情報処理学会論文誌　プログラミング　Vol.14　No.1　1–14 (Jan. 2021)

Branch chainingのRXマイコン向け実装

千葉 雄司[1,a)]　永井 佑樹[2]　中川 満[3]

受付日 2020年5月25日, 採録日 2020年9月30日

概要：コードサイズを削減する最適化の1つに，branch chainingがある．branch chainingは，分岐命令のサイズが分岐先までの距離の増加に応じて増えるアーキテクチャ向けの最適化であり，遠方への分岐命令の近傍に，同一箇所への分岐命令があるとき，一方の分岐先を他方の分岐命令にすることで，分岐の実現を，他方の分岐命令経由とすることと引き換えに，分岐先を近傍とし，命令のサイズを小さくする．Leverettらは，branch chainingにおいて，どの分岐命令のサイズを小さくすべきかの組合せ最適化問題の最適解を得るアルゴリズムを示したが，その有用性を評価していない．そこで本論文では，Leverettらが提案したアルゴリズムと，本論文で提案する，近似解を求めるアルゴリズムをRXマイコン向けに実装，評価した結果を示す．組込機器向けベンチマークEEMBCによる評価の結果，コードサイズの削減量は，Leverettらのアルゴリズムでは0.40%，近似解を求めるアルゴリズムでは0.35%になることが分かり，実行命令数の増加については，いずれも相乗平均で0.14%になることが分かった．ビルド時間については，それぞれ最大で314倍，1.01倍になった．

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 Branch Chaining for RX Microcontrollers

Yuji Chiba[1,a)]　Yuki Nagai[2]　Mitsuru Nakagawa[3]

Received: May 25, 2020, Accepted: September 30, 2020

Abstract: Branch chaining is one of the optimizations to reduce code size and works for an architecture whose branch instruction size increases in accordance with the branch distance. Branch chaining reduces branch instruction size by changing the branch destination to a near-by branch instruction sharing the destination, and to select which branch instruction to shorten is a combinatorial optimization problem. Leverett proposed an algorithm to find the optimal solution, but the effectiveness is not clear. This paper implements and evaluates branch chaining algorithms. One of algorithms the is based on the Leverett's algorithm and expanded for RX microprocessor, and the others seeks for a suboptimal solution. The evaluation showed that Leverett's algorithm reduces EEMBC code size by 0.40% while suboptimal one does by 0.35%, increase the geometrical mean number of instructions to execute by 0.14% for both of the two, and extend build time 314 times and 1.01 times at most.

Keywords: compiler, optimization, code size, embedded device

1. はじめに

近年，組込み機器に対する要求には，コストの削減と，ソフトウェアの高機能化があるが，これらの両立は必ずしも容易でない．なぜなら，コストを削減するためには，廉価な組込みマイコンを採用するのが有効だが，廉価な組込みマイコンが内蔵するソフトウェアの記憶領域，すなわち内蔵フラッシュメモリの容量は小さいことが多く，高機能なソフトウェアの格納に適すとは限らないからである．たとえば，本論文執筆時点で市場に流通しているマイコンの，最も廉価なモデルの内蔵フラッシュメモリのサイズは，8から16 bitマイコンでは0.5から2 KByte [1], [2], [3], [4], [5], [6], [7],

1 株式会社日立製作所
Hitachi, Ltd., Kokubunji, Tokyo 185–8601, Japan
2 株式会社日立ソリューションズ
Hitachi Solutions, Ltd., Shinagawa, Tokyo 140–0002, Japan
3 ルネサスエレクトロニクス株式会社
Renesas Electronics Corporation, Kodaira, Tokyo 187–8588, Japan
a) yuji.chiba.pd@hitachi.com

© 2021 Information Processing Society of Japan

32 bit マイコンでは 4 から 8 KByte [8], [9], [10], [11] にすぎない．サイズの小さな領域に高機能なソフトウェアを収められるようにする手段の 1 つは，ソフトウェアの無駄を省いてサイズを小さくすることだが，手作業で小さくすると大きなコストがかかって問題になる．

この問題に対処する手段の 1 つに，言語処理系の最適化がある．言語処理系の中でも，特に，組込みマイコン向けのものは，ソフトウェアのサイズを削減する最適化機能を提供し，ソフトウェアのソースコードをコンパイル，アセンブル，リンクする各段階で様々な最適化を適用する．サイズを削減する最適化に関する研究は過去に数多く存在するが，本論文では，過去に Leverett らが提案した最適化である branch chaining [12] をとりあげる．

本論文では，まず 2 章で Leverett らが提案した branch chaining と，その効果を最大にするアルゴリズムについて述べる．続く 3 章では，Leverett らのアルゴリズムの RX マイコン向け拡張と，計算量を削減する手法を提案する．4 章では，branch chaining の効果を最大にはできないが，計算量の少ないアルゴリズムを提案し，5 章で各アルゴリズムがコードサイズや実行命令数，ビルド時間に与える影響を評価する．6 章は結論である．

2. 関連研究

Leverett らが提案した branch chaining は，分岐命令のサイズが分岐先までの距離の増加に応じて増えるアーキテクチャ向けの最適化であり，遠方への分岐命令の近傍に，同一箇所への分岐命令があるとき，一方の分岐先を他方の分岐命令にすることで，分岐の実現を，他方の分岐命令経由とすることと引き換えに，分岐先を近傍とし，命令のサイズを小さくする．

たとえば**図 1** の命令列について考える．図 1 の命令を始め，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の RX マイコン [9] のものである．図中の命令 `BRA` や `BEQ` が分岐命令である．RX マイコンの分岐命令には分岐可能な距離に応じたバリエーションを持つものがあるが，図中の分岐命令がどのバリエーションに相当するかは，分岐命令に後続する `.A` などのサイズ指定子によって判別できる．サイズ指定子は分岐命令のサイズを定めるものでもある．RX マイコンの分岐命令に付与しうるサイズ指定子と，サイズ指定子に対応する分岐命令のサイズを**表 1** に示す．

さて，図 1 (a) の命令列には 4 行目と 11 行目に `BB128`，10 行目と 15 行目に `BB256` への分岐命令がある．これらの分岐命令のサイズを小さくするために，Leverett らが提案した branch chaining は，たとえば 11 行目の分岐命令の分岐先を 4 行目の分岐命令に，10 行目の分岐命令の分岐先を 15 行目の分岐命令に変更する．変更前後の図 1 の (a) (b) の命令列を比較すると，分岐先の変更により，11 行目と 10 行目の分岐命令のサイズ指定子が変化し，それぞれのサイズが小さくなっていることを読み取れる．図 1 (a) から図 1 (b) への書き換えは，プログラムのサイズを小さくする一方で，実行時に分岐命令の実行回数を増やし，実行速度の低下を招くものでもある．なお，Wulf らは branch chaining という語を，Leverett らの提案とは逆の最適化，すなわち図 1 (b) を図 1 (a) に書き換え，実行速度を向上する最適化を意味するものとして用いているが [13]，本論文では Leverett らにならい，図 1 (a) から図 1 (b) への書き換えを branch chaining と呼ぶものとし，Wulf らの最適化は，その llvm [14] における実装の名称にならい，branch folding と呼ぶものとする．

図 1 (a) への branch chaining への適用に際しては，4 行目と 11 行目のどちらを書き換えることも，10 行目と 15 行目のどちらを書き換えることもできる．図 1 (b) への書き換えで 11 行目と 10 行目を書き換えた理由は，11 行目を書き換えて小さくしたうえで，10 行目を書き換えることで，10 行目の分岐命令のサイズを最も小さな 1 バイトにするためだが，この書き換えでは，たとえば 4 行目の分岐命令のサイズが大きなままなので，それをまたぐ 7 行目の分岐命令のサイズは小さくできない．一般に，branch chaining の効果を最大にするには，多様な書き換えパターンの組合

```
 1:  BB1:
 2:    ...
 3:
 4:    BRA.A BB128
 5:  BB3:
 6:    CMP R2, R3
 7:    BEQ.W BB1
 8:  BB4:
 9:    CMP R4, R5
10:    BEQ.W BB256
11:    BRA.A BB128
12:  BB6:
13:    CMP 65536, R3
14:
15:    BEQ.W BB256
```
(a) 適用前

```
 1:  BB1:
 2:    ...
 3:  BB2:
 4:    BRA.A BB128
 5:  BB3:
 6:    CMP R2, R3
 7:    BEQ.W BB1
 8:  BB4:
 9:    CMP R4, R5
10:    BEQ.S BB7
11:    BRA.B BB2
12:  BB6:
13:    CMP 65536, R3
14:  BB7:
15:    BEQ.W BB256
```
(b) 適用後

図 1　Branch chaining
Fig. 1　Branch chaining.

