情報処理学会論文誌　プログラミング　Vol.12　No.1　1–9 (Jan. 2019)

RL78マイコン向け命令スケジューラによるパイプラインストールの削減

千葉 雄司[1,a)]　大石 圭一[2]　永井 佑樹[2]　中川 満[3]

受付日 2018年6月19日, 採録日 2018年9月26日

概要：RL78 マイコンのようなアキュムレータマシンでは，多くの命令において，オペランドとして使えるレジスタがアキュムレータのみとなっており，その結果，命令スケジューリングの際に，命令を単独で移動しようとしても，アキュムレータに関する依存に移動を阻まれることが多い．この問題を回避して命令をスケジュールする方法として，移動の対象を，単独の命令ではなく，命令列，具体的には，アキュムレータに値を代入する命令から，代入した値を最後に利用する命令までの命令列にする方法を提案する．提案した方法を RL78 マイコン向け C コンパイラ製品 CC-RL の命令スケジューラに適用し，その効果を，CoreMark や FFT，SHA256 などのベンチマークによって評価したところ，最大で 2.12%実行を高速化できることが分かった．

キーワード：コンパイラ，最適化，命令スケジューリング，アキュムレータ

Reducing Pipeline Stalls by an Instruction Scheduler for the RL78 Microcontroller

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

Received: June 19, 2018, Accepted: September 26, 2018

Abstract: An accumulator machine such as the RL78 microcontroller impose its instructions to use its accumulator register as one of the operands. This makes the almost all instructions have dependencies to the accumulator and the dependencies prevent the instruction scheduler to move a single instruction elsewhere. To cope with this problem, we propose an instruction scheduling technique to schedule not a single instruction but an instruction sequence which starts from the definition of a value in the accumulator and ends with the last use of the value. We implemented the proposed method in CC-RL, a commercial C compiler for the RL78 microcontroller and evaluated the effect using benchmarks such as CoreMark, FFT and SHA256 to find it improves the performance by 2.12% at most.

Keywords: compiler, optimization, instruction scheduling, accumulator

1. はじめに

パイプライン処理を行うプロセッサ向けの最適化技術に，命令スケジューリングがある．パイプライン処理では，命令を実行する段階で，実行に必要なデータの生成がまだ終っていない場合，終わるまでパイプラインを止めるが，止めると実行効率が低下する．この実行効率の低下を回避する最適化が命令スケジューリングであり，データを生成する命令と，使用する命令の間に，他の命令を配置することで，データの生成を待つ間，プロセッサに他の命令を実施させ，実行効率が低下しないようにする．

しかしながら，命令スケジューリングはつねに実施できるとは限らない．なぜなら命令の間に依存，すなわち実行順序の制約がある場合，命令の並び順を任意には変更でき

[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 187–0022, Japan
[a)] yuji.chiba.wz@bp.renesas.com

```
1:   movw ax, [sp+0x04]
2:   movw de, ax
3:   movw ax, [de] // ストール
4:   movw bc, ax
5:   movw ax, [sp+0x08]
6:   addw ax, 2
7:   movw de, ax
8:   movw ax, [de] // ストール
```

図 1　スケジュール対象の命令列
Fig. 1　Instructions to be scheduled.

ないからである．ここで命令間の依存には，真の依存と偽の依存の 2 種類があり，前者は先行する命令が生成した値を後続する命令が使用することから生まれる依存を表し，後者は先行する命令の読込元と後続する命令の書込先が重複していることから生まれる依存を表す．

たとえば**図 1** に示す RL78 マイコン [1] の命令列については，次のことがいえる．

- 1 行目のロード命令と 2 行目の複写命令は真の依存関係にある．なぜなら 2 行目の複写命令が読み込む値を 1 行目の命令がロードしてるからである．
- 4 行目の複写命令と 5 行目のロード命令は偽の依存関係にある．なぜなら 4 行目の複写命令の読込元と 5 行目のロード命令の書込先がともにレジスタ ax だからである．

ここで偽の依存については，命令の読み書きするレジスタを変更することで解消でき，解消すれば命令の並び順を変更可能になる．読み書きするレジスタの変更を通じて命令を移動可能にする技術は古典的かつ有効だが [2], [3], [4], [5], [6], [7], [8]，変更できるか否かはアーキテクチャに依存しており，たとえばアキュムレータマシンのように，オペランドとして利用可能なレジスタを著しく制限するアーキテクチャでは利用しにくい．

ここでアキュムレータマシンとは，オペランドの 1 つをアキュムレータと呼ぶレジスタにすることを要求するアーキテクチャのことを表す．アキュムレータマシンでは，アキュムレータの読み書きを頻繁に行うため，アキュムレータに関する偽の依存がスケジューリングを大きく制限する一方で，オペランドをアキュムレータ以外のレジスタには変更できないので，スケジュールの余地を広げるには，レジスタの変更以外の対策が必要になる．ここでレジスタの変更以外の対策として，本論文では，命令の移動を命令単位ではなく，命令列単位で実施する技法を提案する．また，提案技法が，アキュムレータマシンの 1 種である RL78 マイコンの性能にもたらす影響を評価し，その結果を示す．本論文では，まず 2 章で RL78 マイコンの特性を示し，次に 3 章で提案技法の詳細を明らかにする．4 章では，スケジューリングの対象を個々の命令とする，従来のスケジューリング方法との関係について述べる．5 章では我々による提案技法の実装について述べる．6 章では提案技法の評価結果を示し，7 章では関連する研究について述べる．8 章は結論である．

