情報処理学会論文誌　プログラミング　Vol.18　No.4　1–9 (Oct. 2025)

マイコン用対話的言語処理系BlueScriptのための メモリ消費量の少ないデータ圧縮手法

渡邉 純一[1,a)]　望月 文香[1,b)]　山崎 徹郎[1,c)]　千葉 滋[1,d)]

受付日 2025年2月17日, 採録日 2025年6月4日

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，高い対話性を備えている．BlueScript はアプリケーションを実行する際，ホストマシン上でコンパイルし，そのネイティブコードを Bluetooth Low Energy（BLE）によってマイコンへ転送する．しかし，BLE は低消費電力であるが帯域幅が狭く，ライブラリをインポートしているアプリケーションを実行する場合など，比較的大きいネイティブコードを転送する際に通信時間が増大し，結果的に BlueScript の対話性が低下してしまう．そのためにデータ圧縮を利用して転送サイズを減らすことが考えられるが，既存の圧縮アルゴリズムは限られたマイコンのメモリ資源を多く消費してしまい，ロードできる BlueScript のネイティブコードのサイズなどに影響を及ぼしてしまう．そこで本論文では ZIP を例にとり，マイコン上で必要なメモリ消費量を抑えつつデータ圧縮を行って通信時間を削減することで，BlueScript の対話性を改善するための 3 つの手法を提案する．実験を通して，提案手法は既存手法よりもメモリ消費量を大幅に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を確認する．

キーワード：BlueScript，マイコン，組み込みシステム，対話的言語処理系，データ圧縮

Data Compression Method with Low Memory Consumption for BlueScript, an Interactive Language Runtime System for Microcontrollers

Junichi Watanabe[1,a)]　Fumika Mochizuki[1,b)]　Tetsuro Yamazaki[1,c)]　Shigeru Chiba[1,d)]

Received: February 17, 2025, Accepted: June 4, 2025

Abstract: BlueScript is an interactive language runtime system for microcontrollers (MCUs) with a novel architecture. It compiles applications on a host machine and transfers the native code via Bluetooth Low Energy (BLE). However, limited BLE bandwidth increases transfer time, especially for larger applications with imported libraries, reducing interactivity. To mitigate this, we consider utilizing data compression to reduce transfer size. Traditional compression methods, however, demand significant MCU memory, limiting the size of executable native code. In this paper, we propose three methods to improve the interactivity of BlueScript by taking ZIP as an example, aiming to reduce communication time through data compression while minimizing memory consumption on microcontrollers. Experimental results show significant memory reduction and improved interactivity compared to the current approach.

Keywords: BlueScript, microcontroller unit, embedded system, interactive language runtime system, data compression

[1] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] watanabe@csg.ci.i.u-tokyo.ac.jp
[b)] fumika.maejima@csg.ci.i.u-tokyo.ac.jp
[c)] yamazaki@csg.ci.i.u-tokyo.ac.jp
[d)] chiba@csg.ci.i.u-tokyo.ac.jp

1. はじめに

マイコン開発では開発が複雑化する中で対話的な開発環境の需要が高まっているが，アプリケーションプログラムの実行時性能も犠牲にすることはできない．BlueScript [1]

はそういったニーズを受けて開発された対話性と高い実行時性能を両立させた開発環境および言語処理系である．BlueScript では動的に追加されるプログラムをホストマシン上でコンパイル・リンクしてから，ネイティブコードを Bluetooth Low Energy（BLE）を通じてマイコンへと転送し，実行する．

BlueScript ではライブラリをインポートできる機能が提供されている．インポートされるライブラリもまた，他のプログラムと同様にホストマシン上でコンパイルされてからマイコンへと BLE で転送される．このとき，もしインポートしたライブラリが巨大であると，BLE の通信速度は遅いため，BlueScript の対話性を損なってしまう．

通信時間の短縮のためにデータ圧縮により転送サイズを削減することが考えられるが，マイコン開発では SRAM の消費量を抑える必要がある．たとえばマイコンの一種である ESP32 は 520 KB の SRAM を搭載しているが，これに対してマイコンでも使用される ZIP ライブラリである MINIZ は約 40 KB のメモリを消費する．解凍に使用するデータを容量の大きなフラッシュに配置するという方法も考えられるが，フラッシュへの書き込みは時間がかかるため，やはり対話性への影響が懸念される．

そこで本論文ではマイコン用対話的言語処理系である BlueScript 向けの，低 SRAM 消費な ZIP 解凍の実装方法を提案する．提案手法は ZIP 圧縮に使用される deflate アルゴリズムをベースに，SRAM 消費量の大きい LZ77 辞書やハフマン表を処理系内部のデータを再利用する，事前計算したものをフラッシュに配置して使用するなどの方法で SRAM の消費量を抑える．これにより，提案手法はわずか 124 バイトの SRAM 消費だけで圧縮されたネイティブコードを解凍することができる．また，BlueScript のような対話的な環境ではリセット後に同じようなプログラムが再実行されやすいという特徴をふまえ，リセット時に退避したデータをリセット後に再利用することで，より圧縮性能を向上させる．