表 1　RX マイコンの分岐命令のサイズ指定子
Table 1　RX microcontrollers' branch instruction size specifiers.

サイズ指定子	分岐命令先頭から分岐先までの距離（byte）	命令サイズ（byte）	付与可能な分岐命令
.S	3 から 10	1	BRA,BEQ,BNE
.B	-2^7 から 2^7-1	2	全部
.W	-2^{15} から $2^{15}-1$	3	BRA,BEQ,BNE
.A	-2^{23} から $2^{23}-1$	4	BRA

© 2021 Information Processing Society of Japan

せの中から，最良の組合せを選ぶ必要があるが，組合せの総数はしばしば膨大になり，最良の組合せを求めるのを困難にする．この困難の克服を目指し，Leverett らは最良の組合せを効率的に求めるアルゴリズムを提案している．

Leverett らのアルゴリズムは，効率化の手段として，サイズの小さな分岐命令の分岐可能な距離が短いことを利用する．すなわち，分岐命令に長いものと短いものの 2 種類があり，長いものはどこにも分岐でき，短いものが最大 s バイト分岐できるとおくとき，プログラムを s バイト単位に切り分けると，切り分けた個々の区画の中にある短い分岐命令は，隣接する区画には分岐できても，それより遠くには分岐できなくなる．このことは，ある区画の中にある分岐命令のサイズの影響を受ける分岐命令が，隣接する区画より遠くにはないことも意味しており，そこで Leverett らのアルゴズムでは，隣接する区画どうしで書き換えパターンの組合せのうち実施可能なものを選ぶ操作を繰返すことで，最良の組合せを求める．Leverett らのアルゴリズムの計算量は，$O(C^{2l}n)$ であり，ここで記号 C，l，n はそれぞれ次の数を表す．

C　分岐命令の書き換えパターンの数

l　個々の区画内に存在する分岐命令で，サイズの選択の対象とするものの数

n　プログラムを切り分けてできた区画の数

これらのうち C を Leverett らは 3 としている．その理由は，Leverett らのアルゴリズムでは分岐命令の書き換えパターンが次の 3 つだからである．

(1) 分岐先を書き換えず，書き換え前の分岐先までの直接分岐のまま
(2) 上流側の最近傍にある分岐先候補まで分岐する短い分岐命令への書き換え
(3) 下流側の最近傍にある分岐先候補まで分岐する短い分岐命令への書き換え

ここで分岐先候補は次のいずれかに該当する箇所である．

(1) 書き換え前の分岐先
(2) 分岐先が同一の分岐命令のうち，分岐元で分岐すれば必ず分岐するもの

たとえば図 1 の 10 行目にある条件分岐命令 `BEQ.W BB256` の分岐先候補になるのは，書き換え前の分岐先である `BB256` と，15 行目の条件分岐命令である．ここで 15 行目の条件分岐命令が分岐先候補になる理由は，10 行目の条件分岐命令と分岐先が同一で，なおかつ分岐する条件が，10 行目の条件分岐命令の分岐する条件を包含しており，10 行目で分岐すれば必ず分岐するといえるからである．

Leverett らのアルゴリズムの計算量 $O(C^{2l}n)$ の C^{2l} は隣接する区画ごとの計算量をあらわす．隣接する区画ごとの計算量が C^{2l} になる理由は，隣接する 2 つの区画の中にあり，サイズを選択する分岐命令の数が $2l$ で，そのそれぞれが C 通りの書き換えパターンを持つからである．

Leverett らのアルゴリズムの計算量 $O(C^{2l}n)$ は，プログラム中の全分岐命令のバリエーションの組合せを単純に全探索する場合の計算量 $O(C^{nl})$ よりは小さいが，それが現実的かはどうかの評価を Leverett らは示していない．また Leverett らは branch chaining がコードサイズや実行サイクルに与える影響も示していない．コードサイズに与える影響を示す文献として Goyle らによるものはあるが [15]，その影響がどのような branch chaining の実装によるものか明らかでない．そこで本論文では branch chaining の有用性を示すことを目的として，次のアルゴリズムによる branch chaining を RX マイコン向けに実装し，それぞれがビルド時間やコードサイズ，実行命令数に与える影響を示す．

(1) Leverett らのアルゴリズムを RX マイコン向けに拡張した，最適解を求めるアルゴリズム
(2) 近似解を求めるアルゴリズム

なお，本論文で示す実装は，一部，RX マイコン向けに特化しているが，多くは他のアーキテクチャにも適用できる汎用的なものである．

3. Leverett らのアルゴリズムの RX マイコン向け実装

Leverett らが示したアルゴリズムを RX マイコン向けに実装するにあたり，我々は次の変更を加えた．これらの変更について順に述べる．

(1) 条件分岐命令のサポートの強化
(2) 3 種類の命令サイズへの対応
(3) ビルド時間対策

3.1 条件分岐命令のサポートの強化

条件分岐命令のサポートの強化は，条件分岐が，条件分岐命令と無条件分岐命令の組合せでも実現できることから必要になるものである．表 1 に示すように，RX マイコンでサイズ指定子 `.W` を付与できる条件分岐命令は，`BEQ` と `BNE` だけであり，他の条件で遠方に分岐する場合，**図 2** に示すように，条件分岐命令と無条件分岐命令を組み合わせることになる．

遠方への条件分岐の実現が，条件分岐命令と無条件分岐命令の組合せになることは，RX マイコンに限らず，他のアーキテクチャでもあることで [1], [2], [3], [4], [5], [6]，さ

```
1:     BGE.B BB1
```
(a) 短距離

```
   1:     BLT.B BB2
   2:     BRA.W BB1
3:  BB2:
```
(b) 長距離

図 2　分岐先までの距離に応じた条件分岐の実現

Fig. 2　Conditional branches implementations for branch distance variations.

© 2021 Information Processing Society of Japan

	(a) 最適化前		(b) 最適化後
1:	BGT.? BB1	1:	BGT.B BB2
2:	···	2:	···
3:		3:	BB2:
4:	BNE.? BB1	4:	BEQ.S BB4
5:		5:	BB3:
6:		6:	BRA.W BB1
7:		7:	BB4:
8:	···	8:	···
9:	BLE.? BB1	9:	BLE.B BB2

図 3 条件分岐の分割による分岐先候補の増加
Fig. 3 Branch destination increase by split branches.

らに，遠方への条件分岐命令を持つアーキテクチャでも，条件分岐の実現を，条件分岐命令と，無条件分岐命令の組合せにできる．ここで条件分岐の実現を，条件分岐命令と無条件分岐命令の 2 命令にすると，無条件分岐命令が，他の分岐命令の分岐先の候補になり，最適解に影響することに注意が必要である．

たとえば**図 3** (a) の中間表現について考える．図 3 (a) は branch chaining を適用する前のものであり，その 1，4，9 行目に同一の分岐先 BB1 への条件分岐が存在する．図 3 (a) では 1，4，9 行目の条件分岐のサイズ指定子をいずれも?と表記しているが，これは branch chaining を適用する前の段階では分岐先までの距離が未定で，したがってサイズ指定子も未定であることを意味する．図 3 (a) の 1，4，9 行目の条件分岐の分岐条件については，4 行目の分岐条件が成立すれば 1 行目の分岐条件は成立し，したがって 1 行目の条件分岐は 4 行目の条件分岐の分岐先の候補になるが，他に一方が他方の分岐先の候補になる関係は存在しない．ただし，条件分岐の実装を条件分岐命令と無条件分岐命令の組合せにすればその限りでなく，たとえば図 3 (a) の 4 行目にある BNE.?の実現を，図 3 (b) の 4 から 6 行目に示すように，条件分岐命令と無条件分岐命令の組合せにすると，無条件分岐命令が 1，9 行目の分岐先の候補になる．また，図 3 (b) の 4 行目の条件分岐命令は，1 行目の条件分岐命令の分岐先の候補にもなる．なぜなら 4 行目の条件分岐命令は 1 行目の条件分岐命令の分岐条件が成立したときには決して分岐せず，したがって 1 行目から 4 行目に分岐すると，次に実行するのは必ず 5 行目の無条件分岐命令になるからである．1 行目の条件分岐命令のサイズを小さくするうえで，その分岐先を，5 行目の無条件分岐命令より近い 4 行目の条件分岐命令にできることは有益である．

