そこで利用されている静的ハフマン符号化による圧縮データサイズの増加量がヘッダサイズの減少量を上回ったため、提案手法 2 とは異なり、圧縮率が悪化してしまっている．しかし、圧縮率が向上しているにも関わらず待機時間が既存手法よりも増加しているのは、提案手法 2 ではハフマン符号を復号する度に読み出し速度が SRAM よりも 5.6 倍遅いフラッシュメモリへアクセスする必要があるためである．それでも尚、無圧縮の場合に比べ待機時間の減少率は 9.58 % となって

ベンチマーク・グループ別で見ると、グループ A では提案手法 2 の圧縮率は既存手法から平均で 1.19 ポイント悪化しており、待機時間は平均で 2.00 % 改善している．よってグループ A においては提案手法 2 と既存手法の性能の差が小さいことがわかる．グループ B では提案手法 2 の圧縮率は既存手法から平均で 8.23 ポイント改善し、待機時間は平均で 10.18 % 悪化している．圧縮率が大きく上昇しているのはグループ B のコードサイズは小さいため、先述したヘッダサイズの減少の影響がより強く影響しているためである．そして、待機時間が増加しているのは、BLE 転送時間が小さいことによる解凍処理時間の割合の増加、特にフラッシュメモリへのアクセス時間の問題影響を強く受けているためだと考えられる．しかし、実験 4.3 でも述べたように、グループ B の待機時間はそもそも小さいため、ここでの性能悪化は対話性の観点から問題にはならない．最後にグループ C では提案手法 2 の圧縮率は既存手法から平均で 4.87 ポイント悪化し、待機時間は平均で 7.8 % 悪化している．このような圧縮率の減少は実験 4.3 でもあったように、静的ハフマン符号化はランダムなデータをうまく圧縮できないことに由来するためであるが、実験 4.3 の静的ハフマン符号化と比べ提案手法 2 の圧縮率は平均で 2.98 ポイント高くなっている．このような性能の向上は複数の BlueScript のネイティブコードから生成した静的ハフマンテーブルを使用することで、図 3.3 で示したようなテーブル間のさをうまく吸収した結果であると考えることができる．

このように提案手法 2 を使用することによって、確かに既存手法より待機時間が 6.54 % 増加してしまうが、圧縮しない場合に比べ 9.58 % 待機時間が減少する．さらに、対話性に問題のないグループ B を除いた場合だと、提案手法 2 の待機時間は無圧縮の場合に比べ 25.69 % 減少し、対話性を大きく改善することがわかる．そして、提案手法 2 では既存手法から SRAM 消費量が 99.70 % をも削減しており、マイコン上で SRAM をほとんど消費することなく対話性の向上が可能である．

最後に対話性が低いグループ A とグループ C の性能評価する．圧縮率は既存手法から平均でそれぞれ 3.03 ポイントの向上となっており、待機時間の減少率は平均で -2.90 % となっている．これはヘッダサイズの減少によって圧縮サイズが減ったものの、フラッシュメモリアクセスのオーバヘッドによって解凍時間が多少伸びてしまったためである．このようにして提案手法 2 は既存手法より少しの対話性悪化のオーバヘッドで、フラッシュメモリ上の静的ハフマンテーブルの利用により約 8 KB も SRAM を節約することができることがわかった．

4.5　既存手法と提案手法 3 の比較

次に既存手法と提案手法 3 おける圧縮率と待機時間の性能を比較する．ソフトリセット前後のネイティブコードの変化が提案手法 3 の性能にどのような影響を与えるかを調べるために、1) ソフトリセット前後で異なるアプリケーションを実行した場合、 2) ソフトリセット前後で少しだけ異なるアプリケーションを実行した場合、 3) ソフト前後で同じアプリケーションを実行した場合、という三つの状況を設定し、それぞれにおいて測定を行った．状況 1 を再現するために、ソフトリセット前にグループ A の lz78 を実行し、ソフトリセット後にそれ以外の各ベンチマークを実行したときの性能を計測した．状況 2 を再現するために、ソフトリセット前では各ベンチマークを実行し、ソフトリセット後では各ベンチマークに 4.1 で述べた関数を各ベンチマークに挿入したものを実行したときの性能を測定した．最後に、状況 3 ではソフトリセット前後に同じベンチマークを実行したときの性能を測定した．このようにして lz78 を状況 1 に使用するため、計測対象から外している．圧縮率の結果を図 4.6 、待機時間を 5 回計測した平均を図 4.7 に載せている．

