

# 組込マイコン向けCコンパイラにおける ストアの融合とループ化の実装

千葉 雄司<sup>1</sup> 平下 敬之<sup>2</sup>

受付日 2019年5月19日, 採録日 2019年9月26日

**概要:** 組込アプリケーションの初期化では, しばしば連続する記憶領域に定数をストアするが, この処理がコードサイズの増大の原因になることがある. 当該処理のサイズを削減する最適化に, ストアの融合やループ化があるが, その実装方法, たとえば併用方法や, 組込アプリケーションに与える効果は必ずしも定かでない. そこで本論文では, ストアの融合やループ化の併用方法と, 現実的な組込アプリケーションのコードサイズに与える影響を明らかにする. 評価の結果, ストアの融合とループ化によってコードサイズを最大 8.52%削減できることが分かった.

**キーワード:** コンパイラ, 最適化, コードサイズ, 組込機器

## An Implementation of Store Coalescing and Transformation to Loop by a C Compiler for Embedded Microcontrollers

YUJI CHIBA<sup>1</sup> TAKAYUKI HIRASHITA<sup>2</sup>

Received: May 19, 2019, Accepted: September 26, 2019

**Abstract:** An embedded application often stores constant values to continuous memory area, and the store instruction sequence sometimes bloats the code size. Traditional optimization such as store coalescing and transformation to loop reduces the code size, but their implementation method, such as combined use, or effect to embedded applications are not entirely apparent. This paper describes combined use of these optimizations and the effect of these optimizations to practical embedded applications. Our evaluation showed store coalescing and transformation to loop reduce practical application code size by 8.52% at most.

**Keywords:** compiler, optimization, code size, embedded device

### 1. はじめに

近年, パーソナルコンピュータや携帯電話といったデバイスは著しくその性能を高め, 非常に大きな記憶容量を持つことが一般的になっているが, 組込機器には, コストの要求に対応するため, 大きな記憶領域を持たないものもある. 各ベンダが製造している組込マイコンの内蔵フラッシュメモリの容量は, 最も小さなものでは, 8/16 bit マイコンで 0.5 から 2 KByte [1], [2], [3], [4], [5], [6], [7], 32 bit マイコンでも 4 から 8 KByte [8], [9], [10], [11] であり, こうした組込マイコン向けコンパイラにとっては, コードサ

イズの削減が今なお大きな課題でありつづけている.

本論文では組込アプリケーションの, 特に初期化処理に類出する命令列である, 連続領域への定数のストアに有効な最適化について述べる. 具体的には, ストアの命令列を融合 [12], [13] あるいはループ化する最適化について, その実装方法の詳細, 特に組合せ方を明らかにし, また, それらの最適化がコードサイズにもたらす影響を, 実践的な組込アプリケーションによって評価した結果を示す. 本論文で示す評価結果はルネサスエレクトロニクスのコンパイラ CC-RL/CC-RX [14], [15] によるものだが, 本論文で示す実装方法はマイコンやコンパイラによらず適用できる.

本論文の構成について述べる. 本論文では, まず 2 章において, 本論文における最適化の対象である, 連続領域への定数のストアについて, その発生源となるソースコードの記述例を示す. 次に, 3 章で, 当該ストアのサイズ削減に

<sup>1</sup> 株式会社日立製作所  
Hitachi, Ltd., Kokubunji, Tokyo 185-8601, Japan

<sup>2</sup> ルネサスエレクトロニクス株式会社  
Renesas Electronics Corporation, Kodaira, Tokyo 187-8858, Japan

有効な最適化である、ストアの融合とループ化について述べる。4章でストアの融合とループ化の実装方法について詳述し、5章でストアの融合とループ化の効果を明らかにする。6章で関連研究を示す。最後に7章で結論を述べる。

## 2. 連続領域への定数のストア

本論文で最適化の対象とするストアの命令列について、その発生源となるソースコードの記述例を図1(a)に示す。図1(a)では構造体の個々のメンバに順に定数を代入している。図1(a)の記述は、文字どおりに解釈するならば、個々のフィールドごとに定数をストアすることをコンパイラに要求するものである。したがって、少なくとも、最適化せずにコンパイルすると図1(b)の命令列になる。なお、図1(b)の命令列をはじめ、本論文で示すアセンブリコードはすべて、ルネサスエレクトロニクス社のRXマイコン[9]のものとする。

図1(a)の記述は、代入先が構造体全体である場合、代入処理を図1(c)のようにも記述できる。図1(c)の記述は、構造体のどのフィールドにどの順番でどのビット幅でストアするか規定せず、その判断をコンパイラに要求する。図1(c)のコンパイル結果は、図1(b)に示した、フィールドごとに定数をストアするコードにも、図1(d)の疑似コードに示す、標準ライブラリ関数 `memcpy()` を使って構造体に値を代入するコードにもできる。そこでコンパイラは、最適化を行うならば、こういったコードにするのが適切か判断することになる。たとえばユーザの要求がコードサイズを小さくすることであれば、図1(b)と図1(d)のどちらにする方がコードサイズを小さくできるか見積もり、小さくなると判断した方にコンパイルする。

図1(c)の記述はコンパイラに最適化を要求するものとも解釈できるが、最適化を重視するプログラマであっても、ソースコードを図1(c)のように記述せず、図1(a)のように記述することがある。その理由の1つは、図1(a)の記述の方が、図1(c)の記述よりも、どのフィールドにどんな値を代入しているかを読み取りやすいからである。読み取りやすさの問題は、図1(e)に示すように、初期化に複合リテラルを使うことでも解決できる。複合リテラルを使えば、代入するリテラルの中にフィールド名を記述可能になることに加え、リテラル中のフィールドの記述順序もプログラマの思いどおりにできる。ただし、複合リテラルが比較的新しい機能で、広くサポートされているC言語の規格ANSI X3.159-1989にないこと、また、複合リテラルでは構造体を部分的に初期化したいという要求や、フィールドの初期化順を制御したいといった要求に対応できないといったこともあって、図1(a)の記述で構造体を初期化するソースコードがあり、このため最適化Cコンパイラの実装に際しては、図1(a)のような、連続する記憶領域に定数をストアするソースコード向けの最適化が必要になりうる。

```

1: #define OFF 0
2: #define ON 1
3: typedef struct{
4:     _Bool _switch0;
5:     _Bool _switch1;
6:     ...
7:     _Bool _switchN;
8: } GlobalVariableType;
9: GlobalVariableType GlobalVariable;
10: void init(void){
11:     GlobalVariable._switch0 = OFF;
12:     GlobalVariable._switch1 = ON;
13:     ...
14:     GlobalVariable._switchN = OFF;
15: }
```

(a) フィールドごとに初期化するソースコード

```

16: _init:
17:     MOV.L #_GlobalVariable, R1
18:     MOV.B #00H, [R1]
19:     MOV.B #01H, 01H[R1]
20:     ...
21:     MOV.B #00H, NH[R1]
22:     RTS
```

(b) フィールドごとに初期化するアセンブリコード

```

23: void init(void){
24:     GlobalVariable =
25:         { OFF, ON, ..., OFF };
26: }
```

(c) 構造体全体を初期化するソースコード

```

27: #include<string.h>
28: void init(void){
29:     static const _Bool
30:         initialValues[] = { OFF, ON, ..., OFF };
31:     memcpy(&GlobalVariable, initialValues,
32:           sizeof(GlobalVariable));
33: }
```

(d) 構造体全体を初期化する疑似コード

```

34: void init(void){
35:     GlobalVariable = (GlobalVariableType){
36:         ..switch1 = ON;
37:         ...
38:         ..switchN = OFF;
39:         ..switch0 = OFF;
40:     };
41: }
```

(e) 複合リテラルで構造体全体を初期化するソースコード

図1 構造体の初期化

Fig. 1 Initialization of a structure.