3.2 3 種類の命令サイズへの対応

3 種類の命令サイズへの対応が必要になる理由は，Leverett らのアルゴリズムが分岐命令を長短 2 種類と想定したものであるのに対し，RX マイコンが，表 1 に示すように，より多くの種類の分岐命令をサポートするためである．本論文の実装では表 1 のサイズ指定子のうち S，B，W の 3 種類をサポートした．サイズ指定子 A は未サポートだが，A が必要になるのは大規模なアプリケーションのコンパイル時のみである．本論文の 5 章で branch chaining の評価に使うアプリケーションはいずれもサイズ指定子 A を必要としないので，A はサポートせずとも最適解を得られる．

我々の実装では，Leverett らのアルゴリズムでいうところの短い/長い分岐命令をそれぞれサイズ指定子.B/.W の命令と見なし，サイズ指定子.B の分岐命令の分岐距離の絶対値の最大値である 128 バイト以上になるようプログラムを切り分ける．サイズ指定子.S の命令は短い分岐命令の一種と見なす．

3 種類の命令サイズと，条件分岐を条件分岐命令と無条件分岐命令に分割することをサポートするため，我々の実装では，分岐命令の書き換えパターンは**表 2** の 12 種類になる．ただし，書き換えパターンごとに適用可能な分岐命令は異なるため，適用できる書き換えパターンの数は，最も多い BEQ，BNE で 8 になり，したがって C の値は 8 になる．

表 2 書き換えパターン
Table 2 Branch instruction rewrite patterns.

パターン名	分岐命令先頭から分岐先までの距離 (byte)	命令サイズ (byte)	適用可能な分岐命令
.S0	3 から 10	1	BRA,BEQ,BNE
.S1	3 から 10	1+1	BEQ,BNE
.S2	3 から 10	2+1	BRA,BEQ,BNE 以外
.B0-	-2^7 から -1	2	全部
.B1-	-2^7 から -1	1+2	BEQ,BNE
.B2-	-2^7 から -1	2+2	BRA,BEQ,BNE 以外
.B0+	0 から 2^7-1	2	全部
.B1+	0 から 2^7-1	1+2	BEQ,BNE
.B2+	0 から 2^7-1	2+2	BRA,BEQ,BNE 以外
.W0	-2^{15} から $2^{15}-1$	3	BRA,BEQ,BNE
.W1	-2^{15} から $2^{15}-1$	1+3	BEQ,BNE
.W2	-2^{15} から $2^{15}-1$	2+3	BRA,BEQ,BNE 以外

```
1:  void chainBranches(){
2:    decideRewritePattern();
3:    splitBranchInstructions();
4:    rewriteBranchDestination();
5:  }
```

図 4 Leverett らのアルゴリズムの実装
Fig. 4 Our implementation of Leverett's algorithm.

3.3 ビルド時間対策

ビルド時間対策については，適用箇所が我々の実装の様々な箇所に分散していることから，我々の実装の全体を俯瞰しつつ，どこに対策を施したのか示す．

我々の実装を疑似コードで書いたものを**図 4** に示す．

© 2021 Information Processing Society of Japan

```
 1: void rewriteBranchDestination(){
 2:   for(all bb in 全基本ブロックの集合){
 3:     rewriteBranchDestinationForBasicBlock(bb);
 4:   }
 5: }
 6:
 7: // 分岐先が bb の分岐命令の分岐先を書き換える
 8: void rewriteBranchDestinationForBasicBlock(基本ブロック bb){
 9:   書き換え済の分岐命令を格納するベクタ workList;
10:   for(all branchInst in bb への分岐命令の集合){
11:     if (branchInst が割り当てられたサイズ指定子で bb に分岐できる){
12:       branchInst の分岐先を bb に確定;
13:       workList.push_back(branchInst);
14:     }
15:   }
16:   for(unsigned i = 0; i < workList.size(); ++i){
17:     branchInst_i = workList[i];
18:     for(all branchInst_j in bb への分岐命令のうち branchInst_i を経由しうるものの集合){
19:       if (branchInst_j の分岐先が確定済でも書き換え済でもない &&
20:           branchInst_j が割り当てられたサイズ指定子で branchInst_i に分岐できる){
21:         branchInst_j の分岐先を branchInst_i に書き換え;
22:         workList.push_back(branchInst_j);
23:       }
24:     }
25:   }
26: }
```

図 5 分岐先の書き換え

Fig. 5 Branch destination decision.

本論文の疑似コードの構文は C++の構文に準ずる．図 4 ではまず，2 行目で個々の分岐に表 2 のどの書き換えパターンを適用するのかを定める．2 行目で定めた書き換えパターンの中には，条件分岐を，条件分岐命令と無条件分岐命令の組合せに書き換えるよう要求するものもあるが，その書き換えを行うのが 3 行目である．最後に 4 行目で，分岐命令の分岐先を定める．

図 4 の 4 行目で呼び出す，分岐先を定める処理の実装を**図 5** に示す．図 5 の処理では，まず元々の分岐先に直接分岐できる分岐命令の分岐先を定め，次に，分岐先が定まった分岐命令に分岐できる分岐命令について，その分岐先を，分岐先が定まった分岐命令にすることを，全分岐命令の分岐先が確定するまで繰り返す．

さて，図 4 と図 5 の実装は，図 4 の 3 行目を除けば Leverett らのアルゴリズムそのものであり，また図 4 の 3 行目の処理は条件分岐の条件分岐命令と無条件分岐命令の組合せによる実現の一部であって，我々が実装したビルド時間対策ではない．我々が実装したビルド時間対策は，図 4 の 2 行目で呼び出す処理の中にある．図 4 の 2 行目で呼び出す，各分岐に適用する書き換えパターンを定める処理の実装を**図 6** に示す．

```
1: void decideRewritePattern(){
2:   assignRewritePatternCandidates();
3:   eliminateUnavailavleCandidates();
3:   splitIntoGroups();
4:   selectRewritePattern();
5: }
```

図 6 書き換えパターンの決定

Fig. 6 Decision of the rewrite pattern.

図 6 では，まず 2 行目で，個々の分岐に適用する書き換えパターンの候補を割り付ける．ここで割り付ける書き換えパターンの候補は，分岐のオペコードのみを根拠として定め，たとえば無条件分岐には，適用する書き換えパターンの候補として，`.S0`/`.B0-`/`.B0+`/`.W0` の 4 種類を一律に割り付ける．ここで割り付ける候補の中には，分岐先の候補までの距離を考えれば適用しえないものも存在するが，距離などの条件を考慮して候補を絞り込む処理は次の 3 行目で行う．3 行目では個々の分岐に，次の 3 つの処理の適用し，適用する書き換えパターンの候補を絞り込む．

(1) 分岐の分岐先候補のうち，書き換え前の分岐先以外のものから，分岐先の変更によって分岐のサイズを小さくしえないものを除外する．

具体的には，書き換え前の分岐先までの距離を悲観的に見積もって，分岐に必要なサイズ指定子を求めた結

果と，書き換え前の分岐先以外の分岐先の候補までの距離を楽観的に見積もって，分岐に必要なサイズ指定子を求めた結果を比較し，後者の方が分岐のサイズを小さくするのでなければ，後者は分岐のサイズを小さくしえないので，候補から除外する．

(2) 分岐に適用しうる書き換えパターンを，分岐先の候補までの方向および距離の上限と下限に応じて定める．ここではたとえばサイズ指定子.Bを指定すればどの分岐先の候補にも到達できると分かった分岐について，その書き換えパターンの候補から，サイズ指定子.Wに対応するものを除外するといった処理を行う．分岐がとりうるサイズの下限は，分岐先の候補までの距離を楽観的に見積った場合の分岐のサイズのうち，最も小さなものと定め，上限は，悲観的に見積った場合の分岐のサイズのうち，最も大きなものと定める．

(3) 条件分岐のうち，他の分岐の分岐先の候補になっていないものについては，条件分岐命令と無条件分岐命令に分割しても得にならないため，適用しうる書き換えパターンから，必須でない分割をともなうものを除外する．

ここで，分岐先までの距離を楽観的に見積るとは，分岐先に到達するまでの間に飛び越える，サイズ指定子が未定の分岐命令に，それが保持する書き換えパターンの候補のうち，そのサイズを最も小さくする書き換えパターンを適用すると想定して見積もることを表すものとし，逆に，最も大きくする書き換えパターンを適用すると想定して見積もることを，悲観的に見積もると表すものとする．