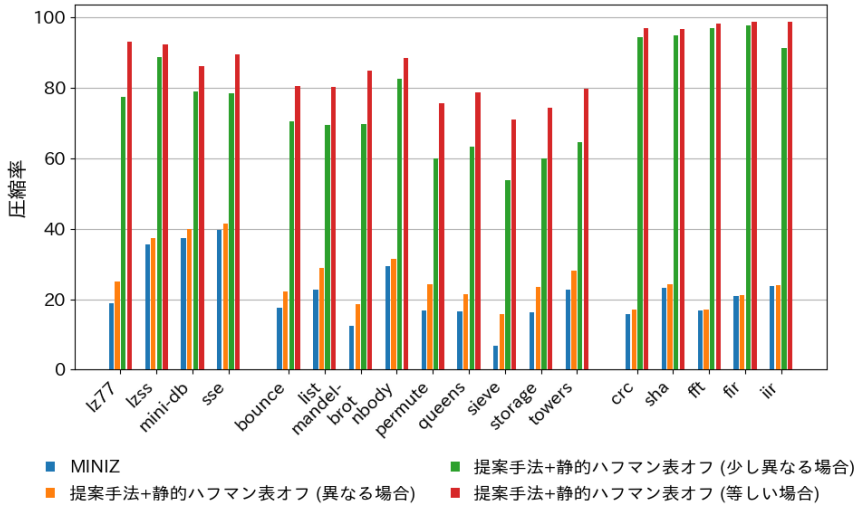

図 4.6. 既存手法と提案手法 3 における圧縮率の比較

まず、提案手法 3 の状況 1, 2, 3 の圧縮率は既存手法の 21.94 % から平均でそれぞれ 4.14 ポイント、60.12 ポイント、69.85 ポイント改善し、待機時間は平均でそれぞれ 2.76 % の悪化、50.31 % の改善、 61.50 % の改善となっている．状況 1 では、ソフトリセット後とは異なるネイティブコードが辞書に含まれいるため、圧縮率の上昇は限定的であり、フラッシュメモリへのアクセスのオーバヘッドにより待機時間が少し伸びている．一方で、状況 2 と状況 3 においては、ソフトリセット前と類似したコードが辞書に含まれているため、非常に高い圧縮率を達成し、転送サイズが大きく減少したことによって、ボトルネックであった BLE 通信時間を削減し、結果的に対話性を大きく改善することができるとわかる．

