情報処理学会論文誌　プログラミング　Vol.11　No.1　1–9 (Feb. 2018)

RL78マイコン向けCコンパイラによる多倍長整数演算の最適化

千葉 雄司[1,a)]　永井 佑樹[2]　中川 満[3]

受付日 2017年7月23日, 採録日 2017年9月28日

概要：特定のプロセッサ向けに開発したソースコードを，よりビット幅の狭い演算器を持つプロセッサ向けに再利用すると，ソースコード内の演算が多倍長になり，コードサイズの増加や，実行速度の劣化といった問題をおこすことがある．問題への対策として，本論文では 2 つの最適化コンパイル技術を提案する．提案する技術の 1 つは配列の長さの上限を手がかりに，演算を多倍長でなくし，もう 1 つは多倍長の演算の中でも頻出する定数の加算を，ルネサスエレクトロニクスの RL78 マイコンで効率的に実現する．提案した技術の効果を，組込プロセッサ向けベンチマーク CoreMark によって評価したところ，実行を 12.8%高速化し，コードサイズを 9.0%小さくすることが分かった．

キーワード：コンパイラ，最適化，多倍長

Optimizing Multiple Precision Arithmetics by a C Compiler for the RL78 Microcontroller

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

Received: July 23, 2017, Accepted: September 28, 2017

Abstract: Source code developed for a processor is sometimes re-used for a narrower bit-width processor, but the source code often contains arithmetics whose precision are multiple for the narrower bit-width processor, and the multiple precision arithmetics decrease performance and increase code size. To cope with the problem, we propose two optimization techniques: One truncates the arithmetics bit-width using the upper limit of array length, and the other improve performance of addition of constant by the Renesas electronics RL78 microcontroller, which is one of the most popular multiple precision arithmetics. We evaluated the effect of these techniques using CoreMark, a bencmark suite for embedded processors, and found they improve performance by 12.8% and reduce code size by 9.0%.

Keywords: compiler, optimization, multiple precision

1. はじめに

近年，ローエンドのプロセッサの性能向上にともない，従来，より高機能なプロセッサで実現していた機能を，ローエンドのプロセッサで実現する機会が生じている．そのようなケースでは，より高機能なプロセッサ向けに開発したソフトウェアを，ローエンドのプロセッサで再利用することがある．再利用に際し，より高機能なプロセッサとローエンドのプロセッサの命令セットが互換でない場合，少なくともソフトウェアをコンパイルしなおす必要が生じるが，コンパイルしなおすだけでは，実行はできても実行効率が上がらなかったり，コードサイズが大きくなったりすることがある．

たとえば**図 1** のソースコードについて考える．図 1 のソースコードは C 言語で記述したもので，1 から 2 行目に現れる型 `uint32_t` と `uint8_t` はそれぞれ 32 bit と 8 bit の

[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

© 2018 Information Processing Society of Japan

```
1:  void func(uint32_t N, uint8_t* A){
2:    uint32_t i, j;
3:    for(i = 0; i < N; ++i){
4:      for(j = 0; j < N; ++j){
5:        A[i*N+j] = 0;
6:      }
7:    }
8:  }
```

図 1　32 bit プロセッサ向けに書いたソースコード
Fig. 1　Source code written for 32 bit processors.

```
1:  addw ax, 1
2:  xch a, c
3:  addc a, 0
4:  xch a, c
5:  xch a, b
6:  addc a, 0
7:  xch a, b
```

図 2　RL78 マイコンによる多倍長加算の命令列
Fig. 2　RL78 instruction sequence for multiple precision addition.

符号なし整数型とする．

図 1 のソースコードは，より高い演算能力を持つ 32 bit のプロセッサ向けにも，ローエンドのたとえば 16 bit のプロセッサ向けにもコンパイルでき，どちらでも正しく実行できるものの，16 bit のプロセッサでは効率的に実行できなかったり，コードサイズが大きくなったりといった問題をおこしうる．問題をおこす原因は変数 N，i，j のビット幅が 32 bit で，それらに対する演算を 1 命令で実現できないことにある．たとえば 3 行目にある演算 ++i をルネサスエレクトロニクスの 16 bit のローエンドマイコン RL78 [1] で実現しようとすると，**図 2** に示すように，いくつもの命令が必要になる．

図 2 の命令列は変数 i の下位 16 bit を RL78 マイコンの汎用レジスタ ax に，上位 16 bit を汎用レジスタ bc に収めてある場合向けのものである．RL78 マイコンは 16 bit の汎用レジスタを 4 本持つが，4 本の汎用レジスタ ax，bc，de，hl はそれぞれ 2 つに分割でき，8 bit の汎用レジスタ a，x，b，c，d，e，h，l として利用することもできる．図 2 の命令列では下位 16 bit は 16 bit の加算命令 addw で加算しているものの，上位 16 bit については 8 bit のキャリ付き加算命令 addc で 8 bit ずつ加算している．ここで上位 16 bit の加算に 16 bit のキャリ付き加算命令を使わない理由は，RL78 マイコンが 16 bit のキャリ付き加算命令を提供しないからである．また，図 2 の命令列には，加算命令のほかに，レジスタの内容を交換する命令 xch が入っているが，命令 xch が必要になる理由は，RL78 マイコンの命令セットでは原則として命令のオペランドの 1 つをアキュムレータと呼ぶ汎用レジスタ a もしくは ax にしかできないためである．たとえば命令 addc でレジスタに定数を加算する場合，命令 addc のオペランドになれるレジスタがアキュムレータ a だけであるため，被加数をアキュムレータ a に移動しなくてはならない．