3 行目の処理で分岐に適用する書き換えパターンの候補を絞り込むと，分岐がとりうるサイズの範囲が狭くなり，結果として，分岐をまたぐ分岐に適用する書き換えパターンの候補をさらに絞り込める場合が生じる．そこで，分岐がとりうるサイズの範囲が狭くなった場合には，分岐をまたぐ分岐を再度絞り込みの対象とする．とりうるサイズが唯一になった分岐については，3 行目の時点で，適用する書き換えパターンの検討対象から除外する．

3 行目の処理が終ったら 4 行目に進み，プログラムを前から順に 128 バイト以上の連続する区画に分割し，1 から n までの番号を付与する．分割は基本ブロック単位で実施し，したがって区画のサイズは 128 バイトを超えることもある．

4 行目ではプログラム全体のコードサイズが最小になるよう個々の分岐の書き換えのパターンを定める．図 6 の中の，我々の実装に固有なビルド時間対策は，3 行目で適用する書き換えのパターンの候補を絞り込むところにある．

図 6 の 4 行目で呼び出す処理の詳細を**図 7** に示す．図 7 の 1 行目で定義している配列 path はプログラムを分割した区画の数が n であるとき $n+2$ 個のベクタを持ち，その i 番目のベクタは，プログラムの i 番目の区画の中にある，サイズ指定子が未定の分岐の書き換えパターンの全組合せ Σ_i の数だけの要素を持つものとする．ここで 0 番目と $n+1$ 番目の区画はプログラム全体の前後に対応する区画を表現するものとし，0 番目と $n+1$ 番目の区画も含め，サイズ指定子が未定の分岐を持たない区画の Σ は空集合でなく，選ぶものが何もないことを表現する要素を 1 つ持つ集合とし，したがってその区画に対応するベクタは唯一の要素を持つものとする．配列 path の i 番目のベクタの j 番目の要素の役割は，i 番目の区画の j 番目の書き換えパターンを選択したときに，i 番目の区画の終端までの距離を最小にするには，$i-1$ 番目の区画の何番目の書き換えパターンを選択すればよいのかを記憶することにある．図 7 の処理は，配列 path の 0 番目のベクタから順に，その内容を埋めてゆくものであり，最後に $n+1$ 番目のベクタの内容を埋めた時点で，個々の区画に何番目の書き換えパターンを適用すれば，プログラム全体のサイズを最小にできるかが確定する．確定したパターンを確認するには，最後に

```
 1: std::vector<uint64_t> path[n + 2];
 2:
 3: typedef std::multimap<uint64_t, unsigned>
 4:   PriorityQueue;
 5:
 6: void selectRewritePattern(){
 7:   path[0][0] = end;
 8:
 9:   PriorityQueue combIDs[2];
10:   combIDs[0].insert({{ 0, 0 }});
11:   for(i=1; i<=n+1; ++i){
12:     combIDs[i&1].clear();
13:     for(j=0; j<sizeof(Σ_i); ++j){
14:       assumeRewritePattern(i, Σ_i[j]);
15:       if (!compatible(any, Σ_i^intra[j])){
16:         continue;
17:       }
18:       for(all p in combIDs[1-(i&1)]){
19:         unsigned k = p.second;
20:         assumeRewritePattern(i − 1, Σ_{i−1}[k]);
21:         if (compatible(backward, Σ_i^inter[j]) &&
22:             compatible(forward, Σ_{i−1}^inter[k])){
23:           unsigned size =
24:             estimateSize(i, Σ_i[j]) + p.first;
25:           combIDs[i&1].insert({ size, j });
26:           path[i][j] = k;
27:           break;
28:         }
29:       }
30:     }
31:   }
32: }
```

図 7　書き換えパターンの選択
Fig. 7　Rewrite pattern selection.

© 2021 Information Processing Society of Japan

埋めた path[$n+1$] から逆順に配列 path をたどればよい．具体的には，最初に見る path[$n+1$] については，その構成要素は 1 つだけなので，path[$n+1$][0] の内容を確認する．このとき path[$n+1$][0] の値がたとえば 3 だったら，n 番目の区画には 3 番目の書き換えパターンを適用すればよいということになり，さらに path[n][3] の値が 2 であれば，$n-1$ 番目の区画には 2 番目の書き換えパターンを適用すればよいということになる．この操作を，配列 path の最初の要素まで繰り返せば，全区画に適用すべき書き換えパターンを得られる．

図 7 の 3 から 4 行目で定義している型 PriorityQueue は，符号なし整数のペアを，ペアの第 1 要素を優先順位と見なして昇順に並べるものである．6 行以降の処理では PriorityQueue を使って，第 1 要素がプログラムのサイズ，第 2 要素が書き換えパターンの識別子のペアを，第 1 要素の小さなものから順に並べ替える．

図 7 の 6 行目以降では次の要領で最適解を得る．まず 7 行目で path の 0 番目を終端を表わす値 end で初期化し，続く 9 行目で優先順位つき待ち行列を 2 つ用意する．これらの待ち行列は，11 から 24 行目のループで，$i-1$ 番目と i 番目の区画に適用する書き換えパターンの組合せを検討する際に，一方は $i-1$ 番目の区画の書き換えパターンの組合せを走査する順番を保持し，もう一方には i 番目の区画の書き換えパターンの組合せを次の周回で走査する順番を記録する．10 行目では待ち行列の一方に値 {{0, 0}} を代入しているが，これは 11 から 24 行目のループを最初に周回する際に，0 番目の区画の書き換えパターンの組合せを走査する順番である．ここで代入する値 {{0, 0}} の 1 つ目の 0 はプログラムの先頭から 0 番目の区画の終端までの距離が 0 バイトであることを表し，2 つ目の 0 は，0 番目の区画の書き換えパターンの組合せとして唯一存在するもの，すなわち選ぶものが何もないことを表現する要素の識別子を表す．

11 から 31 行目のループでは，まず 12 行目で，2 つの優先順位つき待ち行列うち，次の周回で走査する順番を記録する方を空にし，続く 13 行目から 30 行目のループで i 番目の区画の書き換えパターンのそれぞれについて，$i-1$ 番目の区画のどの書き換えパターンとなら組合せ可能かを調べ，そのなかで最も i 番目の区画の終端までのサイズを小さくできるものを求める．13 行目から 30 行目のループの内側にある，18 行目から 29 行目のループでは，i 番目の区画の j 番目の書き換えパターンと組み合わせて，i 番目の区画の終端までのサイズを最小にできる，$i-1$ 番目の区画の書き換えパターンを求める．18 行目から 29 行目のループは候補を最初に見つけた時点で終了になるが，最初に見つけた時点で終了にできる理由は，$i-1$ 番目の区画の書き換えパターンを走査する順番を，$i-1$ 番目の区画の終端までのサイズを小さくするものから順にしてあり，したがって最初に見つけた候補が i 番目の区画の終端までのサイズを最小にするといえるからである．

13 行目から 30 行目のループでは，14 行目で i 番目の区画の書き換えパターンを $\Sigma_i[j]$ と想定し，15 行目で $\Sigma_i[j]$ を構成する個々の分岐の書き換えパターンのうち，分岐先の候補がすべて i 番目の区画の中にあるものの集合 $\Sigma_i^{intra}[j]$ を対象に，書き換えパターンの適用の可否，すなわち書き換えパターンに従って書き換えた分岐命令が分岐先に到達できるか否かを判定し，不可なら書き換えパターン $\Sigma_i[j]$ の適用を諦め，可なら 18 行目から 29 行目のループに進む．

18 行目から 29 行目のループでは，$i-1$ 番目の区画の書き換えパターンを，$i-1$ 番目の区画までのプログラムのサイズを小さくするものから順にたどり，たどった書き換えパターン $\Sigma_{i-1}[k]$ と，i 番目の区画の書き換えパターン $\Sigma_i[j]$ を組み合わせて適用することが可能かどうか 21 行目と 22 行目で判定する．21 行目では，i 番目の区画に書き換えパターン $\Sigma_i[j]$ を適用するできるか否かを検証するが，ここで検証を行う関数 compatible() に渡す 1 つ目の引数 $backward$ はオフセットが負，すなわち $i-1$ 番目の区画に向かうの分岐のみを検証の対象とすることを表す．2 つ目の引数 $\Sigma_i^{inter}[j]$ は，書き換えパターン $\Sigma_i[j]$ から，分岐先の候補が i 番目の区画の外にもある分岐向けのものを抽出したものだが，ここで $\Sigma_i[j]$ でなく $\Sigma_i^{inter}[j]$ を渡す理由も，i 番目の区画から $i-1$ 番目の区画に向かう分岐の書き換えパターンのみを検証の対象にするためである．22 行目では $i-1$ 番目の区画に書き換えパターン $\Sigma_{i-1}[k]$ を適用できるか検証し，$\Sigma_i[j]$ と $\Sigma_{i-1}[k]$ の双方を適用できる場合には 23 から 25 行目に進み，次の周回で j 番目の書き換えパターンを何番目に走査すべきかを優先順位つき待ち行列に記録し，続く 26 行目において，i 番目の区画の j 番目の書き換えパターンとの組合せで i 番目の区画の終端までのサイズ最小にできるのは $i-1$ 番目の区画の k 番目の書き換えパターンであると path に登録し，27 行目でループから抜ける．