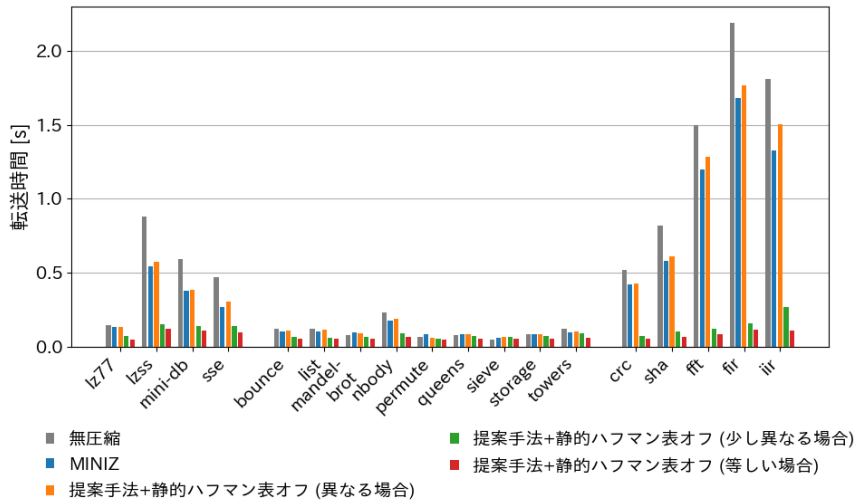

図 4.7. 既存手法と提案手法 3 における実行時の待機時間の比較

グループ別で見ると、グループ A では各状況の圧縮率は既存手法から平均でそれぞれ 3.33 ポイント、51.33 ポイント、60.94 ポイント改善し、待機時間は平均でそれぞれ 1.29 % の悪化、59.69 % の改善、74.16 % の改善となっている．今まで通り、グループ A においては高い圧縮率に伴う対話性の大幅な改善が見られる．グループ B では各状況の圧縮率は既存手法から平均でそれぞれ 6.44 ポイント、55.50 ポイント、69.10 ポイント改善し、待機時間は平均でそれぞれ 5.38 % の悪化、 25.52 % の改善、39.60 % の改善となっている．圧縮率の上昇が平均を大きく上回っているのは、既存手法はグループ B に対する圧縮率が 17.98 % と元々低かったため、上昇幅の余地が大きかったからである．一方で待機時間の改善が平均より劣っているのは、コードサイズが小さく BLE 通信時間の待機時間全体への割合が小さいためである．

グループ C では各状況の圧縮率は既存手法から平均でそれぞれ 0.64 ポイント、75.47 ポイント、78.32 ポイント改善し、待機時間は平均でそれぞれ 0.77 %、87.42 %、90.81 % 改善している．状況 2 と状況 3 においては元々の圧縮率が低かったこともあり、圧縮率が大きく上昇し、結果的に待機時間も大幅に短縮していることがわかる．グループ C においてのみ状況 1 における圧縮率の向上が 0.64 ポイントと限定的なのは、ランダムなデータが多く含まれているためだと考えられる．

最後に対話性が低いグループ A とグループ C における性能を評価する．各状況の圧縮率は既存手法から平均でそれぞれ 1.84 ポイント、64.74 ポイント、70.60 ポイントの向上となっており、待機時間の減少率はそれぞれ -0.15 %、75.09 %、83.41 % となっている．このように提案手法 3 は SRAM 消費が追加で発生することなく、対話性が低いアプリケーションの実行において、最善の場合で無圧縮から 87.38 % も対話性が向上し、最悪の場合でも既存手法と同程度の向上が達成できることがわかった．

4.6　既存手法と全提案手法の比較

最後に既存手法と全提案手法における圧縮率と待機時間の性能を比較する．これまでの実験と同様に、lz77、bounce、crc から作成した静的ハフマンテーブルを利用し、ソフトリセット前後でコードが変化する 3 つの状況を用意した．それに伴い、lz77、bounce、crc、lz78 の 4 つは計測対象から外している．圧縮率の結果を図 4.8 に、待機時間を 5 回計測した平均を図 4.9 に載せている．