図 2 の命令列は，実行効率もコードサイズも優れているとは言い難い．実行効率やコードサイズを改善する手段の 1 つは図 1 のソースコードを手動で修正して変数 N，i，j の型を，たとえばポインタと同じサイズの整数型にすることである．しかしながら，手動で修正する場合，ソフトウェアを見直してどこに問題があるのか探し出さなければならず，見直しに相応のコストがかかってしまう．修正の目的が実行効率の改善である場合には，かかるコストを軽減するために，実行頻度の高い場所に限って修正することもできるが，修正の目的がコードサイズの削減である場合には，そういった方法も適用しにくい．

そこで本論文では，より高機能なプロセッサ向けに開発したソースコード中の多倍長整数演算がもたらす非効率を，コンパイラの最適化によって軽減する方法を提案する．多倍長整数演算向けの最適化は古くからあり，既存のものは大きく次の 2 つに分類できるが，本論文ではこの前者と後者に分類できる技術を 1 つずつ提案する．

- 演算のビット幅を削減して多倍長でなくする技法
- 演算のビット幅はそのままで効率を改善する技法

本論文で提案する技術のうち，前者に分類できるものは，ビット幅を削減できる機会を増やすためのものであり，後者に分類できるものは，RL78 マイコンにおける 32 bit の定数の加算を高速化するものである．

本論文の構成を次に示す．まず，2 章でビット幅の削減に関連する技術を提案し，次に 3 章で 32 bit の定数の加算を高速化する技術を提案する．続く 4 章では提案した技術の評価結果を示し，5 章で関連研究について述べる．6 章は結論である．

2.　ビット幅の削減

ビット幅の削減は古典的な最適化技法の 1 つであり，たとえば図 1 の 5 行目にある，32 bit の整数の積と和を求める式 i*N+j について，そのビット幅を削減し，下位 16 bit の積と和に最適化する．ここで削減を可能にする根拠は，式 i*N+j の計算結果の用途がもっぱらアドレス参照の添字であるので，ポインタのビット幅が 16 bit ならば上位 16 bit の計算を行っても意味がないことに求められる．

この例が示すように，ビット幅の削減では，まず，ビット幅を削減可能にする根拠を求め，次に，求めた根拠に基づいて削減を行う．したがって，どれだけ削減できるかは，根拠を求める機能がどれだけ充実しているかに依存する．

根拠を求めることができなければ最適化できないということは必ずしもなく，プログラムの一部を複製して [2], [3], [4]，ビット幅を削減できる場合のみ複製した部分に分岐するよ

© 2018 Information Processing Society of Japan

```
 1:  void func(uint32_t N, uint8_t* A){
 2:    if (N < 65536){
 3:      // N を 16bit で表現できる場合向けのループ
 4:      uint16_t N16 = (uint16_t)N;
 5:      uint16_t i16, j16;
 6:      for(i16 = 0; i16 < N16; ++i16){
 7:        for(j16 = 0; j16 < N16; ++j16){
 8:          A[i16*N16+j16] = 0;
 9:        }
10:      }
11:    }
12:    else{
13:      // N を 16bit で表現できない場合向けのループ
14:      uint32_t i, j;
15:      for(i = 0; i < N; ++i){
16:        for(j = 0; j < N; ++j){
17:          A[i*N+j] = 0;
18:        }
19:      }
20:    }
21:  }
```

図 3　ループの versioning
Fig. 3　Loop versioning.

うにすれば，複製した部分にビット幅を削減する最適化を適用可能になる．たとえば図 1 のプログラムについて，実行プロファイルを取得した結果，3 から 7 行目のループの実行頻度が高く，なおかつ変数 N の値が 65536 より小さいことばかりと分かったら，ループを複製し，その一方に変数 N の値が 65536 未満の場合のみ到達するよう分岐を設ける．そして，設けた分岐を根拠に変数 N と，変数 N より小さな値しか持ちえない変数 i，j のビット幅を 16 bit に削減する．ループの複製とビット幅の削減を行った結果を**図 3** に示す．図 3 の 4，5 行目に現れる型 uint16_t は 16 bit の符号なし整数型とする．図 3 の 6 から 10 行目にあるループでは 16 bit の変数 N16，i16，j16 を利用して実行効率を改善している．

プログラムの一部を複製して最適化する方法は適用範囲が広く有用だが，コードサイズを増やすため，コードサイズの削減が重要な課題であるローエンドマイコン向けコンパイラでは利用しにくい．

そこで本論文ではビット幅を削減可能にする根拠を求める機能の強化を目指す．根拠を求めてビット幅を削減する方法なら，実行速度の改善と，コードサイズの削減を両立できる．本論文で提案するのは，この根拠を求める技法の 1 つである．本論文で提案する技法を使えば，たとえば図 1 のソースコードを**図 4** のソースコードに最適化できる．

本論文で提案する技法では，次の要件を満たすループについて，ループ帰納変数の値域を定め，そのビット幅を削減可能にする．

```
1:  void func(uint32_t N, uint8_t* A){
2:    uint16_t N16 = (uint16_t)N;
3:    uint16_t i16, j16;
4:    for(i16 = 0; i16 < N16; ++i16){
5:      for(j16 = 0; j16 < N16; ++j16){
6:        A[i16*N16+j16] = 0;
7:      }
8:    }
9:  }
```

図 4　16 bit プロセッサ向けに最適化したソースコード
Fig. 4　Source code optimized for 16 bit processors.