### 3. ストアの融合とループ化

連続領域に定数をストアする命令列のサイズ削減に有効な既存の最適化に、ストアの融合とループ化がある。本章ではこれら既存の最適化について述べる。

#### 3.1 ストアの融合

ストアの融合は、アーキテクチャが提供するストア命令のうち、ストアする値のビット幅が最大でないものが複数あるとき、それらをまとめて1つのストア命令にする最適化である。ストアの融合によると、たとえば図 2(a) に示したソースコードのコンパイル結果を図 2(b) から図 2(c) に最適化できる。

ストアの融合の対象となるストア命令は、次の条件を満たす必要がある。

- (1) 連続する記憶領域にストアすること
- (2) ユーザがストアの最適化を許容すること
- (3) ストアする値が定数であること
- (4) 最適化後のストア命令が境界調整 (alignment) の条件を満たすこと

ここで(2)の条件を満たすために必要なことは、ユーザが融合対象のストア先を2つ以上、volatileで修飾しないことである。volatileはC言語が提供する修飾子であり、修飾対象へのロードやストアをソースコードの記述どおりに実行することを要求する。したがって、volatileで修飾した対象へのストアどうしはストアの融合の対象にしてはならない。なぜなら、volatileで修飾した対象へのストアどうしを融合すると、それらの実行順序が同時、すなわちソースコードの記述どおりでなくなるからである。なお、融合対象のストアの1つのみ、ストア先にvolatileの修飾があるとき、それを他のストアと融合し、ストアのアクセス幅を変更することを許容するかどうかは処理系依存である。

(3)の条件を満たす必要がある理由は、定数でない値をストアする場合、ストアする値を作成する命令列が必要になり、命令数を増やしかねないためである。たとえば図 3(a)の、定数でない値のストアの命令列を融合した結果は図 3(b)に示すとおりで、図 3(a)の命令列より命令数が増えている。

(4)の条件は、境界調整の条件を持つマイコン、たとえばRL78マイコン[5]向けのストアの融合を難しくする。ここで境界調整の条件とは、ストアなどメモリアクセスを行う命令を実行する際にアクセス先の番地が満たすべき条件であり、RL78マイコンは、アクセス先の番地がアクセスするデータ幅の倍数であることを要求する。したがって、1バイトのデータにアクセスする場合は、アクセス先の番地は1の倍数、すなわち任意の値にできるが、2バイトのデータにアクセスする場合は、アクセス先の番地を2の倍数に

```

1:  struct{
2:     char _charField0;
3:     char _charField1;
4:  } GlobalPair;
5:
6:  void initializePair(void){
7:     GlobalPair._charField0 = 0x01;
8:     GlobalPair._charField1 = 0x23;
9:  }
    
```

(a) ソースコード

```

10: MOV.L #_GlobalPair, R1
11: MOV.B #01H, [R1]
12: MOV.B #23H, 02H[R1]
    
```

(b) コンパイル結果 (ストアの融合なし)

```

13: MOV.L #_GlobalPair, R1
14: MOV.W #2301H, [R1]
    
```

(c) コンパイル結果 (ストアの融合あり)

図 2 ストアの融合

Fig. 2 Store coalescing.

```

1:  MOV.B R2, [R1]
2:  MOV.B R3, 01H[R1]
    
```

(a) ストアの融合なし

```

3:  SHLL #8, R3, R4
4:  MOVU.B R3, R5 ; 上位 24 ビットをゼロに
5:  OR R4, R5
6:  MOV.W R5, [R1]
    
```

(b) ストアの融合あり

図 3 定数以外のストアの融合

Fig. 3 Non-constant value store coalescing.

する必要がある。この条件は、たとえば図 2(a)のソースコードの7行目と8行目にある1バイトのデータのストアの融合を難しくする。なぜなら、これらのストアを融合して2バイトのストアに融合するには、構造体GlobalPairの先頭アドレスを2の倍数にする必要があるが、構造体GlobalPairの型から先頭アドレスの境界調整の条件を求めると、構造体GlobalPairのメンバの境界調整の条件がいずれも1の倍数であることから、構造体全体の境界調整の条件も1の倍数にしかならず、したがって構造体GlobalPairの先頭アドレスが2の倍数になることを、構造体の型がもたらす条件から保証できないからである。

RL78マイコン向けに7行目と8行目のストアを融合するには、型がもたらす条件とは別の形でアクセス先の番地を2の倍数にする必要がある。図 2(a)のソースコードに限れば、アクセス先が同一ファイルで定義する大域変数だと分かるので、当該大域変数の先頭番地を2の倍数に変更すればよく、それに限れば容易に実現できるが、大域変数の定義が別ファイルにある場合、ファイルをまたぐ最適化

```

1:  unsigned char GlobalArray[4];
2:  void clearArray(void){
3:      GlobalArray[0] = 0;
4:      GlobalArray[1] = 0;
5:      GlobalArray[2] = 0;
6:      GlobalArray[3] = 0;
7:  }

```

(a) ソースコード

```

8:  MOV.L #_GlobalArray, R1
9:  MOV.B #00H, [R1]
10: MOV.B #00H, 01H[R1]
11: MOV.B #00H, 02H[R1]
12: MOV.B #00H, 03H[R1]

```

(b) コンパイル結果 (ループ化なし)

```

13: MOV.L #_GlobalArray, R1
14: MOV.L #0, R2
15: MOV.L #4, R3
16: LOOP:
17: MOV.B R2, [R1+]
18: SUB #1, R3
19: BNZ LOOP

```

(c) コンパイル結果 (ループ化あり)

```

20: MOV.L #_GlobalArray, R1
21: MOV.L #0, R2
22: MOV.L #4, R3
23: SSTR.B

```

(d) コンパイル結果 (RX マイコン固有命令あり)

図 4 同じ定数をストアする命令列のループ化

Fig. 4 Transformation to loop for identical store values.

の実装が必要になり、実装にかかる手間が大きくなる。またストア先がどのデータ構造か分からない場合には、調整する手段がないので融合をあきらめるよりない。境界調整の条件がストアの融合の適用範囲に与える影響については5章で評価する。

### 3.2 ループ化

ループ化はストア命令の列をループにまとめる最適化であり、既存の最適化である loop rerolling [16], [17] で実現できることもある。ループ化には、ストアする定数が同じ場合向けのものと、異なる場合向けのものの2種類がある。

#### 3.2.1 同じ定数をストアする命令列のループ化

同じ定数をストアする命令列のループ化は、たとえば図 4(a) のソースコードのコンパイル結果を図 4(b) の命令列から図 4(c) の命令列に最適化する。ここで RX マイコンには図 4(c) の 16 から 19 行目を実現する命令 SSTR ががあるので、それを使うとさらに図 4(d) の命令列にできる。

ここでループを使う図 4(c) の命令列は、使わない図 4(d) の命令列に比べて命令数が多いため、より多くのストア命

```

1:  struct{
2:      unsigend short _fieldS1;
3:      unsigend short _fieldS2;
4:      unsigend short _fieldS3;
5:      unsigend char _fieldC1;
6:      unsigend char _fieldC2;
7:  } GlobalStruct;
8:
9:  void initializeStruct(void){
10:     GlobalStruct._fieldS1 = 0x1010;
11:     GlobalStruct._fieldS2 = 0x1010;
12:     GlobalStruct._fieldS3 = 0x1010;
13:     GlobalStruct._fieldC1 = 0x10;
14:     GlobalStruct._fieldC2 = 0x10;
15:  }