また我々は提案手法が SRAM の消費量を削減したことで，圧縮率，解凍時間，転送時間がどの程度悪化するのかを，初回時とリセット後 2 通りの状況で確認する実験を行った．実験の結果，初回の実行では提案手法は ZIP 圧縮と比較して圧縮率の大きな変化は見られず，解凍時間は増大するものの，合計の転送時間には圧縮率による BLE 通信の時間への影響が大きくあまり増大しないことが観察された．リセット後に少しだけ異なるプログラムを再実行した場合は ZIP 圧縮と比較して圧縮率が大幅に改善し，やはり解凍時間は増大するものの，合計の転送時間が大幅に短縮されることが観察された．

以降本論文では次のような構成となっている．まず，2 章で BlueScript の対話性を改善するために高圧縮ライブラリをそのまま用いることによる問題点について述べる．次に 3 章で提案手法である SRAM 消費量を削減する ZIP 解凍の実装方法の詳細について述べる．そして 4 章で提案手法と既存手法の性能を測定・比較する．5 章では本論文の関連研究をいくつか紹介する．最後に 6 章で全体の内容の総括を載せる．

2. BlueScript の対話性の向上のために既存の高効率圧縮を利用した際の問題点

マイコンの開発では実行時性能とともに対話性が大事だが，たとえば MicroPython [2] は対話的な実行が可能である一方で実行時性能は決して高くはない．その一方で，BlueScript のように対話性と高い実行時性能を両立させた開発環境・言語処理系も提案されている．マイコン上でアプリケーションを開発する際，開発者はマイコンでの動作を確認しながら，機能やパラメータの調整，デバッグなどの小規模な変更を繰り返すため，対話的なマイコン向け開発環境が求められている．そのような要請に応えるべく設計された BlueScript はマイコンの開発を対話的に行うためのノートブック開発環境および言語処理系である．ユーザがセルと呼ばれる領域にプログラムを書き，実行すると，セル内のプログラムがネイティブコードへコンパイルされたのちに BLE によりホストマシンからマイコンへと転送・ロード・実行される．これにより，容易に機能を追加したり，処理を高い実行時性能で実行したりできる．

しかしながら，現在の BlueScript では，ライブラリのインポートなどのために長いプログラムを転送する際に，転送に長い時間がかかり対話性が損なわれるという問題がある．BlueScript では，あらかじめ準備されたライブラリをインポートすることで，その機能を利用できるようになる．ユーザがセル内でライブラリをインポートし，実行すると，セル内のプログラムと同様にライブラリもネイティブコードへとコンパイルされ，BLE によってマイコンへと転送される．このときライブラリは，セル内に直接書かれるプログラムに比べ長い場合があり，また BLE は通信速度がとても遅いため，転送に長い時間がかかり，その結果対話性が損なわれてしまうことがある．たとえば，ライブラリをインポートせずにセルを実行する場合，コンパイルを開始してからマイコンでの実行が完了するまでの時間は長くとも 0.5 秒前後であるが，Display というライブラリをインポートする場合，約 30 KB のネイティブコードが転送されるため，応答に約 4 秒もかかってしまう．

大きなライブラリコードの転送にかかる時間を短縮する方法の 1 つはネイティブコードを送る前に圧縮して小さくすることであるが，既存の高圧縮ライブラリではマイコン上の SRAM を多く消費してしまうため，そのままでは採用しづらい．既存の高圧縮ライブラリの代表として ZIP や ZStandard [3] があげられる．これらはどちらも LZ77 圧縮 [4] とハフマン符号化 [5] を組み合わせて使用す

るため，圧縮と解凍のどちらでも LZ77 辞書とハフマン表という大きなテーブルが必要となる．これらのテーブルの置き場所としてアクセス速度は速いが容量が小さい SRAM と，容量は大きいがアクセス速度が遅いフラッシュメモリという 2 種類のメモリデバイスが考えられる．対話性を考慮するとこれらのテーブルは SRAM に配置する方が望ましいが，容量が小さい SRAM がこれらにより圧迫されてしまう．BlueScript は実行時性能のためにネイティブコードを SRAM 上にロードするが，転送時間を短縮するためにこれらの圧縮をそのまま利用すると，開発に利用できる SRAM が減ってしまう．これにより，ユーザが開発できるプログラムのサイズが制限され機能追加やデバッグができなくなったり，メモリ不足による実行時エラーによりセンサネットワーク上で障害を引き起こしたりする可能性がある．

3. 解凍時の SRAM 消費を減らすデータ圧縮の実装方法

前章で述べた問題に対処するために，圧縮されたネイティブコードを少ない SRAM 消費で解凍するマイコン上の解凍器の実装方法を提案する．この実装方法は ZIP 圧縮に，1) コードの LZ77 辞書化，2) 静的ハフマン表，3) リセット前のコードの再利用，という 3 つの手法を組み合わせることで実現する．

3.1 ZIP 圧縮を用いたコードの転送

まず，マイコン向け対話的言語処理系に ZIP を単純に適用した場合の様子を，BlueScript を例に，**図 1** に示す．提案する実装方法は ZIP を拡張したものなので，先に ZIP について述べる．ZIP 圧縮と解凍を行う際，それぞれ LZ77 圧縮・ハフマン符号化，ハフマン復号化・LZ77 解凍の順で処理が発生する．