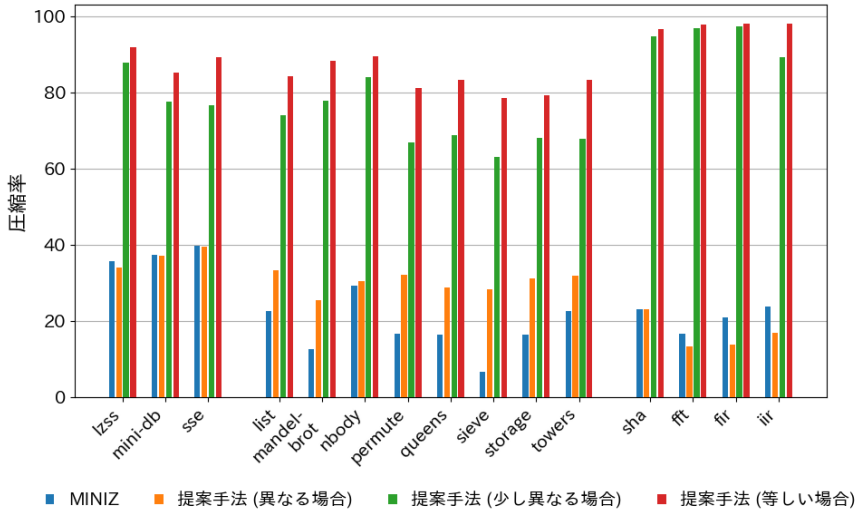

図 4.8. 既存手法と全提案手法における圧縮率の比較

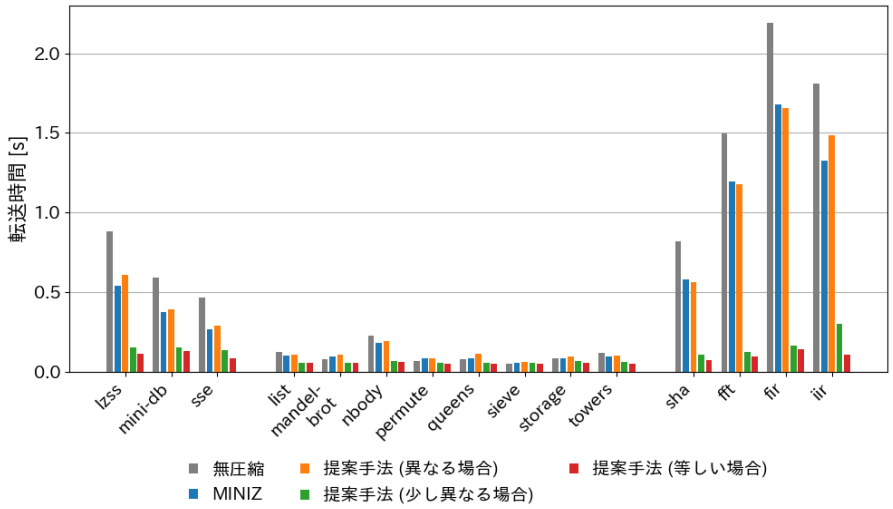

図 4.9. 既存手法と全提案手法における実行時の待機時間の比較

まず、全提案手法の状況 1, 2, 3 の圧縮率は既存手法の 21.14 % から平均でそれぞれ 4.34 ポイント、60.16 ポイント、70.03 ポイント改善し、待機時間は平均でそれぞれ 5.63 % の悪化、51.08 % の改善、60.03 % の改善となっている．これは実験 4.5 での全体平均と同じ傾向を示している．グループ別で見ると、グループ A では各状況の圧縮率は既存手法から平均でそれぞれ 0.64 ポイントの悪化、48.05 ポイントの改善、56.25 ポイントの改善となっており、待機時間は平均でそれぞれ 3.65 % の悪化、 66.32 % の改善、 77.27 % の改善となっている．状況 1 においては実験 4.4 での提案手法 2 の圧縮率より 0.5 ポイント程度向上している一方で、待機時間が伸びている．これは辞書が増加したことにより圧縮率が向上したものの、フラッシュメモリアクセスによるオーバヘッドが増加したためであると考えられる．グループ B では各状況の圧縮率は既存手法から平均でそれぞれ 10.56 ポイント、 57.92 ポイント、71.76 ポイント改善し、待機時間は平均でそれぞれ 6.80 % の悪化、28.30 % の改善、38.57 % の改善となっている．グループ C では各状況の圧縮率は既存手法から平均でそれぞれ 4.37 ポイントの悪化、73.71 ポイントの改善、76.89 ポイントの改善となっており、待機時間は平均でそれぞれ 4.78 % の悪化、85.20 % の改善、90.03 % の改善となっている．グループ B と C も実験 4.5 でのそれぞれの結果と同じ傾向を示していることがわかる．