```

(a) ソースコード

```

16: MOV.L #_GlobalStruct, R1
17: MOV.W #1010H, [R1]
18: MOV.W #1010H, 02H[R1]
19: MOV.W #1010H, 04H[R1]
20: MOV.B #10H, 06H[R1]
21: MOV.B #10H, 07H[R1]

```

(b) コンパイル結果 (ループ化なし)

```

22: MOV.L #_GlobalStruct, R1
23: MOV.L #10H, R2
24: MOV.L #8, R3
25: SSTR.B ; 1 バイトの値 0x10 を 8 回ストアする

```

(c) 1 バイトごとにストアするループ化

```

26: MOV.L #_GlobalStruct, R1
27: MOV.L #1010H, R2
28: MOV.L #4, R3
29: SSTR.W ; 2 バイトの値 0x1010 を 4 回ストアする

```

(d) 2 バイトごとにストアするループ化

図 5 ビット幅の異なるストアのループ化

Fig. 5 Transformation to loop for different bit-width stores.

令をまとめないとコードサイズを削減できないことに注意が必要である。また図 4(c), (d) の命令列は、ともに、図 4(b) の命令列に比べて実行効率に優れるとは限らない点にも注意が必要である。

なお、同じ定数をストアする命令列のループ化は、ストアする定数を同じにできれば、ビット幅が異なるストアにも適用できる。たとえば図 5(a) のソースコードは 16 ビットの値 3 つと 8 ビットの値 2 つをストアするが、ストアする定数を構成するバイトはどれも 0x10 なので、図 5(c) のコードに最適化できる。あるいは図 5(a) のソースコードは 16 ビットの値 0x1010 を 4 つストアすると考えることも可能で、そう考えれば図 5(d) のコードにも最適化できる。ただし、RX マイコンのように、ストアする定数の大きさに命令長が比例するアーキテクチャでは、定数の小

```

1: unsigned char GlobalArray[4];
2: void initializeArray(void){
3:     GlobalArray[0] = 0;
4:     GlobalArray[1] = 3;
5:     GlobalArray[2] = 1;
6:     GlobalArray[3] = 2;
7: }

```

(a) ソースコード

```

8:     MOV.L #_GlobalArray, R1
9:     MOV.B #00H, [R1]
10:    MOV.B #03H, 01H[R1]
11:    MOV.B #01H, 02H[R1]
12:    MOV.B #02H, 03H[R1]

```

(b) コンパイル結果 (ループ化なし)

```

13:    MOV.L #_GlobalArray, R1
14:    MOV.L #_ConstantValues, R2
15:    MOV.L #5, R3
16:    LOOP:
17:    MOV.B [R2+], R4
18:    MOV.B R4, [R1+]
19:    SUB #1, R3
20:    BNZ LOOP
21:    cdots
22:    _ConstantValues:
23:    .db 00H, 03H, 01H, 02H

```

(c) コンパイル結果 (ループ化あり)

```

24:    MOV.L #_GlobalArray, R1
25:    MOV.L #_ConstantValues, R2
26:    MOV.L #5, R3
27:    SMOVF
28:    cdots
29:    _ConstantValues:
30:    .db 00H, 03H, 01H, 02H

```

(d) コンパイル結果 (RX マイコン固有命令あり)

図 6 異なる値をストアする命令列のループ化

Fig. 6 Transformation to loop for different store values.

い図 5(c) のコードの方が、小さなサイズになる。

図 5(d) のコードはサイズでは図 5(c) のコードに劣り、実行サイクル数では、ループ化しない図 5(b) のコードに劣る。この図 5(d) のコードが有用になるのは、ユーザが 16 ビットのストアのアクセス幅の変更を禁止した場合である。修飾子 `volatile` がアクセス幅の変更の禁止を意味する処理系では、図 5(a) のソースコード中の 16 ビットのメンバ変数を `volatile` で修飾すると、図 5(c) のコードに最適化できなくなるが、このとき 8 ビットのメンバ変数を `volatile` で修飾していないなら、図 5(d) のコードには最適化できる。

### 3.2.2 異なる値をストアする命令列のループ化

異なる定数値をストアする命令列のループ化は、たとえ

ば図 6 (a) のコンパイル結果を図 6 (b) の命令列から図 6 (c) の命令列に最適化する。ここで RX マイコンには図 4 (c) の 16 から 20 行目を実現する命令 `SMOVF` があるので、それを使うとさらに図 4 (d) の命令列にできる。

同じ定数をストアする命令列のループ化と、異なる定数をストアする命令列のループ化の違いは、後者がストアする定数を配列に保持し、ループを周回するたびに、ストアする定数を配列からロードすることにある。定数を保持する配列のサイズは、ストアする定数の数に比例して大きくなるので、異なる定数をストアする命令列のループ化がもたらすコードサイズの削減幅は、同じ定数をストアする命令列のループ化がもたらす削減幅ほど大きくない。アーキテクチャによっては、配列に格納する定数のサイズが、定数をストアする命令のサイズを上回ることもある。実際、RX マイコンでは、32 ビット、すなわち 4 バイトの定数をストアする最小の命令のサイズが 3 バイトであり、ストアする定数より小さい。このような命令を多く含む命令列に、異なる定数をストアする命令列のループ化を適用すると、かえってコードサイズを増やすので注意を要する。

なお、ループ部分のコード、たとえば図 4 (c) の 16 から 19 行目や図 6 (c) の 16 から 20 行目の部分のコードは、ある程度の大きさになるが、それが問題になる場合には、たとえば図 1 (c) のようなサブルーチン呼出しにしたり、あるいは `procedural abstraction` [18], [19], [20], [21], [22] という最適化によってまとめることができる。

## 4. 実装

本論文では、3 章で述べたストアの融合とループ化をルネサスエレクトロニクスのコンパイラ製品 CC-RL V1.08/CC-RX V3.01 に追加実装し、その効果を評価する。これらのコンパイラ製品はオープンソースのコンパイラインフラストラクチャ LLVM-2.3 [23] を利用して開発したものであり、コンパイルに際しては、LLVM と同様に、ソースコードをビットコードと呼ぶ機種非依存の中間表現に変換し、種々の最適化を適用したうえで、命令選択を行って機種依存の中間表現に変換し、最終的にアセンブリコードを得る。

我々はストアの融合とループ化を、いずれもビットコードに対する最適化として実装した。なぜなら、ビットコードならストア命令にバリエーションがなく、実装規模を小さくできるからである。機種依存の中間表現に対する最適化として実装すると、ストア命令が機種ごとに異なり、なおかつ、そのどのオペランドにストアする値やベースレジスタ、オフセットを受けとるのが異なることに対応する必要が生じるため、実装規模が大きくなってしまう。

また我々はストアの融合とループ化でまとめるストアの探索範囲を基本ブロック内に限定した。このことは実装規模を小さくする一方で、最適化の適用範囲を限定する。

ストアの融合とループ化の実装に際しては、最適化対象

が同一であることから、その使い分けが問題になった。ここではまず、ストアの融合とループ化の使い分けについて述べ、次に、2種のループ化、すなわち同じ定数向けと、異なる定数向けのループ化の使い分けについて述べる。

#### 4.1 ストアの融合とループ化の使い分け

ストアの融合と、ループ化は、最適化対象が同一なので、併用に際しては、効果の大きい方を適用する仕組が必要になる。この仕組の実装方法の1つは、それぞれの効果を見積もって、大きい方を選ぶ処理を実現することだが、実現には工数がかかる。