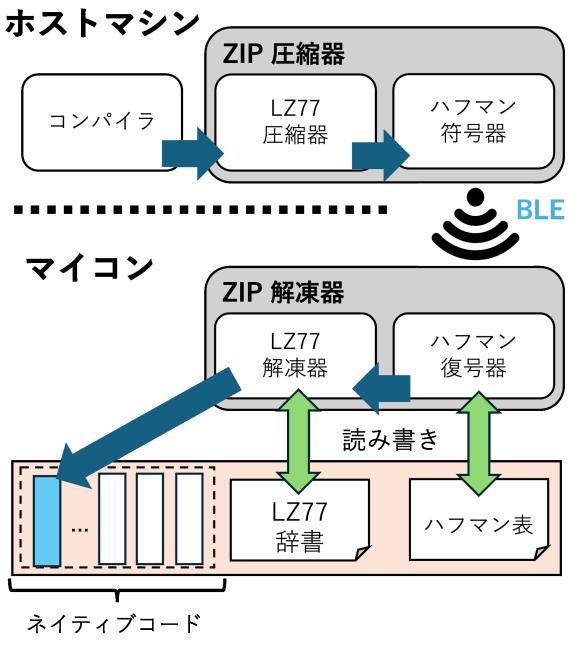


図 1　BlueScript で ZIP を利用する際の処理の流れ
Fig. 1　Processes for using ZIP in BlueScript.

LZ77 圧縮はデータ内で繰り返し出現するパターンを，最初の出現位置への参照に置き換えるデータ圧縮アルゴリズムである．LZ77 圧縮でパターンが冗長であるかを探索するために，また LZ77 解凍を行うために，特定の範囲までの過去データを保持する辞書と呼ばれる領域が必要である．LZ77 解凍時には，圧縮データを辞書内で解凍してから，解凍されたデータを出力するが，これにより辞書内に過去データがそのまま追加された状態となる．

ハフマン符号化はデータを文字列として見たとき，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を割り当てることでより短い符号列を生成する符号化である．ZIP では一定の入力ごとに，LZ77 圧縮し，その結果に対して，文字とハフマン符号の対応関係を記したハフマン表を新しく作成し，それを使用してハフマン符号化を行う．そして，ハフマン表を復号器側へ共有するために，新しく作成されたハフマン表の情報を ZIP の圧縮データのヘッダに含める．このように頻繁にハフマン表を更新することで効率的に符号化できるが，復号化する前にヘッダ内の情報からハフマン表を再構築する必要があり，そのための複数の作業領域を用意する必要がある．

図 1 では，コードがロードされるまでの BLE 以外の流れを青の矢印，各テーブルへのアクセスを緑の矢印で表している．コンパイラはネイティブコードを生成すると，ZIP 圧縮器へと渡し，圧縮データを出力する．圧縮データは BLE によってホストマシンからマイコンへと転送され，ZIP 解凍器へと渡される．ZIP 解凍器のハフマン復号器は，まず ZIP ヘッダからハフマン表を再構築し，次に LZ77 圧縮データへと復号化する．LZ77 解凍器は辞書で圧縮データをコードへと解凍し，最終的な出力先であるネイティブコードの配置先へとコピーすることでロードが完了する．

3.2 コードの LZ77 辞書化

提案する最初の手法は配置済みのネイティブコードの LZ77 辞書化である．本手法では，LZ77 解凍器が SRAM 上のネイティブコードの配置先をそのまま LZ77 辞書として利用する．元々 LZ77 解凍器は受け取ったデータを LZ77 辞書の末尾に追加される形で解凍してから，最終的な出力先であるネイティブコードの配置先へとコピーしていた．本手法では，LZ77 解凍器にネイティブコードの配置先を LZ77 辞書として渡すことで，ネイティブコードが配置先上でそのまま末尾に追加される形で解凍されるようにしている．BlueScript ではネイティブコードの配置先の位置は実行された順に逐次的に決まるが，本手法を利用してもネイティブコードは実行順に SRAM 上に追加して配置されていくため，そのまま実行可能となる．これによって，

LZ77 辞書のために追加の SRAM 領域を確保する必要がなくなる．

3.3　静的ハフマン表

次に提案する手法ではまず，BlueScript のファームウェアをマイコンに書き込む際に，複数の異なる BlueScript プログラムからそれぞれ事前作成されたハフマン表をフラッシュメモリに保存する．コードを圧縮する際にどのハフマン表を使用するかは圧縮側のホストマシンが決定し，そのハフマン表に割り振られているインデックスを転送データのヘッダに含める．解凍側のマイコンはヘッダから使用するハフマン表を決定し解凍する．このとき，圧縮側であるホストマシンは実行時性能が良い PC であるため，複数のハフマン表を使用して圧縮を実際に行い，最も圧縮できた表を選択する．

SRAM の節約のためにフラッシュメモリを利用してハフマン復号化を行いたいが，フラッシュメモリからのデータの読み出しは SRAM の数倍程度であるの対し，書き込みは SRAM の数百倍程度にも達するため，受信した ZIP ヘッダからハフマン表をフラッシュメモリ上で再構築することはオーバヘッドが非常に大きい．それを回避するためにハフマン表を事前にフラッシュメモリに書き込み，復号化時に読み出しのみを行うようにする．これによりハフマン表を再構築するための作業領域が必要なくなり，ハフマン表を保持するための領域を SRAM 上に確保する必要がなくなる．事前作成されたハフマン表は，表の作成元に用いたネイティブコードと圧縮対象であるネイティブコードの間における各文字の出現頻度の違いによって圧縮性能が変わってくるが，傾向の異なるハフマン表を複数用意することで，どのようなネイティブコードに対しても良い圧縮効率を達成するようにする．