(1) ループ帰納変数の初期値が 0 である．
(2) ループ帰納変数をループを周回するたびに 1 ずつカウントアップし，なおかつ，ループ帰納変数を変更する要因をほかに持たない．
(3) ループの出口条件はループ帰納変数がループ不変な値 N に達することのみである．
(4) ループの周回時に必ず通過する箇所に配列参照がある．
(5) (4) で参照する配列のアドレスがループ不変である．
(6) (4) の配列参照の添字がループ帰納変数，もしくはループ帰納変数とループ不変な値の和である．
(7) 無限に周回しうる内側ループを持たない．
(8) ループ内に，volatile なメモリ参照など，ループに入ってからループを出るまでの計算の途中経過を非同期に観測可能にする要因がない．

これらの条件を満たすループでは，いったんループに入ったら，ループから抜けるまでの間に配列の連続する N 個の要素を参照することが保証される．ここで配列の要素数 M は，ポインタのビット幅を n，配列要素の大きさと整列境界をそれぞれ s，a とおくとき，次の式で表現できる．

$$M = (2^n - a)/s \tag{1}$$

式 (1) の中にある記号 / は整数の除算を意味し，商の少数点以下を切り捨てるものとする．式 (1) でアドレス空間の広さ 2^n から整列境界値 a を減じるのは，null ポインタの実現に 0 を使うことを想定した結果である．C 言語の仕様では null ポインタを，いかなるデータもしくは関数のアドレスとも一致しないものと定めているので，null ポインタの実現に 0 を使うなら，0 を先頭アドレスとする配列は作成できず，したがって配列の先頭アドレスの最小値は a となり，配列を配置できるアドレス空間の広さは最大でも $2^n - a$ になる．このアドレス空間の広さを配列要素の大きさ s で除したのが式 (1) である．

プログラマが式 (1) の値以上に連続する要素を参照するプログラムを書いた場合，書いたプログラムの動作は C 言語の規定により未定義になる．ここでプログラマが動作の未定義なプログラムを書かないと想定してよいなら，式 (1) の値がすなわちループ帰納変数の上限値 N のとりうる最

© 2018 Information Processing Society of Japan

大値となる．このことを使ってループ帰納変数やその上限値のビット幅を切り詰めるのが本論文の提案技法である．

提案技法による最適化の例として，図 1 のソースコードを図 4 のソースコードに最適化する過程を示す．図 1 のソースコードでは，4 から 6 行目のループが要件 (1) から (8) のすべてを満たしているため，本論文で提案した技法により，ループ帰納変数 j およびその上限値 N のとりうる最大値を式 (1) によって定められる．ここでポインタのビット幅を 16 bit，1 バイトのデータの整列境界を 1 バイトとするなら式 (1) の値は $(2^{16}-1)/1 = 65535$ になる．ループ帰納変数 j の下限値は 0 であり，また上限値 N も型が符号なしなので，その下限値は 0 であるといえ，したがって変数 j，N の値域は 0 から 65535，すなわち 16 bit の符号なし整数型で表現できる範囲に収まるといえる．この制約はプログラムの制御が 4 から 6 行目のループに到達した場合のみ適用可能になるもので，3 行目でループに一度も入らない場合には成立するとは限らないが，3 行目でループに一度も入らない場合に変数 N がとりうる値は 0 だけで，したがって 3 行目でループに入っても入らなくても，プログラムの制御が 3 行目に到達した時点で変数 N のとりうる最大値は 65535 になっているといえる．したがって変数 N を上限とする外側ループのループ帰納変数 i のとりうる最大値も 65535 といえ，また変数 i の最小値は初期値 0 なのでループ帰納変数 i も 16 bit の符号なし整数型で表現できる．ここまでに得た知見によってループ帰納変数 i，j と，ループ内で参照する変数 N のビット幅を削減すると図 4 のソースコードが得られる．

なお，提案技法がループ帰納変数の上限値 N について定められるのは，その最大値だけで，最小値は定められないことに注意する必要がある．図 1 から図 4 への最適化の例では上限値 N の型が符号なしであることを利用して最小値を 0 と定めたが，型が符号ありだと，型から最小値を 0 と定めることはできなくなる．しかしながら最小値を 0 と定められなくても，提案技法によって最大値を定めれば，一定の最適化を適用できる．

たとえば**図 5** (a) のソースコードに提案技法を適用すると，ループに到達した時点で変数 k，M のとりうる最大値が 65535 になると分かり，また変数 k については初期値が 0 であることから最小値も 0 と分かる．ここまでに分かったことから変数 k については 16 bit の符号なし整数で表現できると分かるのでビット幅を削減できる．一方，変数 M のビット幅は削減できない．なぜなら変数 M の型 int32_t が 32 bit の符号あり整数型であるとすると，型によって定まる最小値が 0 より小さくなり，また型のほかに最小値を定める要因がないからである．ただ，変数 M を参照するのは図 5 (a) の 3 行目にある 32 bit の比較 k < M のみで，この比較は 16 bit のマイコンでは図 5 (b) の 10 から 12 行目に示すように，上位 16 bit と下位 16 bit の比較に分けて実施することになるが，上位 16 bit の比較に最適化の余地がある．上位 16 bit の比較は図 5 (b) の 10，11 行目にある．10，11 行目の比較では，変数 k の上位 16 bit を 0 としているが，その理由は，値域の解析の結果から，変数 k の上位 16 bit はつねに 0 と分かっているからである．ここで 10 行目の比較について考えると，変数 M についてもその最大値が 65535 と分かっているので，変数 M の上位 16 bit は 0 以下といえ，したがって 10 行目の比較の結果は必ず偽といえる．11 行目の比較の結果は真になるか偽になるか不明だが，ループ不変なのでループ外に移動できる．これらの最適化を図 5 (b) に適用すると図 5 (c) のプログラムを得られる．

```
1:  void func(int32_t M, uint8_t* A){
2:    int32_t k;
3:    for(k = 0; k < M; ++k){
4:      A[k] = 0;
5:    }
6:  }
```