そこで我々は、工数をかけずに、ストアの融合と、ループ化の適切な使い分けを実現した。具体的には、最適化の適用順序を調整し、ストアの融合を先に実施することにした。この調整だけで適切な使い分けを実現できるとする理由は、融合した後のストアにもループ化を適用でき、その際に、ループ化を適用する方がサイズを小さくできるか判断しなおすことが可能だからである。

#### 4.2 同じ定数向けと異なる定数向けのループ化の使い分け

同じ定数をストアする命令列に対しては、同じ定数向けと、異なる定数向けの、両方のループ化を適用できる。したがって、最適化の適用に際しては、どちらを適用すべきかの判断が必要になる。ストア先の連続領域の全体に同じ定数をストアする場合、同じ定数向けのループ化の方がコードサイズを小さくできるのが一般的なので、そちらを選べばよいが、同じ定数をストアする記憶領域と隣接する領域に、異なる定数をストアする場合の判断は複雑になる。なぜなら、異なる定数をストアする命令列のループ化は、隣接する領域へのストアにもまとめて適用できるため、まとめての方が有益な場合の考慮が必要になるからである。

たとえば連続する記憶領域の前半に同じ定数を、後半に異なる定数をストアする場合について考える。この場合、前半の領域へのストア命令に対する最適化の選択岐には、次の (i), (ii), (iii) があり、後半の領域へのストア命令に対する最適化の選択岐には、(i), (iii) があるが、双方に (iii) を適用する場合のコードサイズは、前半と後半のコードサイズを個別に見積もった結果の和にならない。

- (i) 何もしない。
- (ii) 同じ定数をストアする命令列のループ化を適用する。
- (iii) 異なる定数をストアする命令列のループ化を適用する。

なぜなら双方に (iii) を適用する場合、前半と後半にまとめて (iii) を適用できるからである。ここで図 6(c) あるいは (d) の命令部分のサイズを  $C_d$ 、前半と後半の記憶領域長をそれぞれ  $n_h$ 、 $n_t$  とおくと、前半のみに (iii) を適用する場合、前半のサイズは  $C_d + n_h$  になり、後半のみに (iii) を適用する場合、後半のサイズは  $C_d + n_t$  になるが、双方に (iii) を適用する場合のサイズは  $C_d + n_h + n_t$  であっ

て、個別に見積もった結果の和である  $2C_d + n_h + n_t$  にならない。このことから、連続領域の前半のみで考えれば (i), (ii), (iii) の (ii)、すなわち同じ定数をストアする命令列向けのループ化を適用するのが最もコードサイズを小さくできるとしても、前半に (ii) を適用するのが最適とは限らないといえる。なぜなら前後半あわせて (iii) の異なる定数値向けのループにするのが最適な可能性があるからである。

##### 4.2.1 最適な適用方法を定めるアルゴリズム

同じ値をストアする領域と、隣接する領域をまとめられることを考慮に入れたうえで、ループ化の最適な適用方法を定めるには、単純には、次のアルゴリズムを用いればよい。

- (1) ストアの命令列を連続領域に同じ定数値をストアするものごとにグループにし、グループをストア先のアドレス順に並べる。グループを構成するストア命令の数は1以上であり、ストア命令のストアする定数値が、隣接するどちらの領域にストアする定数値とも異なる場合、当該ストア命令のみからなるグループを作る。
  - (2) 並べたグループについて、同じ値をストアする命令列向けのループ化を適用しても、コードサイズを小さくできないグループが隣接していたら、それらをまとめて1つのグループにする。まとめたグループには、同じ値をストアする命令列向けのループ化は適用できないが、異なる定数値をストアする命令列向けのループ化は適用できる。
  - (3) 隣接するグループとグループをまとめるか、あるいは仕切るかを表現する変数を作成する。変数はグループとグループの境界の数だけ作成するので、グループの数が  $n+1$  なら作成する変数の数は  $n$  になる。個々の変数は0/1のいずれかの値をとる。1の値をとった場合、まとめることを意味し、隣接するグループを(2)と同様に1つにまとめる。
  - (4) ステップ(3)で作った変数について、それらがとりうる値のすべての組合せに対応する命令列全体のコードサイズを計算し、コードサイズを最小にする変数の値の組合せを求める。変数の値の組合せに対応する命令列全体のコードサイズは、変数の値にしたがってグループ化を行った後、個々のグループのコードサイズの総和として求める。個々のグループのコードサイズは、最適化 (i), (ii), (iii) のうち、当該グループに適用しうるものごとに、適用後のサイズを見積もり、見積もった結果の最小値として求める。
  - (5) ステップ(4)で求めた変数の値の組合せに従ってストアをグループ化し、グループごとに最適化する。
- このアルゴリズムは単純で、4.3節に示すように、容易に実装できるが、計算量に問題がある。具体的には、このアルゴリズムが要求する計算量は、ステップ(3)で作る変

```

1:  for(最適化対象のサイズ = 1; 最適化対象のサイズ < アーキテクチャのワードサイズ; 最適化対象のサイズ *= 2){
2:      for(基本ブロック中の命令を前から順にたどり){
3:          if (今たどっている命令 i が次の条件のどれか 1 つでも満たさないなら
4:              - ストア命令である
5:              - ストアする値は整数である
6:              - ストアする値のサイズが最適化対象のサイズと一致する
7:              - ストアする値のサイズがストア先のアドレスの境界調整と一致する
8:              - ストア先を volatile で修飾していない
9:              - ストア先のアドレスをベース+整数の形で表現できる
10:             - (アーキテクチャが境界調整の条件をもたない ||
11:                ストア先のアドレスのベースの境界調整は、融合後のストアサイズの倍数である ||
12:                ストア先のアドレスのベースの境界調整を、融合後のストアサイズに修正できる)){
13:                 continue;
14:             }
15:         for(命令 i の次から基本ブロックの終端まで命令を順にたどり){
16:             if (今たどっている命令 j が次の条件のどれか 1 つでも満たさないなら
17:                 - ストア命令である
18:                 - ストアする値は整数である
19:                 - ストアする値のサイズが最適化対象のサイズと一致する
20:                 - ストアする値のサイズがストア先のアドレスの境界調整と一致する
21:                 - ストア先を volatile で修飾していない
22:                 - ストア先のアドレスをベース+整数の形で表現できる){
23:                 continue;
24:             }
25:             if (命令 i と命令 j のストア先のアドレスのベースが一致する &&
26:                 命令 i と命令 j のストア先のアドレスの整数の小さい方と最適化対象のサイズの和が、整数の大きい方と一致する &&
27:                 命令 i と命令 j のストア先のアドレスの整数の小さい方が最適化対象のサイズを 2 倍した数の倍数である &&
28:                 命令 i の直後に命令 j を移動できる){
29:                 命令 i でストアする値のビット幅を拡張し、命令 i で命令 j の分までストアするようにする;
30:                 10,11 行目でなく 12 行目の条件を満たしたなら、ストア先のベースの境界調整を融合後のストアサイズに修正する;
31:                 命令 j を削除する;
32:                 break;
33:             }
34:         }
35:     }
36: }

```

図 7 ストアの融合の実装

Fig. 7 Our implementation of store coalescing.