3.4　リセット前のコードの再利用

最後に提案する手法では，ユーザがソフトリセットを実行するとき，ホストマシンでは今まで転送してきたネイティブコードを保持し続け，マイコンではそれまで SRAM 上に配置していたネイティブコードをフラッシュメモリに移動するようにする．そしてソフトリセット後に圧縮側と解凍側の両方で古いネイティブコードが LZ77 辞書に含まれているものとして圧縮・解凍を実行し，圧縮率を改善する．フラッシュメモリに移動することで，追加の SRAM の消費をともなわずに古いネィティブコードも圧縮に利用できる．

BlueScript のノートブック開発環境では，ソフトリセットを実行すると，それまで SRAM 上に配置されたネイティブコードやホストマシン上の状態などを初期化することできる．ノートブック開発環境のユーザはしばしば，ある程度テストが終わると 1 度ソフトリセットをかけて，最終版のプログラムを再実行することがある．この際，ソフトリセット前後で似たプログラムが実行されることが多いため，本来消去される古いネィティブコードを LZ77 辞書として再利用することで圧縮性能を向上させることができる．

提案する手法では，ソフトリセットを行ってもホストマシン上の LZ77 圧縮器も，マイコン上の LZ77 解凍器もどちらも初期化しない．ただし解凍側のマイコンの SRAM 上に配置してあった古いネイティブコードはフラッシュメモリ上にコピーして保存する．ソフトリセット後は，このコピー先のフラッシュメモリ上の領域と元の SRAM 上のネイティブコード領域をこの順で連結したものを，解凍したネイティブコードを配置する論理的なメモリ領域として用いる．ソフトリセットにより，新しく解凍したネイティブコードを配置する空き領域の先頭アドレスは SRAM 上の領域の先頭に戻される．提案する手法ではネィティブコードを配置するメモリ領域を LZ77 辞書として用いるので，このように論理的に連結された 1 つの領域を解凍したネイティブコードを配置するメモリ領域とすることで，古いネイティブコードも LZ77 辞書に含まれているものとして圧縮・解凍を実行できる．フラッシュメモリを用いるので SRAM 上に追加の領域は必要ない．

4.　実験

我々は 3 章で提案した手法を BlueScript 処理系に実装した．本章ではこれを用いて実際に削減された SRAM の消費量を測定し，また提案手法による圧縮効率，解凍速度，転送速度の変化を測定する．

マイコン向け対話的言語処理系に BlueScript ver. 1.0 を使用している．ホストマシンには Apple M1 を搭載した MacBook Pro，マイコンには 520 KB の SRAM と 4 MB のフラッシュメモリを持つ ESP32-D0WDQ6-V3 を使用している．

ZIP ライブラリにはマイコンでの利用事例が多い MINIZ [6] を使用しており，提案手法も MINIZ を拡張して実装している．MINIZ などの ZIP ライブラリで使用される Deflate アルゴリズムでは，ハフマン符号化ではバイト単位で読み込んだデータに対して 16 ビット以下の符号を生成し，また LZ77 圧縮ではバイト単位でパターンマッチを行い，参照先のオフセットを表すために 2 バイト，長さのために 1 バイト使用する．提案手法における静的ハフマン表と古いネイティブコードの配置先として，フラッシュメモリ上にそれぞれ 576 KB と 1 MB の領域を確保する．

性能を測定するために，実験の中で**表 1**，**表 2**，**表 3** にある BlueScript プログラムを転送する．表 1 のプログラムは GitHub 上に公開されている C のプログラムを，表 2 と表 3 はそれぞれ are we fast yet? [7] とマイコンの実行時性能を測るため [8] のベンチマークプログラムを BlueScript に書き換えたものである．また，比較対象として MINIZ

表 1　BlueScript ベンチマークプログラム (1)
Table 1　BlueScript benchmark programs (1).

ベンチマーク名	説明	LOC	コンパイル後のバイナリサイズ [B]
lz77 [*1]	LZ77 圧縮の実装	156	1,431
lz78 [*2]	LZ78 圧縮の実装	299	3,738
lzss [*3]	LZSS 圧縮の実装	1,215	9,692
mini-db [*4]	単純なデータベースエンジンの実装	577	6,546
small-search-engine [*5]	生徒情報管理システムの実装	336	4,868

表 2　BlueScript ベンチマークプログラム (2)
Table 2　BlueScript benchmark programs (2).

ベンチマーク名	LOC	コンパイル後のバイナリサイズ [B]
bounce	106	1,118
list	91	1,106
mandelbrot	95	753
nbody	201	2,490
permute	55	557
queens	73	717
sieve	50	433
storage	69	750
towers	97	1,180

表 3　BlueScript ベンチマークプログラム (3)
Table 3　BlueScript benchmark programs (3).

ベンチマーク名	LOC	コンパイル後のバイナリサイズ [B]
crc	125	5,813
sha	43	9,120
fft	67	17,228
fir	119	25,385
iir	189	22,124

をそのまま使って圧縮転送する処理系も作成した．

4.1　SRAM 消費量の比較

まず，提案手法を MINIZ に実装しない場合と実装する場合における解凍時の SRAM 消費量を比較する．MINIZ の解凍器では，必要な SRAM 上の領域を構造体として一括管理しているため，この構造体のサイズを計測して得られたそれぞれの SRAM 消費量を**表 4** に示す．MINIZ による SRAM 消費量は，提案手法を実装しない場合には 41.2 KB であったが，提案手法を実装した場合には 0.124 KB であった．このことから，提案手法によって SRAM の消費量を 99.7%削減できたと考えられる．これだけ大きな SRAM 消費量の削減を達成できた要因は，32.8 KB，8.30 KB と

[*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

表 4　MINIZ と提案手法における SRAM 消費量 [KB]
Table 4　SRAM consumption of MINIZ and proposal [KB].