(a) 最適化前

```
7:  void func(int32_t M, uint8_t* A){
8:    uint16_t k16;
9:    for(k16 = 0;
10:         (0 < M の上位 16bit) ||
11:         ((0 == M の上位 16bit) &&
12:          (k16 < M の下位 16bit)); ++k16){
13:     A[k16] = 0;
14:   }
15: }
```

(b) 最適化中

```
16: void func(int32_t M, uint8_t* A){
17:   if (0 == M の上位 16bit){
18:     uint16_t M16 = (uint16_t)M;
19:     uint16_t k16;
20:     for(k16 = 0; k16 < M16; ++k16){
21:       A[k16] = 0;
22:     }
23:   }
24: }
```

(c) 最適化後

図 5　上限値の型が符号ありの場合の最適化
Fig. 5　Optimization for signed upper limit.

図 5 (c) に示した符号ありの場合の最適化の結果と，図 4 に示した符号なしの場合の最適化の結果を比較すると，符号ありの場合には，ループ外に分岐が 1 つ追加で必要になることが分かる．

3.　32 bit の定数の加算の高速化

ビット幅の削減は有用な最適化技術だが，変数の値域など，最適化に必要な根拠を求められるか，コードの複製に

© 2018 Information Processing Society of Japan

```
1:   addw ax, 加数の下位 16bit
2:   sknc
3:   incw bc
4:   xchw ax, bc
5:   addw ax, 加数の上位 16bit
6:   xchw ax, bc
```

(a) 加数の上位 16bit の値を問わない命令列

```
7:   addw ax, 加数の下位 16bit
8:   sknc
9:   incw bc
```

(b) 加数の上位 16bit が 0 の場合向けの命令列

```
10:  addw ax, 加数の下位 16bit
11:  skc
12:  decw bc
```

(c) 加数の上位 16bit が 0xffff の場合向けの命令列

図 6　32 bit の定数を加算する命令列
Fig. 6　Instructions to add 32 bit constant.

ともなうコードサイズの増加を許容できなければ適用できず，適用できない場合には，多倍長整数演算を利用せざるをえない．多倍長整数演算の命令列を RL78 マイコン向けコンパイラで生成する際には，RL78 マイコンの命令セットがサポートする 16 bit 演算命令が少ないことから，固有の工夫が必要になるが，本章では多倍長整数演算の中で特に頻出し，なおかつ特徴的なコードの生成技術を必要とする 32 bit の定数の加算について述べる．

RL78 マイコンで多倍長の定数の加算に使う汎用的な命令列は図 2 に示したものである．図 2 の命令列の特徴は，加算の結果，オーバフローが発生した場合，そのことをキャリフラグに反映することにあり，したがって図 2 の命令列はたとえば 64 bit 整数の下位 32 bit の加算に使うこともできる．しかしながら 32 bit 整数の加算を行うだけならばオーバフローの発生をキャリフラグに反映する必要はないので，図 2 の命令列を使う必要はなく，32 bit の加算に特化した**図 6** の命令列を使うこともできる．

図 6 には 3 種類の加算の命令列 (a)，(b)，(c) を示した．命令列 (a)，(b)，(c) はいずれも図 2 と同様に，被加数の上位 16 bit がレジスタ bc に，下位 16 bit がレジスタ ax に収まっている場合向けのものである．命令列 (a) は上位 16 bit の値の値を問わないもので，まず 1 行目で下位 16 bit の加算を行い，続く 2 から 3 行目で下位 16 bit の加算でオーバフローが発生したか判定し，発生したなら上位 16 bit に 1 を加算する処理を行う．2 行目の命令 sknc はキャリフラグが立っていなかった場合に直後の命令の実行をスキップするためのものであり，直後の 3 行目にある命令 incw は 16 bit の 1 を加算する命令である．命令 incw は，加算命令 addw と異なり，被加数のオペランドとして ax 以外のレジスタも受理できる．続く 4 行目では命令 xchw でレジスタ ax と bc の内容を入れ替えることで，5 行目にある上位 16 bit の加算命令が受理できるレジスタ ax に被加数の上位 16 bit を移動する．5 行目までで加算は終了しているが，5 行目までだと被仮数の上位 16 bit と下位 16 bit を収めるレジスタが 4 行目の命令で入れ替わったままであり，それでは問題がある場合には，たとえば 6 行目の命令を使って再度入替を行う必要がある．

表 1　32 bit の定数を加算する命令列の比較
Table 1　Comparison of 32 bit constant additions.

命令列	サイズ (byte)	実行サイクル数	
		コア	
		S1	S2/S3
図 2	11	8	7
図 6 (a)	11	11	6
図 6 (b) (c)	6	4*	3

* 上 16 bit への加算を行う場合は 5

命令列 (a) は 6 命令からなり，図 2 に示した汎用的な加算の命令列とサイズは同じで，実行サイクル数は**表 1** に示すとおりコアによりけりである（RL78 マイコンのコアには S1/S2/S3 の 3 種類があり，各コアは基本的に 1 命令を 1 サイクルで実行するが，S1 コアに限っては 16 bit の演算命令の実行に 2 サイクルを必要とする）．したがって図 2 の命令列と図 6 の命令列 (a) に関してはコアに応じて適切な方を使うべきといえる．

命令列 (b)，(c) は加数の上位 16 bit が 0 もしくは 0xffff の場合に利用できる命令列である．命令列 (b)，(c) は，加数の上位 16 bit が 0 もしくは 0xffff であることを利用して，命令列 (a) から加数の上位 16 bit の加算命令と，その前後の命令 xchw を省略したもので，省略により，サイズも実行サイクル数もコアを問わず最小になっている．