数の数  $n$  に対し、指数オーダで大きくなるので、 $n$  が大きいと膨大な計算が必要になる。変数の数を少なくするために、ステップ (2) で、グループの数を減らしているが、変数の数を十分に少なくできる保証はない。

#### 4.2.2 計算量の少ないアルゴリズム

そこで本論文では、計算量が  $n$  に比例するだけで済むアルゴリズムの実用性についても検証する。検証対象のアルゴリズムは 4.2.1 項に示したアルゴリズムのステップ (3) 以降を次の処理に差し替えたものである。

- (3) グループの並びに残っているグループが 1 つだけなら、当該グループを単独で最適化して処理を終了する。さもなければステップ (4) に進む。
- (4) グループの並びの先頭 2 つについて、それぞれ個別に最適化すると、2 つを 1 つのグループにまとめて最

適化すると、どちらがコードサイズを小さくできるか見積もる。

見積の結果、まとめる方が小さくできると判断したら、まとめてステップ (3) に戻る。まとめない方が小さくできると判断したら、先頭のグループを単独で最適化してグループの並びから外し、ステップ (3) に戻る。

#### 4.3 各最適化の実装の詳細

我々が実装したストアの融合のアルゴリズムの詳細を図 7 に、ループ化のアルゴリズムの詳細を図 8, 図 9, 図 10, 図 11, 図 12, 図 13 に示す。ループ化のアルゴリズムは同じ定数向けと異なる定数向けのループ化の使い分けも含んでおり、まず図 8 の処理で基本ブロック中の連続したストア命令の列を抽出し、抽出したストア命令の

---

```

1:  for(ii = 基本ブロックの先頭を指示するイテレータ; ii != 基本ブロックの終端;){
2:      if (ii の指示する命令が次の条件のいずれか 1 つでも満たさないなら
3:          - ストア命令である
4:          - ストアする値が定数である){
5:          ++ii;
6:          continue;
7:      }
8:      最適化対象候補のストア命令列 = 空の配列;
9:      basePtr = ii のストア先をベース+整数と表現する場合のベース;
10:     最適化対象候補のストア命令列の末尾に ii の指示するストア命令を追加する;
11:     while(++ii != 基本ブロックの終端){
12:         if (ii の指示する命令が次の全条件を満たすなら
13:             - ストア命令である
14:             - ストアする値が定数である
15:             - ストア先のアドレスをベース+整数と表現する場合のベース == basePtr){
16:             if (最適化対象候補のストア命令列のいずれかのストア先が, ii のストア先と重なる){
17:                 break;
18:             }
19:             最適化対象候補のストア命令列の末尾に ii の指示するストア命令を追加する;
20:         }
21:         else if (ii の指示する命令がメモリを読み書きする){
22:             break;
23:         }
24:     }
25:     最適化対象候補のストア命令列について, 連続領域へのストアごとに最適化を試みる;
26: }

```

---

図 8 ループ化の実装

Fig. 8 Our implementation of transformation to loop.

---

```

1:  while(最適化対象候補のストア命令列が空でない){
2:      連続領域へのストア命令列 = 空の配列;
3:      最適化対象候補のストア命令列から最初のストア命令を取り出し, 連続領域へのストア命令列に納める;
4:      for(sii = 最適化対象候補のストア命令列の先頭; sii != 最適化対象候補のストア命令列の終端;){
5:          if (sii の指示するストア命令のストア先が連続領域へのストア命令列のストア先の直後である &&
6:              (最適化対象候補のストア命令列の先頭から sii の直前までの全ストア命令のストア先を volatile で修飾していない ||
7:               sii の指示するストア命令のストア先を volatile でない)){
8:              最適化対象候補のストア命令列から sii の指示する命令を取り出し, 連続領域へのストア命令列の末尾に追加する;
9:              sii を最適化対象候補のストア命令列の先頭に戻す;
10:         }
11:         else if (sii の指示するストア命令のストア先が連続領域へのストア命令列のストア先の直前 &&
12:                 (連続領域へのストア命令列中のストア命令のストア先と, 最適化対象候補のストア命令列の
13:                  先頭から sii の直前までの全ストア命令のストア先を volatile で修飾していない ||
14:                  sii の指示するストア命令のストア先を volatile でない)){
15:             最適化対象候補のストア命令列から sii の指示する命令を取り出し, 連続領域へのストア命令列の先頭に追加する;
16:             sii を最適化対象候補のストア命令列の先頭に戻す;
17:         }
18:         else{
19:             ++sii;
20:         }
21:     }
22:     連続領域へのストア命令列の最適化を試みる;
23: }

```

---

図 9 連続するストア命令のループ化の実装

Fig. 9 Our implementation of transformation of sequential stores to loop.

---

```

1:  typedef struct{
2:     最適化方法の候補;
3:     int _accessWidth;
4:     _Bool _hasVolatile;
5:     int _storeValue;
6:     グループ内のストア命令列;
7: } StoreGroup;
8:
9: StoreGroup を納める配列 groups = 空の配列;
10: for(sii = 連続領域へのストア命令群の先頭; sii != 連続領域へのストア命令群の末尾;){
11:     StoreGroup group = (StoreGroup){
12:         .最適化方法の候補 = { なにもしない, 同じ値をストアする命令列のループ化, 異なる値をストアする命令列のループ化 },
13:         ._accessWidth = sii の指示するストア命令がストアする整数数のビット幅,
14:         ._hasVolatile = sii の指示するストア命令のストア先を volatile,
15:         ._storeValue = sii の指示するストア命令のストアする整数数,
16:         .グループ内のストア命令列 = sii の指示するストア命令 };
17:     while(++sii != 連続領域へのストア命令群の末尾){
18:         if (group._accessWidth == sii の指示するストア命令がストアする整数数のビット幅){
19:             if (group._storeValue != sii の指示するストア命令がストアする整数数){
20:                 break;
21:             }
22:         } else if (group._hasVolatile){
23:             break;
24:         } else if (group._accessWidth < sii の指示するストア命令がストアする整数数のビット幅){
25:             if (sii の指示するストア命令がストアする整数数は group._storeValue の繰り返しでできていない){
26:                 break;
27:             }
28:         } else{
29:             if (group._storeValue は sii の指示するストア命令がストアする整数数の繰り返しでできていない){
30:                 break;
31:             }
32:         }
33:         group._accessWidth = sii の指示するストア命令がストアする整数数のビット幅;
34:         group._storeValue = sii の指示するストア命令がストアする整数数;
35:     }
36:     group._hasVolatile |= sii の指示するストア命令のストア先を volatile で修飾している;
37:     group のグループ内のストア命令列に sii の指示するストア命令を追加する;
38: }
39: if (group のグループ内のストア命令列に, 同じ値をストアする命令列のループ化を適用してもコードサイズが小さくならない){
40:     if (配列 groups は空でない &&
41:         配列 groups の末尾の要素の最適化方法の候補に, 同じ値をストアする命令列のループ化がない &&
42:         配列 groups の末尾の要素と group の 2 つあわせて, 異なる値をストアする命令列のループ化を適用できる){
43:         配列 groups の末尾の要素のグループ内のストア命令列の末尾に group のグループ内のストア命令列を追加する;
44:         continue;
45:     }
46:     group の最適化方法の候補から, 同じ値をストアする命令列のループ化を除く;
47: }
48: 配列 groups の末尾に group を追加する;
49: }
50: 配列 groups 中の各 group の最適化方法の候補の組合せの中で適切なものを選ぶ;
51: 選んだ組合せに従って配列 groups 中の各要素を最適化する;