	LZ77 辞書	ハフマン表	その他	合計
MINIZ	32.8	8.30	0.104	41.2
提案手法	0	0	0.124	0.124

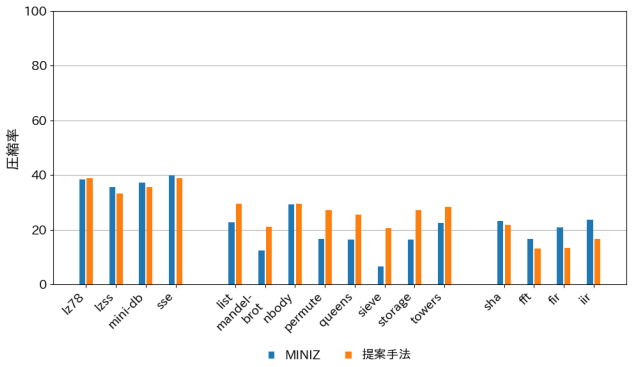


図 2　MINIZ とリセット前の提案手法の圧縮率の比較
Fig. 2　Compression ratio of MINIZ and proposal before reset.

SRAM 上で大きな容量を占めていた LZ77 辞書とハフマン表が，それぞれ手法 1 と手法 2 によって SRAM を消費しなくなったことであると考えられる．

4.2　既存手法とリセット前の提案手法の性能の比較

次に表 1，表 2，表 3 のプログラムについて，既存手法である MINIZ をそのまま使用する BlueScript と提案手法に基づいた圧縮を行う BlueScript を比較する．それぞれで実行した際の圧縮効率，解凍時間，転送時間を測定した．ここで圧縮効率は次の式 (1) によって計算される圧縮率 CR で測定し，転送時間は圧縮，通信，解凍の時間を含む．また，用意したプログラムのうち，lz77，bounce，crc は静的ハフマン表を事前作成するために使用しており，測定対象から外した．

$$CR = \frac{Size_{uncompressed} - Size_{compressed}}{Size_{uncompressed}} \quad (1)$$

図 2 から，提案手法により圧縮率は最善の場合 14 ポイント（sieve）上昇しており，最悪で 7 ポイント程度（fir）低下している．また圧縮率が最も向上したのは sieve, storage, permute, queens である一方で，最も低下したのは fir, biquad，fft，lzss であり，これらはいずれもコンパイル後

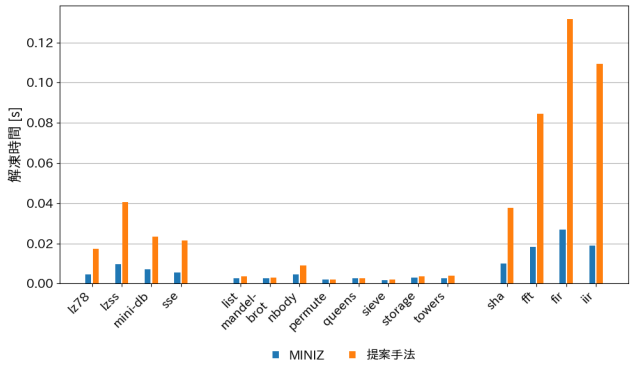


図 3　MINIZ とリセット前の提案手法の解凍時間の比較

Fig. 3　Decompression time of MINIZ and proposal before reset.

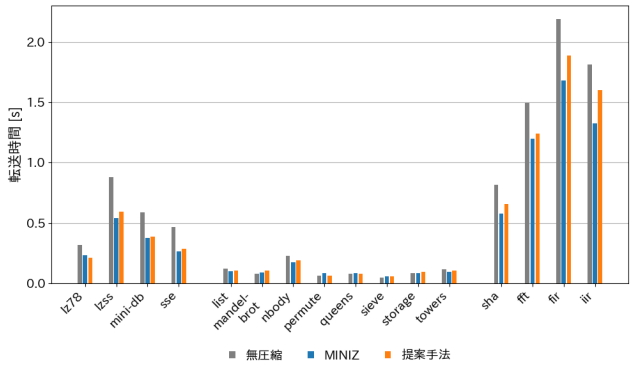


図 4　MINIZ とリセット前の提案手法の転送時間の比較

Fig. 4　Transfer time of MINIZ and proposal before reset.