なお，図 2 や図 6 は被加数がレジスタ上にある場合向けの命令列だが，被加数はメモリ上にある場合もある．被加数がメモリ上にある場合，基本的には図 2 や図 6 の命令列の前後に，被加数をメモリからレジスタにロードする命令列と，和をレジスタからメモリにストアをする命令列を追加すれば加算を実施できるが，ロードやストアと加算の実施順序に配慮するとコードサイズや実行サイクルを改善できる．たとえば図 6 (a) の命令列の前後にロードとストアの命令列を挿入すると**図 7** (a) の命令列になるが，被加数が volatile でないなら，下位 16 bit の加算，ストア後に上位 16 bit をロードする図 7 (b) の命令列を利用できる．

図 7 (b) の命令列から，その 17 行目にある加数の上位 16 bit の加算命令を除去すると，被加数が volatile でなく，加数の上位 16 bit が 0 の場合の命令列**図 8** (a) を作ることができるが，図 8 (a) の命令列には最適化の余地がある．すなわち，図 8 (a) の 7 行目にある和の上位 16 bit のストアはキャリフラグが立っていない場合には実行する必要がなく，また 4 行目にある被加数の上位 16 bit のロード

© 2018 Information Processing Society of Japan

```
 1:   movw ax, [sp+6]
 2:   movw bc, ax
 3:   movw ax, [sp+4]
 4:   addw ax, 加数の下位 16bit
 5:   movw [sp+4], ax
 6:   sknc
 7:   incw bc
 8:   movw ax, bc
 9:   addw ax, 加数の上位 16bit
10:   movw [sp+6], ax
```

(a) 被加数が volatile である場合

```
11:   movw ax, [sp+4]
12:   addw ax, 加数の下位 16bit
13:   movw [sp+4], ax
14:   movw ax, [sp+6]
15:   sknc
16:   incw ax
17:   addw ax, 加数の上位 16bit
18:   movw [sp+6], ax
```

(b) 被加数が volatile でない場合

図 7　メモリ上の被加数への定数の加算

Fig. 7　Instructions to add constant value to a value on memory.

についても，キャリフラグが立っておらず，なおかつ，25行目のストアより先にレジスタ ax の内容を参照する命令がなければ実行する必要がないことから，たとえば図 8 (a) の命令列を図 8 (b) の命令列に書き換えられることがある．ただし，この書き換えだけでは有用なコードは得られない．具体的には，図 8 (a) から図 8 (b) への書き換えでコードサイズは変化せず，実行サイクル数は，**表 2** に示すとおり，S1 コアでキャリフラグが立たなかった場合に小さくなるものの，ほかは同じか，むしろ大きくなる．大きくなる原因は，キャリフラグが立たなかった場合に実行しない命令の数が増えると，スキップ命令 sknc を利用できなくなり，代わりに条件分岐命令 bc を使うことになるが，RL78 マイコンの条件分岐命令は実行に多くのサイクル数を必要とするからである．RL78 マイコンの条件分岐の実行に必要なサイクル数は，コアを問わず一定で，分岐条件の不成立時が 2，成立時が 4 である．ここで条件不成立時の実行サイクルが 2 と比較的小さいことに着目すると，条件不成立時に特化した最適化を適用することが可能になる．すなわち，加数が十分に小さな値である場合には，下 16 bit の加算の結果としてキャリフラグが立つ確率は十分に低いと想定でき，その場合には分岐確率を考慮し，図 8 (b) の命令列を図 8 (c) の命令列に書き換えると，26 行目に無条件分岐命令が増える分，コードサイズは増えるとしても，表 2 に示すように，キャリフラグが立たなかった場合の実行サイクル数を改善できる．

また，図 8 (b) の命令列を最適化する手段としては，

```
 1:     movw ax, [sp+4]
 2:     addw ax, 加数の下位 16bit
 3:     movw [sp+4], ax
 4:     movw ax, [sp+6]
 5:     sknc
 6:     incw ax
 7:     movw [sp+6], ax
```

(a) 定数の上位 16bit の加算の削除

```
 8:     movw ax, [sp+4]
 9:     addw ax, 加数の下位 16bit
10:     movw [sp+4], ax
11:     bnc Lnocarry
12:     movw ax, [sp+6]
13:     incw ax
14:     movw [sp+6], ax
15:   Lnocarry:
```

(b) キャリフラグが立たなかった際のロードストアの省略

```
16:     movw ax, [sp+4]
17:     addw ax, 加数の下位 16bit
18:     movw [sp+4], ax
19:     bc Lcarry
20:   Lnocarry:
21:     ...
22:   Lcarry:
23:     movw ax, [sp+6]
24:     incw ax
25:     movw [sp+6], ax
26:     br Lnocarry
```

(c) 分岐の最適化

```
27:     movw ax, [sp+4]
28:     addw ax, 加数の下位 16bit
29:     movw [sp+4], ax
30:     movw hl, sp
31:     sknc
32:     incw [hl+6]
```

(d) メモリオペランドの利用

図 8　メモリ上の volatile でない値と上位 16 bit が 0 の定数の加算

Fig. 8　Instructions to add a non volatile value on memory and a 32 bit constant whose upper 16 bits are 0.

表 2　図 8 の命令列の実行サイクル数

Table 2　Execution cycles of the instruction sequences in Fig. 8.