```

---

図 10 連続領域へのストアのループ化の実装

Fig. 10 Our implementation of transformation to loop for stores to continuous space.

```

1: 適用する最適化方法の組合せ = { 配列 gropus 中のどの要素に対してもなにもしない };
2: 試行した範囲で最小のサイズ = 配列 groups 中の各要素が保持するストア命令のサイズの総和;
3: for(配列 gropus 中の各要素の最適化方法の候補のすべての組合せについて){
4:     if (その組合せで最適化する場合のサイズ < 試行した範囲で最小のサイズ){
5:         適用する最適化方法の組合せ = その組合せ;
6:         試行した範囲で最小のサイズ = その組合せで最適化する場合のサイズ;
7:     }
8: }
9: return 適用する最適化方法の組合せ;

```

図 11 連続領域へのストアの最適な最適化方法の組合せの選定の実装

Fig. 11 Implementation of an optimal optimization method selection for each store to continuous space.

```

1: 見積結果 = 0;
2: for(i = 0; i < 配列 groups の要素数; ++i){
3:     switch(配列 groups の i 番目の要素の最適化方法){
4:     case なにもしない:
5:         見積結果 += 配列 groups の i 番目の要素のグループ内のストア命令列中の命令のサイズの総和;
6:         continue;
7:     case 同じ値をストアする命令列のループ化:
8:         見積結果 +=
9:             配列 groups の i 番目の要素のグループ内のストア命令列に同じ値をストアする命令列のループ化を適用する場合のサイズ;
10:        continue;
11:     case 異なる値をストアする命令列のループ化: {
12:         StoreGroup fusedGroup = 配列 groups の i 番目の要素のコピー;
13:         for(;i+1 < 配列 groups の要素数; ++i){
14:             if (配列 groups の i+1 番目の要素の最適化方法は異なる値をストアする命令列のループ化でない){
15:                 break;
16:             }
17:             if (fusedGroup._accessWidth < 配列 groups の i+1 番目の要素._accessWidth){
18:                 if (配列 groups の i+1 番目の要素._hasVolatile){
19:                     break;
20:                 }
21:             } else if (fusedGroup._accessWidth > 配列 groups の i+1 番目の要素._accessWidth){
22:                 if (fusedGroup._hasVolatile){
23:                     break;
24:                 }
25:                 fusedGroup._accessWidth = 配列 groups の i+1 番目._accessWidth;
26:             }
27:             fusedGroup.グループ内のストア命令の集合 += 配列 groups の i+1 番目の要素のグループ内のストア命令の集合;
28:         }
29:         見積結果 += fusedGroup のグループ内のストア命令列に異なる値をストアする命令列のループ化を適用する場合のサイズ;
30:     }
31: }
32: }
33: return 見積結果;

```

図 12 最適化後のコードサイズの見積の実装

Fig. 12 Implementation of optimized code size estimation.

列を図 8 の 25 行目から図 9 の処理に引き渡して連続領域へのストア命令列に切り分け、切り分けた個々の命令列を図 9 の 22 行目から図 10 の処理に引き渡して最適化する。図 10 の最適化処理では、まず連続領域へのストア命令列を、17 から 38 行目の処理で切り分けて、同じ値をストア

する命令列のループ化を適用できるものごとにまとめ、39 から 47 行目の処理で、まとめたもののうち、同じ値をストアする命令列のループ化ではコードサイズが小さくなりえないものどうしをまとめ、50 行目でまとめた個々をどう最適化するかを定め、定めた結果に従って 51 行目で最

```

1:  for(i = 0; i < 配列 groups の要素数-1;){
2:      StoreGroup* gi0 = 配列 groups の i 番目の要素;
3:      StoreGroup* gi1 = 配列 groups の i+1 番目の要素;
4:      if ((gi0->_hasVolatile && (gi0->_accessWidth > gi1->_accessWidth)) ||
5:          (gi1->_hasVolatile && (gi1->_accessWidth > gi0->_accessWidth)) ||
6:          (gi0 を単独で最適化する場合の最小サイズ + gi1 を単独で最適化する場合の最小サイズ <
7:           gi0 と gi1 にまとめて異なる値をストアする命令例のループ化を適用する場合のサイズ)){
8:          gi0 の最適化方法の候補が複数あるなら, gi0 を単独で最適化する場合にサイズを最小にする最適化方法のみにする;
9:          ++i;
10:     }
11:     else{
12:         gi0->最適化方法の候補 = { 異なる値をストアする命令列のループ化 };
13:         gi0->_accessWidth = min(gi0->_accessWidth, gi1->_accessWidth);
14:         gi0->_hasVolatile |= gi1->_hasVolatile;
15:         gi0 に gi1 のグループ内のストア命令列を追加;
16:         配列 groups から i+1 番目の要素を除去する;
17:     }
18: }

```

図 13 連続領域へのストアの単純な最適化方法の組合せの選定の実装  
**Fig. 13** Implementation of an simple optimization method selection for each store to continuous space.

最適化を行う。50 行目の、個々をどう最適化するかを定める方法である、4.2.1 項と 4.2.2 項のアルゴリズムのうち、4.2.1 項のアルゴリズムの詳細を図 11 に示す。また、図 11 の 4 行目で行う、最適化後のコードサイズの見積処理の詳細を図 12 に示す。4.2.2 項に示したアルゴリズムの詳細を図 13 に示す。

なお、図 7 から図 13 の実装は、いずれも最適化対象が定数をストアする命令であるため、適用範囲を広げるために、定数にできる式をあらかじめ畳み込んでおく方がよい。そこで我々は、定数にできる式を畳み込んだうえでこれらの最適化を適用した。また、ループ化に関連する最適化として、ループ融合 [24] がある。ループ融合は隣接するループを 1 つのループに書き換える最適化であり、たとえば図 14 (a) の 2 つのループを図 14 (b) の 1 つのループに書き換える。ループ化で生成したループにループ融合を適用することでコードサイズを削減できる可能性はあるが、ループ融合とコードサイズの削減を両立するにはループの周回回数をそろえる必要がある。ループ化で生成するループ群の周回回数をそろえられるかは定かではなく、我々の実装はループ化の後にループ融合を適用しない。

我々のループ化の実装はメモリへのストア回数を変更しないが、変更すればさらにコードサイズを削減できる場合もある。たとえば図 15 のソースコードを最適化する場合について考える。図 15 のソースコードは配列の 1,000 番目の要素にのみ 1 を、他を 0 をストアする。我々のループ化は図 15 を、2 つのループ、すなわち 0 から 999 番目の要素にストアするループと、1,001 から 9,999 番目の要素にストアするループにするが、それより配列の全体に

<pre> for(i=0; i&lt;10; ++i){     a[i] = 0; } for(i=0; i&lt;10; ++i){     b[i] = 0; } </pre>	<pre> for(i=0; i&lt;10; ++i){     a[i] = 0;     b[i] = 0; } </pre>
(a) 融合前	(b) 融合後

図 14 ループ融合  
**Fig. 14** Loop fusion.

```

1:  array[0] = 0;
2:  ...
3:  array[999] = 0;
4:  array[1000] = 1;
5:  array[1001] = 0;
6:  ...
7:  array[9999] = 0;

```

図 15 同一箇所への複数回のストアによるコードの削減  
**Fig. 15** Code size reduction by storing multiple times.

たん 0 をストアしてから 1,000 番目の要素に 1 をストアする方がコードサイズを小さくできる。我々の実装がストア回数の変更を必要とする最適化を行わない理由は、ストア回数の変更を許容できるか否かを保守的に判断した結果である。

## 5. 評価

本章では、3 章で述べた最適化、すなわちストアの融合と、ループ化の効果を明らかにする。これらの最適化は基本的にコードサイズの削減を目的としたものであり、した

表 1 コードサイズの削減率  
Table 1 Code size reduction.