図 7 の中にある，我々の実装に固有なビルド時間対策は，書き換えパターンの検証の冗長さを回避するための，次の措置である．

(1) 分岐先の候補がすべて区画内にある分岐と，区画外にもある分岐の書き換えパターンの検証を分離する．
(2) 区画外への分岐の書き換えパターンの検証において，$i-1$ 番目の区画内の分岐向けの書き換えパターンの検証では，i 番目の区画に向かうもののみ，i 番目の区画内の分岐向けの書き換えパターンの検証では，$i-1$ 番目の区画に向かうもののみを対象とする．
(3) 優先順位つき待ち行列を使って，18 行目から 29 行目のループの周回数を減らす．

次に図 7 の 15, 21, 22 行目で呼び出す関数 compatible() の実装を**図 8** に示す．図 8 の処理では，仮引数として受け

© 2021 Information Processing Society of Japan

```
 1:  bool compatible(検証する分岐の方向 direction,
 2:                  RewritePattern p̄){
 3:    for(all p in p̄){
 4:      if (p に対応するサイズ指定子が.W でない &&
 5:          p に対応する分岐の方向が direction と一致 &&
 6:          書き換えパターンに対応する分岐距離の範囲に
 7:          次のいずれかに該当する分岐先の候補がない
 8:          - 書き換え前の分岐先
 9:          - 分岐の方向が一致
10:          - サイズ指定子がより大きい)
11:        return false;
12:      }
13:    }
14:    return true;
15:  }
```

図 8　書き換えパターンの適用可否の判定
Fig. 8　Compatibility test.

取った $\bar{p}$ の構成要素である，個々の分岐の書き換えパターン p の中に，適用不能なものがあるか，4 から 10 行目で調べ，あるなら 11 行目に進み，適用不能であることを表現する値 `false` を返戻し，ないなら 14 行目に進んで，適用可能であることを表現する値 `true` を返戻する．

4 から 10 行目では，4 行目で，書き換えパターンに対応するサイズ指定子が.W か調べ，.W なら書き換え前の分岐先まで分岐できると見なして検証を終える．.W でないなら 5 行目に進んで分岐の方向を調べ，分岐の方向が仮引数 *direction* と一致する場合のみ 6 行目から先に進むものとし，6 から 10 行目では，書き換えパターンに対応する，表 2 記載の分岐距離の範囲内に分岐先の候補があるか検証する．検証にあたり，我々の実装では，条件分岐が分岐先の候補になるか否かが，一定とは限らず，条件分岐を条件分岐命令と無条件分岐命令に分割するか否かに依存しうることへの配慮が必要になる．

4. 近似解を求めるアルゴリズム

Leverett らのアルゴリズムには，最適解を求められる利点はあるものの，計算量が大きくなる懸念もある．そこで本論文では，より小さな計算量で近似解を求めるアルゴリズムを 2 つ提案する．提案する 2 つのアルゴリズム，すなわち第 1 のアルゴズムと，第 2 のアルゴリズムは，計算量の軽減を目的とした，次の共痛点を持つ．

(1) 条件分岐命令と無条件分岐命令に分割すれば分岐先の候補になる分岐は，分岐先の候補と見なさない．
(2) 分岐先の候補までの距離を求める際に，途中にある分岐のサイズを一律 3 バイトで見積もる．

提案する 2 つのアルゴリズムは，どちらも**図 9** の処理によって近似解を求める．図 9 の 2 から 4 行目のループで個々の基本ブロック *bb* を巡回し，3 行目で，分岐先が *bb* の分岐命令の集合ごとに，近似解を求める．ここで我々が

```
1:  void chainBranches(){
2:    for(all bb in 全基本ブロック){
3:      findSuboptimalSolution(bb);
4:    }
5:  }
```

図 9　近似解を求めるアルゴリズムの共通部
Fig. 9　Common part of algorithms for suboptimal solution.

```
1:  void findSuboptimalSolution(基本ブロック bb){
2:    createDirectedGraph(bb);
3:    findMinimumCostArborescence();
4:    rewriteBranchDestinationForBasicBlock(bb);
5:  }
```

図 10　近似解を求める第 1 のアルゴリズム
Fig. 10　The 1st algorithm for suboptimal solution.

提案する 2 つのアルゴリズムの違いは，3 行目の処理の実装である．本章では 4.1 節で第 1 の，4.2 節で第 2 のアルゴリズムについて述べ，4.3 節でそれぞれの計算量を示す．

4.1 第 1 のアルゴリズム

我々が提案する第 1 のアルゴリズムは，近似解を求める問題を，最小全域有向木を求める問題として解決する．

第 1 のアルゴリズムにおける図 9 の 3 行目の処理の実装を**図 10** に示す．図 10 では，まず，2 行目で問題を表現する有向グラフを構築する．構築する有向グラフのノードは，分岐命令と，書き換え前の分岐先である．エッジは，分岐先の候補から分岐命令に向けて張る．エッジのコストは分岐先の候補まで分岐する際の分岐命令のサイズとする．有向グラフの構築の処理を**図 11** に示す．図 11 の処理では 2 から 5 行目でノードを作成し，6 から 19 行目でエッジを張る．エッジを張る処理の 9 行目では，書き換え前の分岐先への分岐に対応するエッジを張るが，それ以降は，9 行目で張ったものよりコストが小さなエッジのみを張る（14 から 16 行目）．ここでコストが同等以上のエッジを張らない理由は，書き換え前の分岐先に分岐できるなら書き換え前の分岐先に分岐するのが正解であり，したがって 9 行目より先の処理で，コストが同等以上のエッジを張っても，解の改善に貢献しないからである．

図 11 の処理で構築する有向グラフの例を**図 12** (b) に示す．図 12 (b) は図 12 (a) の中間表現から構築したもので，図 12 (b) の中の丸はノード，丸の中の数字はノードが中間表現の何行目に対応するか，矢印はエッジ，矢印の上にある，四角で囲った数字はエッジのコストをそれぞれ表わす．図 12 (b) には 6 行目の分岐命令から 3 行目の分岐命令への分岐に対応するエッジ，すなわち 3 行目の分岐命令に対応するノードから 6 行目の分岐命令に対応するノードへのエッジがないが，その理由は，6 行目から 3 行目に分岐する場合の分岐命令のサイズが，6 行目から書き換え前の

© 2021 Information Processing Society of Japan

```
 1:  void createDirectedGraph(基本ブロック bb){
 2:    CreateNode(bb);
 3:    for(all branchInst in bb への分岐命令){
 4:      CreateNode(branchInst);
 5:    }
 6:    for(all bi in bb への分岐命令){
 7:      unsigned maxSize =
 8:        bb に分岐する際の bi のサイズ;
 9:      AddEdge(bb, bi, maxSize);
10:      for(all bj in bb への分岐命令){
11:        if (bj は bi の分岐先の候補){
12:          unsigned size =
13:            bj に分岐する際の bi のサイズ;
14:          if (size < maxSize){
15:            AddEdge(bj, bi, size);
16:          }
17:        }
18:      }
19:    }
20:  }
```

図 11　有向グラフの構築
Fig. 11　Creation of directed graph.

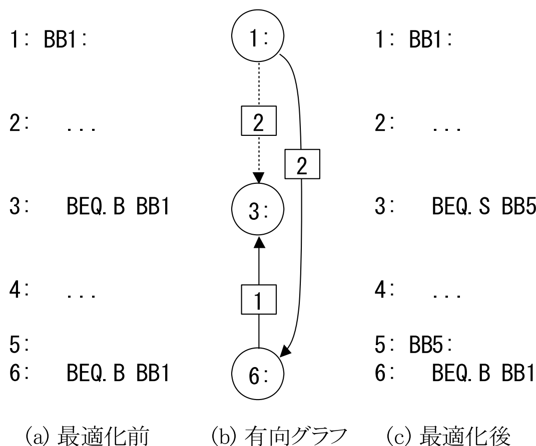

図 12　第 1 のアルゴリズムによる求解
Fig. 12　Solution by the 1st algorithm.