コア	S1		S2/S3	
キャリ	あり	なし	あり	なし
命令列 (a)	13	12	7	7
命令列 (b)	14	10	8	7
命令列 (c)	20	8	14	5
命令列 (d)	11	10	6	6

図 8 (d) に示すように，キャリを加算する命令 incw のオペランドにメモリオペランドを利用する方法もある．図 8 (d) で命令 incw がメモリオペランドのベースとして利用して

© 2018 Information Processing Society of Japan

いるレジスタがスタックポインタ sp でなく hl である理由は，RL78 マイコンの命令セットで命令 incw のベースとして利用できるレジスタが hl だけだからである．ベースレジスタが hl であることから，図 8 (d) の命令列では 30 行目でレジスタ sp から hl へのコピーを行っているが，このコピーは必ずしも 32 行目の命令 incw のためだけに必要になるわけでなく，また 30 行目に必要にならないこともある．その理由は参考文献 [5] にあるようにレジスタ hl のみをベースとして利用できる命令がほかにもあり，それらがレジスタ hl にレジスタ sp と同じ値を保持させるよう要求することがあるからである．図 8 (d) の命令列はレジスタ hl をほかの用途に使っている場合は利用できないが，実行サイクル数は 30 行目の複写命令を含めても表 2 に示すとおり最も少ないので，実行速度を優先する最適化を行う際には積極的に使用すべきといえる．ただし，コードサイズに関しては，30 行目の複写命令を含めれば 15 byte，含めなければ 12 byte であり，図 8 (a) の命令列の 14 byte に比べ必ずしも小さくならない．なお，図 8 (d) の 30 行目の複写命令は，複写の結果の利用者が 32 行目の命令 incw のみであるならば，キャリフラグが立っている場合に実行する必要がなくなるので，図 8 (c) の命令列と同様にキャリフラグが立っている場合のみ実行するようにすることもできる．ただし，我々が評価した範囲では，そのような最適化が有意な結果をもたらすことはなかった．

4. 評価

本章では，提案した最適化が効果を評価した結果を示す．評価の対象は実行速度およびコードサイズとし，測定対象のプログラムは組込マイコンの性能測定を目的としたベンチマークである CoreMark [6] を用いた．CoreMark は，現実的な組込機器向けアプリケーションが含みうる演算から，組込マイコンやコンパイラの生成したコードの性能から大きな影響を受けるものを抜粋したベンチマークで，具体的には次の演算を構成要素とする．

- 行列演算
- リストの操作
- ステートマシンの操作
- 巡回冗長検査

評価にはルネサスエレクトロニクスの最適化 C コンパイラ CC-RL [5], [7] を用い，当該コンパイラで提案した最適化を適用する場合の効果を測定した．CoreMark のコンパイルに際しては，実行速度への影響を測定する際には，オプション -Ospeed を指定して，実行の高速化を目的とした最適化を，コードサイズを測定する際にはオプション -Osize を指定して，コードサイズの削減を目的とする最適化を，それぞれ適用した．いずれの測定においても RL78 マイコンのコアは S3 とした．プログラムの実行には RL78 マイコンそのものではなく，RL78 マイコンの CPU シミュレータを用いた．当該 CPU シミュレータを用いてプログラムを実行する場合，実行サイクル数の測定結果は，プログラムのコンパイル結果に対して唯一に定まり，実行ごとに変化することはない．

表 3　評価結果
Table 3　Evaluation result.

ビット幅の削減	32 bit 定数加算の最適化	コードサイズ削減率 (%)	実行高速化率 (%)
なし	なし	0.0	0.0
なし	あり	2.1	4.3
あり	なし	6.5	9.4
あり	あり	9.0	12.8

評価結果を**表 3** に示す．表 3 のコードサイズ削減率の欄から定数加算の最適化によってコードサイズが減っていることが分かる．定数加算の最適化が必ずしもコードサイズを削減しないにもかかわらず減った理由は次の 2 つである．

(1) コードサイズの小さな命令列，すなわち図 6 の命令列 (b)，(c) を利用できるケースがあった．
(2) オプション -Osize の指定時はコードサイズを増やす最適化，すなわち分岐の最適化を実施しない．

また表 3 の実行高速化率の欄をみると，個々の最適化のみを適用した場合の効果の和が $4.3 + 9.4 = 13.7\%$ であるのに対し，双方の最適化を適用した場合の効果が 12.8%にとどまっていることが分かる．その原因は，双方の最適化対象が重複していて，ビット幅を削減すると多倍長演算の最適化が効かなくなることにある．なお，2 章で提案したビット幅の削減は CoreMark 中のループの 33%に適用できた．

5. 関連研究

ビット幅の削減やその実現に必要になる値域の解析といった技術は汎用プロセッサ向けコンパイラの最適化を構成する技術として発展してきた [8], [9], [10], [11], [12], [13], [14] ほか，C 言語で記述したソフトウェアを FPGA などのハードウェアで実現する際に演算器のビット幅を必要最小限にするための技術としても発展してきた [15], [16], [17], [18], [19], [20], [21], [22], [23]．ビット幅の削減に必要な値域の情報を配列の添字として使われる値から求める技法については Stephenson ら [16] が，配列のサイズの上限をアドレス空間の広さから求める技法については Nayak ら [19] がすでに言及しているが，本論文はアドレス空間の広さとループの特徴を併用してビット幅を削減する技法を示している点で過去の研究と異なる．ループの特徴を利用する最適化技法 [8] には，適用範囲が狭いという問題があり，値域を求めることを目的とした技法の多くは個々の演算，たとえば配列参照の添字として値を使用することがもたらす制約を，データフローに

© 2018 Information Processing Society of Japan

沿って前方あるいは後方に伝播することで値域を求めてきた [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]．こういった技法がもたらす値域を改善するために，本論文で示した知見，すなわち，null を 0 で表現する C 言語の処理系では，配列の先頭アドレスが 0 になりえない分だけ配列要素の最大数が少なくなることを活用できる．