ベンチ マーク 種別	コードサイズ (削減前, KByte)		コードサイズの削減率 (%)			
	RL78	RX	融合のみ適用		ループ化を併用	
			RL78	RX	RL78	RX
産業	6	10	0.00	0.00	0.00	0.00
	2	3	0.00	0.00	0.00	0.00
民生	59	71	0.00	0.30	0.05	0.30
	19	26	0.09	0.11	0.12	0.14
	31	35	-0.01	0.04	-0.01	0.04
	39	48	0.09	0.15	0.12	0.15
OA	33	16	0.00	0.02	0.00	-0.02
	130	150	0.10	0.12	0.15	0.13
	4	4	0.00	0.07	0.00	0.07
	18	10	0.23	0.52	0.18	0.42
	24	20	0.18	0.12	0.21	3.53
車載	17	17	-0.10	0.18	-0.10	0.18
	17	19	0.05	0.04	0.21	-0.19
	5	6	0.08	0.12	0.08	0.12
	3	4	0.06	0.15	0.06	0.15
	21	25	0.00	0.00	0.00	0.00
	13	16	0.00	0.06	0.00	0.06
	13	15	0.00	0.00	0.00	0.00
	5	3	3.90	7.16	7.07	8.52
	38	46	0.00	0.19	0.00	0.19
	23	40	0.06	0.27	0.23	0.27
	107	119	0.03	0.21	0.07	0.25
	163	209	0.56	1.29	0.65	1.30

がって評価対象はコードサイズにあたる影響とする。ストアの融合に限れば実行速度の改善にも有効だが、主な最適化対象が初期化処理で、その実行頻度が低いため効果は大きくない。実際、組込マイコンの速度の計測に用いるベンチマークである CoreMark [25] のスコアにあたる影響は、CC-RL で 0%、CC-RX で 0.000003% にすぎなかった。

コードサイズの測定対象は車載/民生/Office Automation (OA と略記する)/産業の各分野から集めた 23 個の実アプリケーションとした。測定に際しては、CC-RL/CC-RX いずれのコンパイラにも最適化の目標をコードサイズの削減とするオプションを指定した。

評価結果を表 1 に示す。表 1 には各アプリケーションにストアの融合やループ化を適用しない場合に比べて、適用することでどれだけコードサイズが減るかを示した。異なる定数をストアする命令列のループ化は、コードサイズを減らす代わりにデータサイズを増やす最適化であり、たとえば図 6(c) の 23 行目にあるようなデータを作りだすが、作りだしたデータのサイズは最適化後のコードのサイズに加算した。ループ化の適用方法を定めるアルゴリズムについては、4.2.1 項に示したものを採用したが、4.2.2 項に示したものを使っても結果は同じだった。

表 1 から、ストアの融合やループ化を広範なアプリケー

表 2 境界調整のしなおいしが融合対象のストア命令数に与える影響  
Table 2 Effect of align feature to number of stores coalesced.

境界調整の条件がなければ融合できたストア命令の数に対する、同条件があっても融合できたストア命令の数の比率	(%)
境界調整しなおいす対象	
なし	8.9
自動変数+同一ファイルで定義する大域変数	64.4
自動変数+任意の大域変数	97.9

ションに適用でき、それらがもたらすコードサイズの削減率が最大で 8.52% になるといえる。削減率が最大になったアプリケーションは、9 つのファイルからなるが、ストアの融合やループ化の適用先は、そのうち 1 つに集中しており、当該ファイルのコード量は 769 バイトから 474 バイトへと、38.4% 減っていた。

### 5.1 ストアの融合の適用範囲に境界調整が与える影響

表 1 の CC-RX の評価結果と、CC-RL の評価結果を比較すると、CC-RX のみストアの融合の適用範囲が広いことが分かる。具体的には、CC-RX では、評価対象とした 23 のアプリケーションのうち 19 にストアの融合を適用できているが、CC-RL では 14 にしか適用できていない。CC-RL で適用範囲が狭くなる理由は、RL78 マイコンでは境界調整の条件を満たさないアドレスにストアできないため、同条件を持たない RX マイコンに比べ、ストアを融合できる場合が少ないことにある。CC-RL 向けのストアの融合では、ストア先が自動変数もしくは最適化対象の関数と同一ファイルに定義がある大域変数と分かる場合に、当該変数を境界調整しなおしてストア融合の適用箇所数を増やした。具体的には、図 7 の 12 行目で境界調整を修正できると見なす条件を、ストア先が自動変数もしくは同一ファイルに定義がある大域変数であることとした。表 1 は当該機能を有効にして測定した結果である。当該機能の効果を表 2 に示す。

表 2 に示した値は、評価対象の全 23 アプリケーションを通じて、境界調整の条件がなければ融合できたストアの数に比べ、同条件があっても融合できたストアの数がどれだけあったかの比率を示す。表 2 には、我々が実装した境界調整のしなおしの機能を有効にした場合の比率に加え、比較対象として、調整しなおさなかった場合の比率と、自動変数と任意の大域変数を調整しなおした場合の比率も示した。表 2 から分かるように、境界調整の条件が存在する状況下で融合できるストアの数は、同条件が存在しない状況下に比べ、境界調整をしなおさなければ 8.9% にとどまったが、我々が実装した境界調整の機能を使えば 64.4% に、さらに任意の大域変数を境界調整すれば 97.9% になった。

### 5.2 ループ化を使い分けるアルゴリズムの実用性

表 1 の評価は、ループ化の使い分けの実現に 4.2.1 項に

表 3 ループ化の適用箇所数の分布

Table 3 Distribution of number of store sequences transformed to loops.

変数の数	ループ化の適用箇所数	
	コンパイラ	
	CC-RL	CC-RX
0	75	29
1	1	2
2	1	0
3	0	0
4	0	0
5	1	0

示したアルゴリズムを用いても実施できた。4.2.1 項に示したアルゴリズムには、変数の数、すなわち一括して最適化方法を選択するストア命令のグループ数から 1 を減じた数に対し、計算量が指数のオーダーで大きくなるという欠点があるが、それにもかかわらず評価できた理由は、評価対象のアプリケーションの中に、それほど多くの変数を必要とするものがなかったからである。全評価対象を通した変数の数に対するループ化の適用箇所数の分布は表 3 に示すとおりで、変数の数は最大でも 5 だった。

もっとも、評価の過程で、4.2.1 項のアルゴリズムを使うことの利得を見い出すこともなかった。なぜなら、より単純で  $n$  に比例する計算時間しか消費しない 4.2.2 項のアルゴリズムでも同一の最適化結果を得られたからである。

なお、表 3 について、各コンパイラの変数の数の総和は、ループ化の適用箇所数と解釈することもできるが、CC-RX の適用箇所数は  $29+2=31$  と、CC-RL に比べて少ない。少ない理由は、RX マイコンが境界調整の条件を持たないことから、CC-RX ではストアを融合できるケースが多く、結果としてループ化を必要とする箇所が減っていることによる。CC-RX に比べて、CC-RL でループ化の適用箇所数が多いことは、境界調整の条件が原因でストアを融合できないケースの救済措置として、ループ化を活用した結果ともいえる。ストアの融合を抑制した場合に、CC-RL/CC-RX のループ化の適用箇所数がいくつになるか調査したところ、それぞれ 147/344 カ所と、CC-RX の方が多くなった。CC-RX の方が多くなった理由は、RX マイコンがループ化のループ部分に相当する命令 `SSTR` や `SMOVF` を提供することから、CC-RX の方がより短いストアの列にもループ化を適用するためである。