Listing 1: ソフトリセットに挿入するコード

```
1  function foo() {
2      const arr = new Array<integer>(5);
3      for (let i = 0; i < arr.length; i++)
4          arr[i] = i * 2;
5      let sum: integer = 0;
6      for (let i = 0; i < arr.length; i++)
7          sum += arr[i];
8      let max: integer = 0;
9      for (let i = 0; i < arr.length; i++)
10         if (arr[i] > max)
11             max = arr[i];
12 }
```

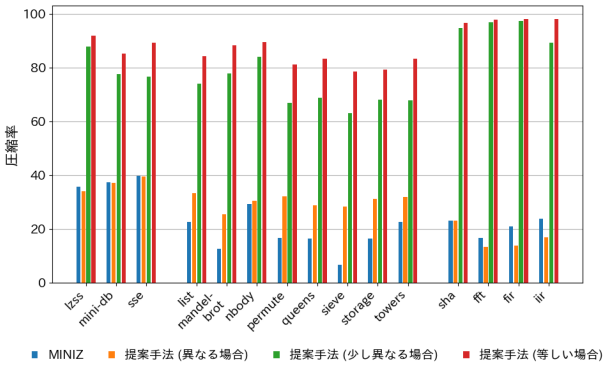


図 5　MINIZ とリセット後の提案手法の圧縮率の比較

Fig. 5　Compression ratio of MINIZ and proposal after reset.

のコードサイズが最小・最大のプログラムである．既存手法では圧縮のたびに作成されるハフマン表で圧縮し，それをヘッダに含めるが，静的ハフマン表では事前作成されたハフマン表で圧縮し，ヘッダには対応するインデックスのみを書き込むため，既存手法よりもハフマン符号化時の圧縮性能は下がり，ヘッダサイズは小さくなる．ネイティブコードが小さい場合，全体に対するヘッダサイズの割合が大きいため，ヘッダサイズの減少による圧縮率への寄与が大きいが，大きい場合，ヘッダサイズの減少による圧縮率への寄与が小さくなり，ハフマン符号化時の圧縮性能低下による圧縮率への影響が大きくなる．

解凍時間でも同様に permute，sieve，queens，storage の順で最も悪化せず，biquad，fir，fft，lzss の順で最も悪化する．既存手法では SRAM 上でハフマン表を再構築してから解凍するが，提案手法ではフラッシュメモリ上のハフマン表でそのまま解凍する．小さいデータを解凍する際，既存手法では解凍処理全体に対する表の再構築によるオーバヘッドの割合が大きくなり，提案手法ではフラッシュメモリアクセスのオーバヘッドの割合が小さくなるため，解凍時間の差は縮まるが，解凍するデータが大きくなるにつれ，フラッシュメモリアクセスのオーバヘッドがより強く解凍時間に反映される．最後に転送時間においては，queens と lz78 以外のすべてのプログラムで提案手法は既存手法から悪化しているが，最悪の場合でも 2 割程度（biquad）の悪化にとどまっていることが分かる．

4.3　リセット後の提案手法の性能の向上

リセット前のコードの再利用の効果を確かめるために，

(1) ソフトリセット前後で実行されるプログラムが異なる場合，

(2) ソフトリセット前後で実行されるプログラムが少し異なる場合，

(3) ソフトリセット前後で実行されるプログラムが等しい場合

の 3 つのシナリオを用意し，提案手法に基づく圧縮を行う BlueScript を用いてそれぞれ測定を行った．

(1) の異なる場合のシナリオでは，ソフトリセット前では lz78 ベンチマークプログラムを実行し，ソフトリセット後にはそれ以外を実行する．(2) の少し異なる場合のシナリオでは，ソフトリセット前に各ベンチマークプログラムを，ソフトリセット後に次の Listing 1 を挿入したプログラムを実行する．MINIZ をそのまま利用する BlueScript における圧縮率，解凍時間，転送時間と，提案手法に基づく BlueScript のソフトリセット後の圧縮率，解凍時間，転送時間の測定結果をそれぞれ**図 5**，**図 6**，**図 7** に示す．

まず (1) のソフトリセット前後で実行するプログラムが

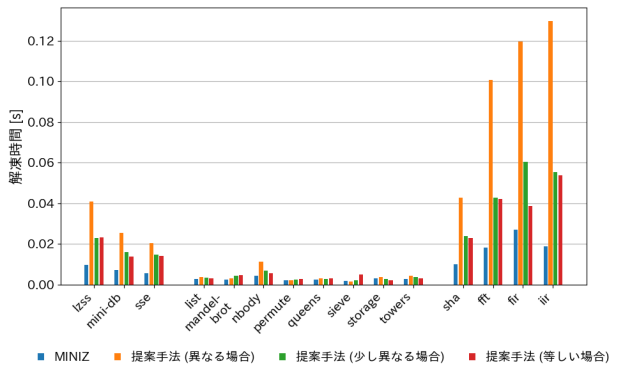


図 6　MINIZ とリセット後の提案手法の解凍時間の比較
Fig. 6　Decompression time of MINIZ and proposal after reset.

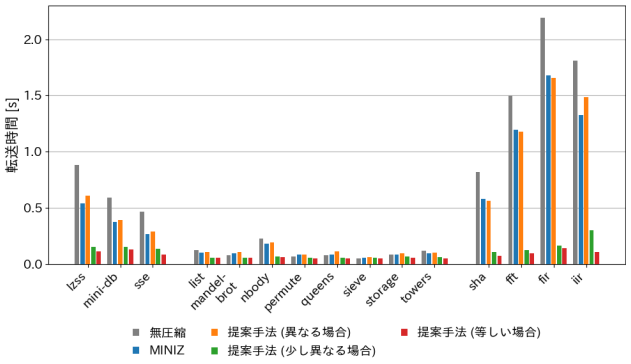


図 7　MINIZ とリセット後の提案手法の転送時間の比較
Fig. 7　Transfer time of MINIZ and proposal after reset.