また本論文では，RL78 マイコンによって効率的に多倍長（32 bit）の定数を加算する命令列を示し，定数が十分小さい場合にはキャリフラグが立ちにくいと予測して基本ブロックを再配置することで性能を改善できることを示した．フラグが立つか否かを静的に予測する技術 [24], [25], [26], [27], [28], [29] や，基本ブロックを再配置する技術 [30], [31] は既知のものだが，本論文ではそれらが RL78 マイコンにおける多倍長の加算の実現に有用であることを指摘し，その効果を定量的に示した．

なお，本論文で示した多倍長加算の命令列は RL78 マイコン向けのものだが，同様の命令列は AVR [32] や PICmicro [33]，F^2MC-16LX [34] でも利用できる．なぜなら，これらのマイコンでは，RL78 マイコンと同様に，最大のビット幅の演算にキャリ付き加算命令がないことから，キャリ付き加算の実現に条件分岐が必要になり，なおかつ，分岐条件の成立時と不成立時で条件分岐の実行サイクルが異なることから，基本ブロックの再配置によって性能を改善できるからである．

6. 結論

本論文では，より高機能なプロセッサ向けに開発したプログラム中に存在する多倍長整数演算向けの最適化技法として，配列の大きさの上限を利用して多倍長整数演算のビット幅を削減する技法と，32 bit の定数の加算を高速化する RL78 マイコン向けの技法を提案した．提案した技法の効果を，組込プロセッサ向けベンチマーク CoreMark によって評価したところ，実行を 12.8%高速化し，コードサイズを 9.0%小さくすることが分かった．CoreMark 以外のソフトウェアに対する提案技法の有効性の検証ついては今後の課題としたい．

参考文献

[1] Renesas Electronics Corporation: RL78 Family (2015). available from ⟨http://www.renesas.com/products/mpumcu/rl78/⟩

[2] Byler, M., Wolfe, M., Davies, J.R.B., Huson, C. and Leasure, B.: Multiple Version Loops, *International Conference on Parallel Processing*, *ICPP'87*, University Park, PA, USA, August 1987, pp.312–318 (1987).

[3] Mikheev, V.V., Fedoseev, S.A., Sukharev, V.V. and Lipsky, N.V.: Effective Enhancement of Loop Versioning in Java, *Proc. 11th International Conference on Compiler Construction, CC '02*, London, UK, UK, pp.293–306, Springer-Verlag (online), available from ⟨http://dl.acm.org/citation.cfm?id=647478.727926⟩ (2002).

[4] Doerfert, J., Grosser, T. and Hack, S.: Optimistic Loop Optimization, *Proc. 2017 International Symposium on Code Generation and Optimization*, *CGO '17*, Piscataway, NJ, USA, pp.292–304, IEEE Press (online), available from ⟨http://dl.acm.org/citation.cfm?id=3049832.3049864⟩ (2017).

[5] 千葉雄司，西村啓成，中川　満：RL78 マイコン向け C コンパイラ CC-RL における機種依存最適化の設計，情報処理学会論文誌プログラミング（PRO），Vol.9, No.3, pp.1–20 (2016).

[6] The Embedded Microprocessor Benchmark Consortium: CoreMark An EEMBC Benchmark (2009). available from ⟨http://www.eembc.org/coremark/⟩

[7] Renesas Electronics Corporation: C Compiler Package for RL78 Family (2015). available from ⟨http://am.renesas.com/products/tools/coding_tools/c_compilers_assemblers/rl78_compiler/index.jsp.⟩

[8] Harrison, W.H.: Compiler Analysis of the Value Ranges for Variables, *IEEE Trans. Softw. Eng.*, Vol.3, No.3, pp.243–250 (online), DOI: 10.1109/TSE.1977.231133 (1977).

[9] Blume, W. and Eigenmann, R.: Symbolic Range Propagation, *Proc. 9th International Symposium on Parallel Processing*, *IPPS '95*, Washington, DC, USA, IEEE Computer Society, pp.357–363 (online), available from ⟨http://dl.acm.org/citation.cfm?id=645605.663081⟩ (1995).

[10] Kawahito, M., Komatsu, H. and Nakatani, T.: Effective Sign Extension Elimination, *SIGPLAN Not.*, Vol.37, No.5, pp.187–198 (online), DOI: 10.1145/543552.512552 (2002).

[11] Ananian, C.S. and Rinard, M.: Data Size Optimizations for Java Programs, *SIGPLAN Not.*, Vol.38, No.7, pp.59–68 (online), DOI: 10.1145/780731.780741 (2003).

[12] Birch, J., van Engelen, R. and Gallivan, K.: Value Range Analysis of Conditionally Updated Variables and Pointers, *Proc. Compilers for Parallel Computing (CPC) 2004*, pp.265–276 (2004).

[13] Rugina, R. and Rinard, M.C.: Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions, *ACM Trans. Program. Lang. Syst.*, Vol.27, No.2, pp.185–235 (online), DOI: 10.1145/1057387.1057388 (2005).

[14] Campos, V.H.S., Rodrigues, R.E., de Assis Costa, I.R. and Pereira, F.M.Q. a.: Speed and Precision in Range Analysis, *Proc. 16th Brazilian Conference on Programming Languages*, *SBLP'12*, Berlin, Heidelberg, pp.42–56, Springer-Verlag (online), DOI: 10.1007/978-3-642-33182-4-5 (2012).

[15] Razdan, R. and Smith, M.D.: A High-performance Microarchitecture with Hardware-programmable Functional Units, *Proc. 27th Annual International Symposium on Microarchitecture*, *MICRO 27*, New York, NY, USA, ACM, pp.172–180 (online), DOI: 10.1145/192724.192749 (1994).