## 6. 関連研究

ストアの融合は既存の最適化であり、ループ化も既存の最適化である loop rerolling で実現できることもある。単純な loop rerolling では、ストアする定数が異なる、あるいは、ストアのビット幅が異なる命令列をループにすることは難しいが、既存のコンパイラの中には、同じ定数をスト

アする命令列に特化した最適化を持ち、ビット幅が異なるストアをループ化できるものがある [23]。本論文では、ストアの融合やループ化、正確には同じ定数をストアする命令列のループ化と、異なる定数をストアする命令列のループ化を使い分ける方法を示した。また、これらの最適化が組込分野の実アプリケーションに与える効果を評価し、また、その効果にマイクロコントローラのアーキテクチャの違い、具体的には境界調整の条件を持つか否かと、連続する記憶領域へのアクセスに特化した命令を持つか否かが与える影響を示した。

本論文で評価の対象としたストアの融合やループ化は、まとめる対象のストアを単一の基本ブロック内から収集するが、ストアの融合に似た最適化である、ロードの融合を、基本ブロックをまたいで適用する研究もある [26]。またストアの融合について、本論文では、定数でない値をストアする場合は、図 3 に示すように、命令数を増やしうることから最適化の対象外としたが、複数のレジスタをまとめて 1 つのレジスタとしてストアできるアーキテクチャや、1 命令で複数の値をストアできるアーキテクチャでは命令を増やすとは限らず、融合が有用になりうる。後者のアーキテクチャの活用については Johnson らの研究がある [13]。

## 7. 結論

連続する記憶領域に定数をストアする命令列のサイズ削減に有用な最適化である、ストアの融合とループ化について、それらの実装方法や、使い分ける方法を示した。また、評価を通じて、マイクロコントローラのアーキテクチャがストアの融合やループ化の効果に与える影響を示した。評価の結果、ストアの融合やループ化によって、実アプリケーションのサイズを最大で 8.52%削減できることが分かった。

## 参考文献

- [1] Microchip Technology Inc.: Microchip AVR MCUs (2019), available from (<https://www.microchip.com/design-centers/8-bit/avr-mcus/>).
- [2] Microchip Technology Inc.: 8-bit PIC MCUs (2019), available from (<https://www.microchip.com/design-centers/8-bit/pic-mcus/>).
- [3] Microchip Technology Inc.: 8051 Microcontrollers (2019), available from (<https://www.microchip.com/design-centers/8-bit/8051-microcontrollers/>).
- [4] NXP Semiconductors: 8-bit S08 MCUs (2019), available from (<https://www.nxp.com/products/processors-and-microcontrollers/additional-architectures/8-bit-s08-mcus:HCS08/>).
- [5] Renesas Electronics Corporation: 8/16-bit Ultra-low energy MCUs (RL78) (2019), available from (<https://www.renesas.com/us/en/products/microcontrollers-microprocessors/rl78.html>).
- [6] STMicroelectronics: STM8 8-bit MCUs (2019), available from (<https://www.st.com/en/microcontrollers-microprocessors/stm8-8-bit-mcus.html>).

- [7] Texas Instruments Incorporated.: MSP430 ultra-low-power sensing & measurement MCUs (2019), available from (<http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html>).
- [8] Microchip Technology Inc.: SAM D MCUs (2019), available from (<https://www.microchip.com/design-centers/32-bit/sam-32-bit-mcus/sam-d-mcus/>).
- [9] Renesas Electronics Corporation: 32-bit High power efficiency MCUs (RX) (2019), available from (<https://www.renesas.com/us/en/products/microcontrollers-microprocessors/rx.html>).
- [10] Silicon Laboratories: EFM32 32-bit Microcontrollers (2019), available from (<https://www.silabs.com/products/mcu/32-bit/>).
- [11] STMicroelectronics: STM32 Ultra Low Power MCUs (2019), available from (<https://www.st.com/en/microcontrollers-microprocessors/stm32-ultra-low-power-mcus.html>).
- [12] Davidson, J.W. and Jinturkar, S.: Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses, *SIGPLAN Not.*, Vol.29, No.6, pp.186-195 (online), DOI: 10.1145/773473.178259 (1994).
- [13] Johnson, N. and Mycroft, A.: Using Multiple Memory Access Instructions for Reducing Code Size, *Proc. Compiler Construction, CC 2004*, Lecture Notes in Computer Science, Berlin, Heidelberg, Springer (2004).
- [14] Renesas Electronics Corporation: C Compiler Package for RL78 Family (2019), available from (<https://www.renesas.com/us/en/products/software-tools/tools/compiler-assembler/compiler-package-for-rl78-family.html>).
- [15] Renesas Electronics Corporation: C/C++ Compiler Package for RX Family (2019), available from (<https://www.renesas.com/us/en/products/software-tools/tools/compiler-assembler/compiler-package-for-rx-family.html>).
- [16] Su, B., Ding, S. and Jin, L.: An Improvement of Trace Scheduling for Global Microcode Compaction, *SIGMICRO Newsl.*, Vol.15, No.4, pp.78-85 (online), DOI: 10.1145/384281.808217 (1984).
- [17] Su, B., Ding, S. and Xia, J.: URPR&Mdash; An Extension of URCR for Software Pipelining, *SIGMICRO Newsl.*, Vol.17, No.4, pp.94-103 (online), DOI: 10.1145/19530.19541 (1986).
- [18] Fraser, C.W., Myers, E.W. and Wendt, A.L.: Analyzing and Compressing Assembly Code, *SIGPLAN Not.*, Vol.19, No.6, pp.117-121 (online), DOI: 10.1145/502949.502886 (1984).
- [19] 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).
- [20] Dreweke, A., Worlein, M., Fischer, I., Schell, D., Meinel, T. and Philippesen, M.: Graph-Based Procedural Abstraction, *Proc. International Symposium on Code Generation and Optimization, CGO '07*, pp.259-270, IEEE Computer Society (online), DOI: 10.1109/CGO.2007.14 (2007).
- [21] Schaeckeler, S. and Shang, W.: Procedural Abstraction with Reverse Prefix Trees, *Proc. 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09*, pp.243-253, IEEE Computer Society (online), DOI: 10.1109/CGO.2009.25 (2009).
- [22] Edler von Koch, T.J., Franke, B., Bhandarkar, P. and Dasgupta, A.: Exploiting Function Similarity for Code Size Reduction, *SIGPLAN Not.*, Vol.49, No.5, pp.85-94 (online), DOI: 10.1145/2666357.2597811 (2014).
- [23] Lattner, C.: The LLVM Compiler Infrastructure (2019), available from (<http://www.llvm.org>).
- [24] Wagner, R.A.: Some Techniques for Algorithm Optimization with Application to Matrix Arithmetic Expressions, PhD Thesis, Department of Computer Science, Carnegie-Mellon University (1968).
- [25] The Embedded Microprocessor Benchmark Consortium: CoreMark An EEMBC Benchmark (2009), available from (<http://www.eembc.org/coremark/>).
- [26] Kawahito, M., Komatsu, H. and Nakatani, T.: Instruction Combining for Coalescing Memory Accesses Using Global Code Motion, *Proc. 2004 Workshop on Memory System Performance, MSP '04*, pp.2-11, ACM (online), DOI: 10.1145/1065895.1065897 (2004).



千葉 雄司 (正会員)

1972年生。1997年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了。株式会社日立製作所においてコンパイラの開発に従事。中央大学非常勤講師、中央大学大学院客員教授を兼任。



平下 敬之

1978年生。2001年明治大学理工学部電子通信工学科卒業。ルネサスエレクトロニクス株式会社においてコンパイラの開発に従事。