異なる場合における提案手法の性能について述べる．図 5 より 4.2 節の実験同様，提案手法によって圧縮率が既存手法から最も改善したのは sieve，最も改善しなかったのは fir である．また，図 2 と図 5 のリセット前後の提案手法の圧縮率を比較すると，リセット後はリセット前に比べ最大 8 ポイント程度（sieve）上昇し，最小でほとんど変化しない（fir）ことが分かる．コードサイズが近い list と nobody の圧縮率を比較するとそれぞれ 4 ポイントと 1 ポイントである．このような差が生じているのは，fir や nbody のコードでは lz78 のコードに存在しない数値の代入が多く存在し，LZ77 圧縮における参照の探索に引っかからないためである．

解凍時間においては図 6 より 4.2 節の実験同様，コードサイズと解凍時間に相関関係が存在しており，最悪の場合で 7 倍程度（iir）悪化している．**図 3** と図 6 を比較すると，解凍時間はリセット前に比べ全体的に増加傾向であり，最大で 2 割程度（nbody）悪化している．リセット後の解凍時間が伸びるのは，LZ77 解凍時に一部フラッシュメモリ上のリセット前のコードを読み出すことによるオーバヘッドが加わるためである．一方で sieve のように 3 割ほど改善している場合もあるのは，圧縮率の向上が向上することによって解凍時間が短縮したためだと考えられる．転送時間に関しては図 7 より，既存手法から最悪の場合で 3 割ほど（queens）悪化し，最善の場合でほとんど変化しない（sha256）．

最後に，ソフトリセット前後でプログラムが少し異なる場合と等しい場合における提案手法の性能について述べる．図 5 より圧縮率では少し異なる場合で 37 ポイント程度以上，等しい場合で 50 ポイント程度以上向上している．両者において最も向上したのはともに fft であり，これは元々既存手法での圧縮率が低いことに加え，提案手法によるリセット後の圧縮率が非常に高いことで，大きな差が生じたためである．解凍時間に関して，既存手法から最大 iir で 3 倍程度悪化している．一方で，iir の少し異なる場合と等しい場合の解凍時間は，異なる場合よりも 2 倍程度良い性能を示しているが，これも圧縮率向上によるものである．最後に，少し異なる場合と等しい場合の提案手法は，既存手法から転送時間が最大で 13 倍程度（fft），最小で 1 割程度（sieve）改善している．

5.　関連研究

ZStandard は高圧縮ライブラリの 1 つであり，パラメータを変化させることで圧縮時間が伸びる代わりに ZIP よりも高い圧縮効率を達成できる．また，ZStandard ではビデオ転送などを行う際のストリーム圧縮・解凍において，解凍データの出力先を stable にすることで，追加の領域を確保する代わりに解凍データの出力先をそのまま LZ77 辞書として利用することができる．本論文では対話環境で次々と送られるコードもストリームとして見なせば同じテクニックが使えることに着目し，提案手法内のコードの LZ77 辞書化の手法としている．一方で，ZStandard は辞書以外での SRAM の消費が大きすぎるため，そのままでのマイコン上での利用は困難である．

マイコンへのコード転送の高速化はこれまで活発に研究がなされてきた．Contiki OS [9] は IoT 向けに開発された軽量な OS であり，アップデートの際に変更箇所の再配置可能コードのみを転送し，マイコン上で動的リンクすることにより，転送時間を削減する．一方で本論文では BlueScript がホストマシン上でリンクするため，シンボルテーブルと再配置テーブルを転送データに含める必要がない点と，ネイティブコードを圧縮し転送サイズをさらに削減する点がこの研究と異なる．

Zephyr [10] や R3 [11] は，無線センサネットワークでファームウェアをバージョンアップする際に利用される転送プロトコルである．両者ともに，新しいファームウェアが現行のものとネィティブコードの構造が相似するように前処理を実行してから，差分を計算し転送する．これらの研究が対象としているファームウェアは異なるバージョン間でもネィティブコードの大半が共通するため，差分アルゴリズムによって効率的に転送サイズを減らしている．本論文で提案する手法のうちリセット前のコードの再利用の部分は，新旧のネィティブコードが似ていることを利用し

ている点で，これらの研究と考え方が似ている．しかし本論文では単純な差分アルゴリズムによって転送するデータ量を削減するのではなく，ZIP 圧縮を用いている点が異なる．

また，プログラム格納用のメモリにかかわる製造コストを抑えるためにできるだけマイコンのネイティブコードの大きさを小さくする研究も存在する．Debray らはコードの大きさを削減する目的で，リンク後のコードに対する最適化をホストマシン上で実施する Squeeze [12] というバイナリ書き換えツールを開発した．このツールはコード内で繰り返される処理を関数に抽象化することでサイズを削減するため，マイコンに転送されるとそのままロード・実行が可能である．しかし本論文では，転送速度が対話性のボトルネックとなるような環境を対象としているため，最適化処理よりもより転送サイズを削減できる圧縮アルゴリズムに基づく方法を提案している．

6. まとめ

本論文では少ない SRAM 消費で ZIP 圧縮されたネイティブコードをマイコン上で解凍するために，コードの LZ77 辞書化，静的ハフマン表，リセット前のコードの再利用，という 3 つの手法を組み合わせる ZIP 解凍器の実装方法を提案した．これにより SRAM 上のネイティブコードの配置先の領域を圧迫することなく，BLE の通信時間を短縮し，ライブラリなど大きいコードを転送する際に発生する対話性の低下を改善することができる．