分岐先である 1 行目に分岐する場合の分岐命令のサイズと同じであり，したがって 6 行目から 3 行目への分岐に対応するエッジが解の改善に貢献しないからである．

図 12 (b) の矢印には線の部分が実線のものと破線のものがあるが，実線のものは，書き換え前の分岐先に対応するノードを根とする最小全域有向木の構成要素である．この最小全域有向木を求めるのが図 10 の 3 行目の処理である．最小全域有向木を求めるアルゴリズムはいくつかあるが [16], [17], [18], [19], [20]，我々の実装では Tarjan のアルゴリズム [18], [19] を使用する．最小全域有向木を求めたら，4 行目に進んで分岐命令の分岐先を書き換える．4 行目で呼ぶ関数 `rewriteBranchDestinationForBasicBlock()` は図 5 の 8 から 26 行目に示したものであり，その 11 行目と 20 行目で参照する分岐命令のサイズ指定子は，最小全域有向木を構成するエッジのコストに応じて定まるものとし，たとえば図 12 (b) の最小全域有向木に従って書き換えを行うと図 12 (c) の結果を得る．

4.2　第 2 のアルゴリズム

我々が提案する第 2 のアルゴリズムは，分岐命令から，書き換え前の分岐先の方向に向かって最も近くにある分岐先の候補までの距離に応じてサイズ指定子を決定する．

第 2 のアルゴリズムにおける図 9 の 3 行目の処理の実装を**図 13** に示す．図 13 では，まず 2 から 15 行目で，分岐先が *bb* の分岐命令を含む基本ブロックを，*bb* より上流に位置するものと下流に位置するものに分け，それぞれを *bb* からみて遠くに位置するものから順に並べる．続く 16 から 48 行目では，上流側と下流側のそれぞれについて，遠くに位置する基本ブロックから順に，その末尾に位置する分岐命令の分岐先の変更を試みる．具体的には，21 から 29 行目で分岐命令から *bb* に向かって最も近傍に位置する分岐先の候補の分岐命令を求める．ここで候補を求められなかった場合と，求めた候補までの分岐に必要なサイズ指定子 d が，*bb* までの分岐に必要なサイズ指定子と同じ場合，32 行目で最適化を諦める．最適化を諦めなかった場合は，35 から 44 行目に進み，サイズ指定子 d で分岐可能な範囲に，他の候補があるか調べ，あるなら，それらのうち最も *bb* 寄りに位置するものを，なければ，21 から 29 行目で見つけた候補を分岐先と定め，46 行目で分岐先を変更する．

第 1 のアルゴリズムと第 2 のアルゴリズムの主な違いは，書き換え前の分岐先からいったん遠ざかることができるか否かである．RX マイコンが提供する，最もサイズの小さな分岐命令は正方向にしか分岐できないので，その活用に際しては，書き換え前の分岐先から遠ざかる方向に分岐する必要が生じることもあり，たとえば図 12 の最適化前後の中間表現を比較すると，3 行目の分岐が，最適化後に，遠ざかる方向の分岐に書き換わっていることが分かる．ここで第 1 のアルゴリズムはいったん遠ざかることを許容するので図 12 (c) の最適化結果を得られるが，第 2 のアルゴリズムは許容しないので図 12 (c) の最適化結果を得られない．このように第 2 のアルゴリズムは最適化の効果の点では第 1 のアルゴリズムに劣るが，第 2 のアルゴリズムには実装の簡素さという利点がある．第 2 のアルゴリズムの実装の実装規模は，たとえば図 13 が 49 行にあることにみるように小さく，第 1 のアルゴリズムが有向グラフの構築処理や最小全域有向木のソルバを必要とするのに比べ簡素である．ただし最小全域有向木のソルバを含めた第 2 のアルゴリズムもそれほど複雑ではなく，3 章で示した Leverett らのアルゴリズムの RX マイコン向け実装に比べれば小さな規模に収まる．

© 2021 Information Processing Society of Japan

```
 1: void findSuboptimalSolution(基本ブロック bb){
 2:   // bb の先行基本ブロックを，bb より上流側に位置するものと
 3:   // 下流側に位置するものに分け，遠いものから順にベクタに格納
 4:   std::set<基本ブロック> 上流側先行基本ブロック群，下流側先行基本ブロック群;
 5:   for(all pbb in bb の先行基本ブロック){
 6:     if ((pbb == bb) || (pbb の末尾に bb への分岐がない)){
 7:       continue;
 8:     }
 9:     if (pbb は bb より上流側に位置する){
10:       pbb を上流側先行基本ブロック群に bb から遠いもの順に挿入;
11:     }
12:     else{
13:       pbb を下流側先行基本ブロック群に bb から遠いもの順に挿入;
14:     }
15:   }
16:   // 上流側と下流側のそれぞれについて，遠いものから順に分岐先の変更を試みる
17:   for(each pbbs of { 上流側先行基本ブロック，下流側先行基本ブロック }){
18:     for(std::set<基本ブロック>::iterator pbbi = pbbs.begin(), pbbe = pbbs.end(); pbbi != pbbe; ++pbbi){
19:       bi = 基本ブロック pbbi の末尾にある分岐命令;
20:       // bi の分岐先になりうる最近傍の分岐命令を探す
21:       std::set<基本ブロック>::iterator pbbj = pbbi;
22:       サイズ指定子 d;
23:       while(++pbbj != pbbe){
24:         bj = 基本ブロック pbbj の末尾にある分岐命令;
25:         if (bi の分岐先を bj に変更できる){
26:           d = bi から bj に分岐するのに必要なサイズ指定子;
27:           break;
28:         }
29:       }
30:       if ((pbbj == pbbe) || // 見つからなかった
31:           (d == bi から bb まで分岐するのに必要なサイズ指定子)){
32:         continue; // 最適化を諦める
33:       }
34:       // 最近傍までと同じサイズ指定子で分岐できる最遠の分岐命令を探す
35:       std::set<基本ブロック>::iterator pbbk = pbbj;
36:       while(++pbbk != pbbe){
37:         bk = 基本ブロック pbbk の末尾にある分岐命令;
38:         if (bi の分岐先を bk に変更できる){
39:           if (d != bi から bk に分岐するのに必要なサイズ指定子){
40:             break; // サイズ指定子が同じでなくなったので探すのを止める
41:           }
42:           pbbj = pbbk;
43:         }
44:       }
45:       // 分岐先を変更する
46:       基本ブロック pbbi の末尾の分岐命令の分岐先を基本ブロック pbbj の末尾の分岐命令に変更;
47:     }
48:   }
49: }
```

図 **13** 近似解を求める第 2 のアルゴリズム

Fig. 13 The 2nd algorithm for suboptimal solution.

© 2021 Information Processing Society of Japan

4.3　近似解を求めるアルゴリズムの計算量

我々が提案する第1，第2のアルゴリズムの計算量は，プログラム中の分岐命令の数を b とおくとき，ともに $O(b^2)$ である．そうなる理由について，第1，第2のアルゴリズムの共通部である図 9 の処理を起点に考える．

図 9 は関数 `findSuboptimalSolution()` を個々の基本ブロックに適用する処理であり，したがってその計算量は，関数 `findSuboptimalSolution()` の計算量の総和といえるが，この総和の大小はプログラム中にある分岐命令の分岐先に依存する．その理由は，後述するように，関数 `findSuboptimalSolution()` の計算量が，引数として受け取る基本ブロックを分岐先とする分岐命令の数に依存するためである．総和が最大になるのは，プログラム中の分岐命令の分岐先がすべて同じ場合である．このとき，分岐先以外の基本ブロックを対象とする関数 `findSuboptimalSolution()` の計算量は，当該基本ブロックを分岐先とする分岐命令，すなわち最適化対象が存在しないことから，無視できるほど小さくなるので，計算量の総和は，b 個の分岐命令の分岐先の基本ブロックに，関数 `findSuboptimalSolution()` を適用する際の計算量として求められる．

ここで図 10 に示した第 1 のアルゴリズムの関数 `findSuboptimalSolution()` の計算量について考える．第 1 のアルゴリズムの関数 `findSuboptimalSolution()` は，図 10 の 2 から 4 行目の 3 つの処理からなるので，その計算量は 3 つの処理の計算量の総和になる．3 つの処理の計算量について順に考える．