最後に対話性が低いグループ A とグループ C における性能を評価する．各状況の圧縮率は既存手法から平均でそれぞれ -2.77 ポイント、+62.71 ポイント、+68.05 ポイント変化しており、、待機時間の減少率はそれぞれ -4.30 %、77.11 %、84.56 % となっている．まず、状況 1 における全提案手法の圧縮率が提案手法 3 を使用していない場合とは異なり、既存手法悪化してしまっているのは、ソフトリセット後に LZ77 辞書が変化し、ハフマンエンコーダに入力される LZ77 符号列に変化が生じたためだと考えられる．このようにして本提案手法は既存手法から 99.70 % も SRAM 消費量を削減すると同時に、対話性が低いアプリケーションの実行において、最善の場合で無圧縮から 89.345 % も対話性を改善し、最悪場合では既存手法から多少の対話性におけるオーバヘッドが発生するが、それでも無圧縮から 24.69 % も改善することがわかった．

第 5 章

結論

5.1 まとめ

本論文は BlueScript で大きなネイティブコードを BLE 転送する場合における対話性の改善の手法として ZIP 圧縮を使用したいが、マイコン上での ZIP 解凍処理による SRAM 消費量が大きすぎると言う問題の解決策として、次の機能を MINIZ 上で実装した.

1. 解凍された SRAM 上のネイティブコードを LZ77 辞書として利用する.
2. 圧縮・解凍時にフラッシュメモリ上の事前作成されたハフマンテーブルを使用する
3. ソフトリセット時に古いコードをフラッシュにコピーし、 LZ77 辞書として利用する.

既存手法では LZ77 辞書のために SRAM から 32 KB のメモリ領域を確保するが、提案手法 1 では ZStadard の Stable Buffer に倣い出力バッファを LZ77 辞書として利用することで、 LZ77 辞書のために SRAM を一切消費しないようにした. また、既存手法では最適ハフマンテーブルを解凍側で再構築するために SRAM から 8 KB のメモリ領域を確保するが、提案手法 2 では事前作成されたハフマンテーブルをフラッシュメモリに配置することで、それらの領域を節約することができる. 実験により、提案手法 1 と提案手法 2 によって、 SRAM 消費量が既存手法から 99.70 % も削減することができ、また提案手法 2 によって多少のフラッシュメモリアクセスのオーバヘッドが発生するものの、既存手法と同程度の対話性の向上を維持することがわかった.

提案手法 3 では、ソフトリセット時にネイティブコードを LZ77 辞書として再利用することによって、追加の SRAM 消費なく既存手法よりも高い圧縮率を達成するようにした. 実験により、一切の SRAM 消費が発生することなく、最悪の場合で既存手法と同程度の水準、最善の場合で既存手法から大幅に高い水準で対話性を改善することがわかった.

そして、すべての提案手法を同時に利用した場合における対話性の改善は、最悪の場合で既存手法から 4.30 % 悪化するが、それでも無圧縮から 24.69 % 対話性が改善するため、十分有効であることがわかる. さらに、最善の場合では既存手法より 84.56 % も対話性が向上することがわかった.

以上のように、本提案手法は ZIP 解凍による SRAM 消費量を大幅に削減しつつ高い圧縮

率を達成しており、効率的な BLE 転送のためにマイコン上で ZIP 解凍を行う際の SRAM 消費量が大きすぎる、と言う問題を十分に対処できていると言える.

5.2 今後の課題

まず、 BLE 通信時間が短縮されるにつれ、フラッシュメモリアクセスによるオーバヘッドが目立ってくるため、より高い対話性の改善のために、それを高速化することが考えられる. 例えば、 ESP-D0WDQ6-V3 には 64 KB のフラッシュメモリ上のキャッシュがあるため、有効に活用することで、キャッシュヒット率を高め、解凍処理時間を短縮する可能性がある.