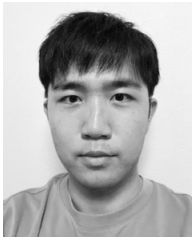
提案手法を評価するために，提案手法を実装した MINIZ と，既存手法であるそのままの MINIZ をそれぞれ使って圧縮転送する 2 つのシステムを用意し，大きさの異なる様々なプログラムを実行した際のリセット前後の圧縮率，解凍時間，転送時間を測定した．またリセット後の性能を評価するために，リセット前後のコードが異なる場合，少し異なる場合，等しい場合の 3 つのシナリオに分けて測定した．リセット前では，提案手法はコードの大きさが大きくなるにつれ圧縮率と解凍時間が既存手法より悪化していくが，転送時間は最大で 1.21 倍の悪化に止まる．異なる場合におけるリセット後では，提案手法の性能はリセット前と同程度であり，少し異なる場合と等しい場合では，解凍時間はリセット前から大幅に改善され，圧縮率は既存手法から最大 48.0 ポイント改善し，転送時間で 12.8 倍改善する．また提案手法は既存手法に比べ SRAM 消費量を 99.7%削減する．

我々は BLE 通信による長い応答時間を改善するための圧縮手法を提案したが，ネイティブコードの転送時間を短縮する他の方法として，Wi-Fi などのより高速な通信方法を併用するという方法も考えられる．たとえばライブラリコードのような大きなデータを転送する場合にのみより高速な通信に切り替えることで，消費電力への影響を抑えつつ転送時間を短縮することが可能であろうと思われるが，これは今後の課題とする．

謝辞　本研究は JSPS 科研費 JP20H00578，JP24H00688 の助成を受けたものです．

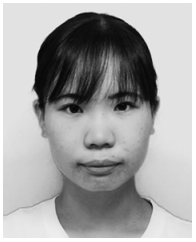
参考文献

[1] csg tokyo: BlueScript, csg-tokyo (online), available from ⟨https://github.com/csg-tokyo/bluescript⟩ (accessed 2025-02-16).
[2] George, D.: MicroPython, George Robotic Limited (online), available from ⟨https://micropython.org/⟩ (accessed 2025-02-16).
[3] Collet, Y. and Kucherawy, M.: Zstandard Compression and the 'application/zstd' Media Type, RFC 8878 (2021).
[4] Ziv, J. and Lempel, A.: A universal algorithm for sequential data compression, *IEEE Trans. Information Theory*, Vol.23, No.3, pp.337–343 (online), DOI: 10.1109/TIT.1977.1055714 (1977).
[5] Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes, *Proc. IRE*, Vol.40, No.9, pp.1098–1101 (online), DOI: 10.1109/JRPROC.1952.273898 (1952).
[6] richgel999: MINIZ, richgel999 (online), available from ⟨https://github.com/richgel999/miniz⟩ (accessed 2025-02-16).
[7] Marr, S., Daloze, B. and Mössenböck, H.: Cross-language compiler benchmarking: Are we fast yet?, *Proc. 12th Symposium on Dynamic Languages*, *DLS 2016*, pp.120–131, Association for Computing Machinery (online), DOI: 10.1145/2989225.2989232 (2016).
[8] Plauska, I., Liutkevicius, A. and Janaviciute, A.: Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller, *Electronics* (*Switzerland*), Vol.12, No.1, p.143 (online), DOI: 10.3390/electronics12010143 (2022).
[9] Dunkels, A., Gronvall, B. and Voigt, T.: Contiki - A lightweight and flexible operating system for tiny networked sensors, *29th Annual IEEE International Conference on Local Computer Networks*, pp.455–462 (online), DOI: 10.1109/LCN.2004.38 (2004).
[10] Panta, R.K., Bagchi, S. and Midkiff, S.P.: Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation, *Proc. 2009 Conference on USENIX Annual Technical Conference*, *USENIX '09*, p.32, USENIX Association (2009).
[11] Dong, W., Mo, B., Huang, C., Liu, Y. and Chen, C.: R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems, *2013 Proc. IEEE INFOCOM*, pp.315–319 (online), DOI: 10.1109/INFCOM.2013.6566786 (2013).
[12] Debray, S.K., Evans, W., Muth, R. and De Sutter, B.: Compiler techniques for code compaction, *ACM Trans. Program. Lang. Syst.*, Vol.22, No.2, pp.378–415 (online), DOI: 10.1145/349214.349233 (2000).



渡邉 純一

2025 年東京大学大学院情報理工学系研究科創造情報学専攻修士課程修了．同校ではホストマシン・マイコン間のコード転送の高速化に関する研究に従事．



望月 文香

2024 年東京大学大学院情報理工学系研究科創造情報学専攻修士課程修了．現在，同専攻博士後期課程に在籍している．マイクロコントローラ向けの言語処理系の研究に従事．

山崎 徹郎　（正会員）

2021 年東京大学大学院情報理工学系研究科博士課程修了，博士（情報理工学）．現在，同研究科助教．メタプログラミングの応用，FFI 環境下でのガベージコレクションの研究に従事．

千葉 滋　（正会員）

1991 年東京大学理学部情報科学科卒業，1996 年同大学大学院理学系研究科情報科学専攻より博士（理学）．2012 年東京大学情報理工学系研究科創造情報学専攻教授．プログラミング言語，システムソフトウェアの研究に従事．