- 2 行目の処理は有向グラフを構築する処理であり，有向グラフの構築にあたっては図 11 の 6 から 19 行目にあるように b 個の分岐命間の関係を調査することになり，その計算量は $O(b^2)$ になる．
- 3 行目の最小全域有向木を求める処理の計算量は，我々が採用した Tarjan のアルゴリズムでは，一般には，有向グラフのノードの数を v，エッジの数を e とおくとき，$O(min\{e \log v, v^2\})$ になる [18]．ここで有向グラフのノードは分岐命令と書き換え前の分岐先に対応して作るものなので，その数 v は $b+1$ に等しく，エッジは最悪の場合，個々の分岐に対応するノードに，自身以外の分岐に対応するノード全部と，書き換え前の分岐先に対応するノードから張ることになるので，その総数 e は b^2 になる．e が b^2，v が $b+1$ であるとき，$O(e \log v) > O(v^2)$ なので，3 行目の処理の計算量は $O(v^2) \simeq O(b^2)$ になる．
- 4 行目は有向グラフの根からエッジ一式を順にたどる処理なので，その最悪の計算量は，エッジの最大数 b^2 により，$O(b^2)$ になる．

すなわち，2 から 4 行目の計算量はいずれも $O(b^2)$ で，したがって第 1 のアルゴリズムの関数 `findSuboptimalSolution()` 全体の計算量も，第 1 のアルゴリズム全体の計算量も $O(b^2)$ と見なすことができる．

次に，図 13 に示した，第 2 のアルゴリズムの関数 `findSuboptimalSolution()` の計算量について考える．図 13 の処理は大きく次の 2 つに分けることができる．

- 1 つは 2 から 15 行目にある，分岐命令を，書き換え前の分岐先から遠い順に並べ替えるループで，その計算量は $O(b \log b)$ である．
- もう 1 つは 16 から 48 行目にある，分岐先をどこに変更するかを決める 2 重ループで，その計算量は $O(b^2)$ である．

これらの計算量のうち，支配的なのは 16 から 48 行目の 2 重ループもので，したがって第 2 のアルゴリズムの関数 `findSuboptimalSolution()` 全体の計算量も，第 2 のアルゴリズム全体の計算量も $O(b^2)$ と見なすことができる．

5.　評価

3 章および 4 章で示したアルゴリズムの評価を目的として，それぞれをルネサスエレクトロニクスのコンパイラ製品 CC-RX V3.02 に追加実装した．CC-RX V3.02 は llvm-2.3 をベースに，ルネサスエレクトロニクスの独自，あるいは llvm の後々のリリースの最適化を追加して開発したものである．

branch chaining を適用するタイミングに関し，Leverett らは，branch folding より後で適用すると，branch chaining の効果を大きくできると指摘しているが，我々の実装では，次に示す 2 つの理由から，branch folding を含め，他のあらゆる最適化より後にした．

(1) branch chaining を適用すると，ループの構造が崩れ，どこにループがあるか分からなくなるといった問題がおきるので，ループの構造を分析する最適化より前に適用するのは適切でない．
(2) 分岐以外の命令が確定した後でないと，厳密には，命令サイズに依存する最適化を適用できない．

これらの理由の (2) は，コンパイル時に branch chaining を適用するのは必ずしも適切でないことも表す．コンパイル時の適用が必ずしも適切でない理由は，リンク時の最適化によって，命令のサイズや，命令をどう整列するかが変化したり，命令が集約の対象になって移動したり [21] することがあるからである．本論文で示す評価結果は，コンパイル時に適用した場合のものだが，次のことはいえる．

- 評価の際にコンパイル対象としたプログラムは命令の整列を要求せず，したがって整列は結果に影響しない．
- 評価の際にコンパイル対象としたプログラムは小規模であり関数呼出しの命令 `call` のサイズをすべて最も小さなものとしてコンパイルできた．したがって，命令 `call` サイズはリンク時に変化しえず，結果に影響しない．

© 2021 Information Processing Society of Japan

なお我々の実装は関数内の分岐のみを最適化対象とし，関数呼出しを最適化対象としない．

評価の対象はコードサイズおよび実行命令数，最大経由数，ビルド時間とした．ここで最大経由数は，最適化により分岐が目的地に到達するまでにいくつの分岐命令を経由するようになったかの最大値を表わす．最適化のかからなかったプログラムでは，経由が発生しないので，最大経由数は 0 になる．branch chaining の実装の中には，性能劣化の回避を目的として，最大経由数に上限を設けるものもあるが [15]，我々が評価した実装は，いずれも上限を設けない．コンパイル対象のプログラムは，**表 3** に示す EEMBC [22] のプログラム [23], [24], [25], [26] とした．

表 3 のプログラムは組込機器の実行速度を評価するためのもので，現実的な組込機器向けアプリケーションから，実行速度の評価に適した部分を抜粋したものである．表 3 のプログラムのコンパイルに際しては，オプション`-cpu=rx600 -size -optimize=max -branch=16` を指定し，生成するコードは RX600 向けのものとし，コードサイズの削減を目的とする最適化を最大限適用した．オプション`-branch=16` は関数呼出しの命令のサイズをすべて最も小さなものにするためのものである．プログラムの実行は，シミュレータで実施したため，実行命令数は一意に定まった．ビルド時間については 6 回計測し，2 回目から 6 回目までの 5 回の計測結果の相乗平均を評価結果とした．

表 3　ベンチマークの内訳
Table 3　Benchmark items.

プログラム	代表的な処理
AutoBench 1.1	車内ネットワーク通信/エンジン制御
ConsumerBench 1.1	デジタルカメラの画像圧縮伸長
Networking 1.1	ネットワーク機器のパケット通信
OABench 1.1	プリンタの画像回転
TeleBench 1.1	モデムの通信

評価の結果を**表 4** に示す．表 4 の最適と近似は，それぞれ 3，4 章で示したアルゴリズムを表す．3 章で示したアルゴリズムについては，3.1 節で述べた，条件分岐を条件分岐命令と無条件分岐命令に分割する機能を除いたものも実装し，評価の対象とした．分割する機能の有無が branch chaining の有用性に与える影響について表 4 から考察する．まず，ビルド時間については，分割する機能がありだと最大 314 倍になっている一方で，なしなら近似解を求めるアルゴリズムと大差ない水準に収まっており，分割する機能がビルド時間に大きな影響を与えたといえる．一方で表 4 から，分割する機能がコードサイズの削減に貢献していることも確認できる．全ベンチマークのコードサイズの総和により，それぞれの削減率を求めると，分割する機能がありの場合が 0.40%，なしの場合が 0.36%で，分割する機能が branch chaining によるコードサイズの削減率を 1 割（$\frac{0.40}{0.36} - 1 \simeq 0.1$）ほど高めることが分かる．

なお，最適解を求めるアルゴリズムは，分割する機能なしでも，近似解を求めるアルゴリズムより多くのコードを削減する．近似解を求める第 1 および第 2 のアルゴリズムがもたらす削減率を，全ベンチマークのコードサイズの総和によって求めると，それぞれ 0.35%，0.33%で，いずれも最適解を求めるアルゴリズムがもたらす削減率に劣る．た

表 4　コードサイズ/実行命令数/ビルド時間への影響
Table 4　Effect on code size, instruction count and build time.