[16] Stephenson, M., Babb, J. and Amarasinghe, S.: Bitwidth Analysis with Application to Silicon Compilation, *SIGPLAN Not.*, Vol.35, No.5, pp.108–120 (online), DOI: 10.1145/358438.349317 (2000).

[17] Yamashita, H., Yasnura, H., Eko, F.N. and Yun, C.: Variable Size Analysis and Validation of Computation Quality, *Proc. IEEE International High-Level Valida-*

© 2018 Information Processing Society of Japan

tion and Test Workshop (*HLDVT'00*), *HLDVT '00*, Washington, DC, USA, IEEE Computer Society, pp.95–100 (online), available from ⟨http://dl.acm.org/citation.cfm?id=518914.822308⟩ (2000).

[18] Budiu, M., Sakr, M., Walker, K. and Goldstein, S.C.: BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations, *Proc. 6th International Euro-Par Conference on Parallel Processing*, *Euro-Par '00*, London, UK, UK, pp.969–979, Springer-Verlag (online), available from ⟨http://dl.acm.org/citation.cfm?id=646665.701200⟩ (2000).

[19] Nayak, A., Haldar, M., Choudhary, A. and Banerjee, P.: Precision and Error Analysis of MATLAB Applications During Automated Hardware Synthesis for FPGAs, *Proc. Conference on Design, Automation and Test in Europe*, *DATE '01*, Piscataway, NJ, USA, pp.722–728, IEEE Press (online), available from ⟨http://dl.acm.org/citation.cfm?id=367072.367906⟩ (2001).

[20] Cao, Y. and Yasuura, H.: Quality-driven Design by Bitwidth Optimization for Video Applications, *Proc. 2003 Asia and South Pacific Design Automation Conference*, *ASP-DAC '03*, New York, NY, USA, ACM, pp.532–537 (online), DOI: 10.1145/1119772.1119883 (2003).

[21] 鈴木　貢，藤波順久，福岡岳穂，渡邊　坦，中田育男：マルチメディア SIMD 命令活用のためのデータサイズ推論，情報処理学会論文誌プログラミング（PRO），Vol.45, No.SIG05 (PRO21), pp.1–11 (2004).

[22] Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R. and Sherwood, T.: Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators, *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, Vol.20, No.11, pp.1355–1371 (online), DOI: 10.1109/43.959864 (2006).

[23] Özer, E., Nisbet, A.P. and Gregg, D.: A Stochastic Bitwidth Estimation Technique for Compact and Low-power Custom Processors, *ACM Trans. Embed. Comput. Syst.*, Vol.7, No.3, pp.34:1–34:30 (online), DOI: 10.1145/1347375.1347387 (2008).

[24] Ball, T. and Larus, J.R.: Branch Prediction for Free, *SIGPLAN Not.*, Vol.28, No.6, pp.300–313 (online), DOI: 10.1145/173262.155119 (1993).

[25] Wagner, T.A., Maverick, V., Graham, S.L. and Harrison, M.A.: Accurate Static Estimators for Program Optimization, *SIGPLAN Not.*, Vol.29, No.6, pp.85–96 (online), DOI: 10.1145/773473.178251 (1994).

[26] Patterson, J.R.C.: Accurate Static Branch Prediction by Value Range Propagation, *SIGPLAN Not.*, Vol.30, No.6, pp.67–78 (online), DOI: 10.1145/223428.207117 (1995).

[27] Calder, B., Grunwald, D., Lindsay, D., Martin, J., Mozer, M. and Zorn, B.: Corpus-based Static Branch Prediction, *SIGPLAN Not.*, Vol.30, No.6, pp.79–92 (online), DOI: 10.1145/223428.207118 (1995).

[28] Deitrich, B., Cheng, B.-C. and Hwu, W.-m.: Improving Static Branch Prediction in a Compiler, *Proc. 1998 International Conference on Parallel Architectures and Compilation Techniques*, *PACT '98*, Washington, DC, USA, IEEE Computer Society, pp.214–221 (online), available from ⟨http://dl.acm.org/citation.cfm?id=522344.825696⟩ (1998).

[29] Buse, R.P.L. and Weimer, W.: The Road Not Taken: Estimating Path Execution Frequency Statically, *Proc. 31st International Conference on Software Engineering*, *ICSE '09*, Washington, DC, USA, IEEE Computer Society, pp.144–154 (online), DOI: 10.1109/ICSE.2009.5070516 (2009).

[30] Calder, B. and Grunwald, D.: Reducing Branch Costs via Branch Alignment, *SIGPLAN Not.*, Vol.29, No.11, pp.242–251 (online), DOI: 10.1145/195470.195553 (1994).

[31] Yang, M., Uh, G.-R. and Whalley, D.B.: Improving Performance by Branch Reordering, *SIGPLAN Not.*, Vol.33, No.5, pp.130–141 (online), DOI: 10.1145/277652.277711 (1998).

[32] Atmel Corporation: AVR Instruction Set Manual (2016). available from ⟨http://www.atmelcom/images/Atmel-0856-AVR-Instruction-Set-Manual.pdf⟩

[33] Microchip Technology Inc.: PICmicro mid-range MCU family reference manual (1997). available from ⟨http://ww1.microchip.com/downloads/en/DeviceDoc/33023a.pdf⟩

[34] Fujitsu Microelectronics Limited: F^{2}MC-16LX 16-bit microcontroller programming manual (2008). available from ⟨http://www.fujitsu.com/downloads/MICRO/fma/mcu/prog16lx-cm44-00201-3e.pdf⟩



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．

© 2018 Information Processing Society of Japan