次に、より汎用的に様々なネイティブコードを圧縮できるように、使用する静的ハフマンテーブルの圧縮性能を高め、テーブル間で異なる傾向を示すようにしたい. 現在ではベンチマーク・グループ A、B、C からランダムに一つずつ選んで利用しているが、今後はより平均的な圧縮率の観点から最適化を行いたい.

そして、現在のシステムではソフトリセット時に退避するデータがホストマシンとマイコンとで等しいという前提で実行しているが、今後はデータ整合性の機能を追加したい. BlueScript はシャドウメモリによってホストマシンとマイコンでは同じネイティブコードを持っているはずだが、伝送路の信頼性やシステム上のバグなどの問題によって両者が持つコードで差異が生じてしまう可能性は否めない. 例えば、ソフトリセット時にホストマシンとマイコンは所持しているネイティブコードからチェックサムを計算・保存し、セッション開始時にホストマシンへと送ることで、ホストマシンは両者が持っているコードが同じものかどうかを識別し、辞書として利用するかどうかを判定することが考えられる.

最後に、現在は一つ前のセッションのネイティブコードまでしか辞書として利用できていないが、複数回先のコードを辞書として利用することで、圧縮率をより高めていくことが考えられる. しかし、無造作に増やすだけでは辞書に冗長なデータだけが増え、結果的に限りあるフラッシュメモリを浪費しかねない. そこで、ホストマシンとマイコン上で有効なパターンだけを抽出する処理を加えることで、より先のコードまで再利用することができ、圧縮率をさらに高めることができる考えられる.

発表文献と研究活動

(1) 渡邉純一, 前島文香, 山崎徹郎, 千葉滋．SRAM とフラッシュメモリの使い分けで高速化を狙うマイコン用のコード領域管理システム．Programming and Programming Language Workshop (PPL), 2024.03.03-2024.03.05．(ポスター発表)

参考文献

[1] csg tokyo. Bluescript. https://github.com/csg-tokyo/bluescript, 2024.

[2] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996.

[3] Yann Collet and Murray Kucherawy. Zstandard Compression and the 'application/zstd' Media Type. RFC 8878, February 2021.

[4] Damien George. Micropython. https://micropython.org/, 2023.

[5] ESPRESSIF. Esp32 series datasheet version 4.8. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf, 2025.

[6] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, Vol. 40, No. 9, pp. 1098–1101, 1952.

[7] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337–343, 1977.

[8] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. 24, No. 5, pp. 530–536, 1978.

[9] richgel999. Miniz. https://github.com/richgel999/miniz, 2025.

[10] ESPRESSIF. esp-idf. https://docs.espressif.com/projects/esp-idf/en/stable/, 2025.

[11] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, 2004.

[12] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Zephyr: efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, p. 32, USA, 2009. USENIX Association.

[13] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, p. 81–94, New York, NY, USA, 2004. Association for Computing Machinery.

[14] Wei Dong, Biyuan Mo, Chao Huang, Yunhao Liu, and Chun Chen. R3: Optimizing

relocatable code for efficient reprogramming in networked embedded systems. In *2013 Proceedings IEEE INFOCOM*, pp. 315–319, 2013.

[15] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded risc processors. *SIGPLAN Not.*, Vol. 34, No. 5, p. 139–149, May 1999.

[16] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, Vol. 22, No. 2, pp. 378–415, March 2000.

[17] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pp. 120–131, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Ignas Plauska, Agnius Liutkevicius, and Audrone Janaviciute. Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller. *Electronics (Switzerland)*, Vol. 12, No. 1, p. 143, dec 2022.

謝辞

本研究を行うに当たり，多大なるご指導をいただきました指導教員の千葉滋教授に心より御礼申し上げます．また，千葉研究室の皆様からは様々なことをご教示いただきました．この場を借りて御礼申し上げます．