		コードサイズ削減率（%）				実行命令数増加率（%）				最大経由数				ビルド時間（倍）			
		最適		近似		最適		近似		最適		近似		最適		近似	
	サイズ	分割				分割				分割				分割			
プログラム	（KByte）	あり	なし	第一	第二	あり	なし	第一	第二	あり	なし	第一	第二	あり	なし	第一	第二
AutoBench 1.1																	
a2time	2.3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.02	1.02	1.00	1.00
aifftr	3.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	1.00	0.99	0.99
aifirf	1.7	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	1.00	0.99	0.99
aiifft	2.8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	0.99	0.99	0.99	1.00
basefp	1.3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	1.00	0.99	0.99
bitmnp	2.9	0.53	0.53	0.53	0.53	1.00	1.00	1.00	1.00	2	2	2	2	1.00	1.00	1.00	1.00
cacheb	2.6	0.19	0.19	0.19	0.19	0.00	0.00	0.00	0.00	5	5	6	6	1.03	1.01	0.99	0.99
canrdr	1.4	2.19	1.97	1.97	1.97	1.32	1.23	1.23	1.23	3	3	3	3	2.38	1.00	1.01	1.00
idctrn	3.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	0.99	0.99	0.99
iirflt	3.2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.01	1.01	1.00	0.99
matrix	4.0	0.23	0.08	0.08	0.05	0.00	0.00	0.00	0.00	1	1	1	1	1.01	1.00	0.99	1.00
pntrch	1.6	0.13	0.13	0.13	0.00	0.00	0.00	0.00	0.00	2	2	1	0	1.00	1.01	1.00	1.00
puwmod	2.1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	1.00	0.99	0.99
rspeed	1.1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	1.01	1.00	0.99
tblock	1.4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	1.01	1.00	1.00
ttsprk	4.8	0.25	0.25	0.19	0.19	0.02	0.02	0.02	0.02	6	6	7	7	1.00	1.00	0.99	0.99
ConsumerBench 1.1																	
cjpeg	158	0.22	0.19	0.18	0.16	0.01	0.01	0.00	0.00	4	5	3	3	1.03	1.00	1.01	1.00
djpeg	192	0.69	0.59	0.57	0.55	0.86	0.86	0.86	0.86	6	9	6	6	314	1.01	0.99	0.99
rgbcmy	0.56	0.36	0.36	0.36	0.36	0.00	0.00	0.00	0.00	1	1	1	1	1.00	1.01	0.99	0.99
rgbhpg	0.28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.01	1.00	1.00	1.01
rgbyiq	0.59	0.34	0.34	0.34	0.34	0.00	0.00	0.00	0.00	1	1	1	1	1.01	1.01	1.00	1.00
Networking 1.1																	
ospf	2.9	0.07	0.07	0.07	0.00	0.00	0.00	0.00	0.00	1	1	1	1	1.01	0.99	1.00	0.99
pktflow	1,3	0.30	0.15	0.15	0.08	0.00	0.00	0.00	0.00	1	1	1	1	1.00	1.01	1.01	1.00
routelookup	0.99	0.20	0.20	0.20	0.10	0.00	0.00	0.00	0.00	1	1	1	1	1.00	1.00	1.00	1.01
OABench 1.1																	
bezier	0.55	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	0.99	1.01	1.01	1.00
dither	0.51	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	1.01	1.01	1.00
rotate	1.4	5.80	5.80	5.80	5.80	0.14	0.14	0.06	0.12	6	13	4	7	0.99	1.01	1.00	1.00
text	1.5	1.52	1.52	1.52	1.25	1.19	1.19	1.39	0.53	4	4	2	2	68.6	1.00	1.01	1.00
TeleBench 1.1																	
autcor	0.34	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	0.99	1.00	1.00	0.99
conven	0.41	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.01	1.00	1.00	1.01
fbital	0.54	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.01	1.01	1.00	0.99
fft	1.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.00	1.00	1.00	1.00
viterb	0.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0	0	1.01	1.01	1.00	1.01

© 2021 Information Processing Society of Japan

だし，最適解を求めるアルゴリズムから分割する機能を除いたものとの比較では，効果の差が 1 割未満にとどまる．

実行命令数については，表 4 から，最適解を求めるアルゴリズムでも，近似解を求めるアルゴリズムでも，大きくは変らないことが分かる．劣化の相乗平均は，最適解を求める場合，分割の機能ありなら 0.14%，なしなら 0.13%であり，近似解を求める場合，第 1/第 2 のアルゴリズムでそれぞれ 0.14%/0.11%だった．

表 4 をみると，コンパイル対象のプログラムによって，branch chaining の効く度合に違いがあることが分かるが，よく効くプログラムには，多くの場合，制御フローをいろいろなところから 1 カ所に集める，たとえば次の記述があった．

- break 文を多く含む switch 文
- 論理 AND 演算子や論理 OR 演算子を多く含む式

EEMBC を使った評価の範囲では，最適解を求める場合であっても，分割する機能をなしにすれば，ビルド時間への影響を小さくできたが，我々が組込み機器のメーカから収集した実用アプリケーションを対象とした評価では，その限りでなく，分割する機能をなしにしても，ビルド時間が大幅に増加する事例を確認しており，本論文に示したビルド時間対策のみでは，最適解を求めるアルゴリズムの実用性を確保できなかった．

EEMBC を使った評価の範囲で，分割する機能をなしにすれば，ビルド時間への影響が大きくならなかった理由は，EEMBC が実行速度の評価に適した部分を抜粋したベンチマークであり，必ずしも多くの分岐を高密度に持たないためと考える．組込機器のメーカから収集した実アプリケーションに branch chaining を適用して得られるコードサイズの削減率は，近似解を求める第 1/第 2 のアルゴリズムを適用した場合で，それぞれ 0.92/0.85%と，EEMBC に適用した場合に比べどちらも 2.6 倍であり，このことから実アプリケーションにおける分岐の密度の高さを推察できる．なお，近似解を求めるアルゴリズムは，いずれも実アプリケーションのビルド時間に大きな影響を与えなかった．また，第 1/第 2 のアルゴリズムがもたらす削減率の差は，EEMBC を対象とした評価では $0.35 - 0.33 = 0.02$%にとどまったが，実アプリケーションを対象とした評価では $0.92 - 0.85 = 0.07$%であり，第 2 のアルゴリズムに対する第 1 のアルゴリズムの有用性は，実アプリケーションを対象とした評価でより顕著だった．

6. 結論

本論文では，Leverett らが提案した branch chaining のアルゴリズムを RX マイコン向けに拡張したものと，近似解を求めるものの実装について詳述し，それぞれの有用性を，コードサイズと実行命令数，ビルド時間に与える影響の評価により示した．評価の結果，組込機器向けベンチマーク EEMBC のコードサイズを，Leverett らのアルゴリズムであれば 0.40%，近似解を求めるアルゴリズムであれば 0.35%削減できることが分かった．このとき実行命令数の増加率は，ともに 0.14%になり，ビルド時間については，それぞれ最大で 314 倍，1.01 倍になることが分かった．

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] Leverett, B.W. and Szymanski, T.G.: Chaining Span-Dependent Jump Instructions, *ACM Trans. Program. Lang. Syst.*, Vol.2, No.3, pp.274–289 (online), DOI: 10.1145/357103.357105 (1980).

[13] Wulf, W.A., Johnsson, R.K., Weinstock, C.B., Hobbs, S.O. and Geschke, C.M.: *The Design of an Optimizing Compiler*, Elsevier Science Inc. (1975).

[14] Lattner, C.: The LLVM Compiler Infrastructure (2019), available from ⟨http://www.llvm.org⟩.

[15] Goyle, A., Mock, G. and Reynoso, A.: ARM Compiler Tips and Code Size Optimization Using DSP/BIOS Link (2005), available from ⟨http://www.ti.com/lit/an/spraac3/spraac3.pdf⟩.

[16] Chu, Y.-J. and Liu, T.-H.: On the Shortest Arborescence of a Directed Graph, *Science Sinica*, Vol.14, pp.1396–1400 (1965).

[17] Edmonds, J.: Optimum Branchings, *Journal of Re-*

© 2021 Information Processing Society of Japan

search of the National Bureau of Standards Section B, Vol.71B, No.4, pp.233–240 (1967).
[18] Tarjan, R.E.: Finding optimum branchings, *Networks*, Vol.7, No.1, pp.25–35 (1977).
[19] Camerini, P., Fratta, L. and Maffioli, F.: A note on finding optimum branchings, *Networks*, Vol.9, pp.309–312 (1979).
[20] Gabow, H.N., Galil, Z., Spencer, T. and Tarjan, R.E.: Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs, *Combinatorica*, Vol.6, No.2, pp.109–122 (online), DOI: 10.1007/BF02579168 (1986).
[21] 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).
[22] The Embedded Microprocessor Benchmark Consortium: Industry-Standard Benchmarks for Embedded Systems (1997), available from ⟨https://www.eembc.org⟩.
[23] The Embedded Microprocessor Benchmark Consortium: AutoBench 1.1, available from ⟨https://www.eembc.org/techlit/datasheets/autobench_db.pdf⟩.
[24] The Embedded Microprocessor Benchmark Consortium: ConsumerBench 1.1 Benchmark Software, available from ⟨https://www.eembc.org/benchmark/consumer_sl.php⟩.
[25] The Embedded Microprocessor Benchmark Consortium: TeleBench 1.1, available from ⟨https://www.eembc.org/techlit/datasheets/telecom_db.pdf⟩.
[26] The Embedded Microprocessor Benchmark Consortium: Embedded Microprocessor Benchmark Consortium (2020), available from ⟨https://github.com/eembc⟩.

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．

永井 佑樹

1991 年生．2015 年東京都市大学大学院工学研究科システム情報工学専攻修士課程修了．株式会社日立ソリューションズにおいてコンパイラの開発に従事．

中川 満

1980 年生．2005 年大分大学工学部大学院工学研究科知能情報システム工学専攻博士前期課程修了．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．

© 2021 Information Processing Society of Japan