2. RL78 マイコンの特性

本章では，まず，RL78 マイコンが提供するレジスタについて述べ，次に，RL78 マイコンのパイプラインがストールする条件を示す．最後に，RL78 マイコンが命令のオペランドに課す制約を示す．

2.1 レジスタ

RL78 マイコンは，8 本の 8 bit の汎用レジスタ a，x，b，c，d，e，h，l を提供し，また，これらを 2 本ずつ組み合わせたレジスタ ax，bc，de，hl を 16 bit の汎用レジスタとして提供する．これらの汎用レジスタのうち，レジスタ a と ax がアキュムレータとしての役割を担う．また，汎用レジスタとは別に，スタックポインタを保持するレジスタ sp を提供する．

2.2 ストールする条件

RL78 マイコンのパイプライン処理では，メモリを参照する命令の直前で，参照先のアドレスの計算につかうレジスタへの代入を行うと，1 サイクルのストールが発生する．たとえば図 1 の命令列では，3 行目と 8 行目の命令でストールが発生する．なぜなら，これらの命令の直前で，ベースレジスタ de への代入を行っているからである．

2.3 オペランドの制約

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している．たとえば図 1 は RL78 マイコンの命令列だが，命令列中のすべての命令がアキュムレータ ax を参照している．

すべての命令がアキュムレータ ax を参照することは，ハードウェアの構造を簡潔にするといった利点をもたらす一方で，多くの命令間に真または偽の依存をもたらし，命令スケジューリングを困難にする．たとえば図 1 の命令列では，どの命令も何らかの形で他の命令に依存しているため，単独で他の場所に移動できる命令は 1 つもない．

なお，RL78 マイコンの命令セットにおいて，命令のどのオペランドをアキュムレータにするかは命令によりけりだが，アキュムレータがメモリ参照のベースレジスタになることはない．ベースレジスタになりうるのはレジスタ bc，de，hl，sp のいずれかであり，アキュムレータはロード命令ではもっぱらロード先，ストア命令ではもっぱらストア元のオペランドになる．

RL78 マイコンのオペランドの制約は，RL78 マイコン向けの命令列に一定の規則をもたらすものでもある．すなわち，RL78 マイコン向けの命令列は，その多くが，次に示

す一連の処理を行う部分的な命令列になる．

- まず最初にアキュムレータに値を代入し
- 必要に応じてアキュムレータ上で演算を行って
- 最後にアキュムレータから他に値を移す

本論文では，この部分的な命令列，より正確には，アキュムレータに値を代入する命令から，アキュムレータ上での値の更新を経て，最後にアキュムレータの値を参照するまでの命令列をバンドルと呼ぶことにする．

3. 命令列単位でのスケジューリング

本論文で提案する命令スケジューリングの技法は，基本ブロック内で命令をスケジュールする技法であり，アキュムレータへの依存が原因で命令を移動できなくなることを回避するために，バンドルを 1 つの命令と見なしてスケジューリングする．バンドルを 1 つの命令と見なしてスケジューリングする理由は，バンドルの前後ではアキュムレータに有意な値が入っていないので，バンドルとバンドルの順番の入れ替えではアキュムレータへの依存を考慮する必要がなくなるからである．

たとえば提案技法で図 1 の命令列をスケジュールする場合，バンドルは次の 3 つとなる．

(1) 1 行目のロード命令と 2 行目の複写命令のペア
(2) 3 行目のロード命令と 4 行目の複写命令のペア
(3) 5 行目のロード命令から 6 行目の加算命令を経て 7 行目の複写命令に至るまでの 3 命令

8 行目のロード命令はアキュムレータへの代入で始まるものの，アキュムレータに代入した値を最後の使用で終端する命令が存在しないのでバンドルにしない．このように，アキュムレータに代入した値を最後の使用で終端しないためにバンドルにならない命令が基本ブロックの末尾に現れ，逆に基本ブロックの先頭には，アキュムレータへの代入で始まらないためにバンドルにならない命令が現れるが，これらの命令とバンドルの順番は入れ替えられない．なぜなら，これらの命令とバンドルの間はアキュムレータに関する依存関係があるからである．

バンドルにまとめたうえでのスケジューリングは，個々のバンドルを 1 つのマクロな命令と見なしてスケジュールする処理と考えることができ，実際我々の実装でもバンドルをマクロな命令に置き換える．たとえば図 1 の命令列のスケジューリングでは，3 つのバンドルをそれぞれ置き換えて**図 2** (a) の命令列とする．図 2 (a) の 1 から 3 行目にある命令には $ax\langle dead\rangle$ というオペランドがついているが，このオペランドは命令の実行中にアキュムレータ ax の内容を破壊することを意味している．このオペランドは，図 2 (a) の 4 行目の命令，すなわち，アキュレータへの代入を行うものの，バンドルにならなかった命令をまたいで，バンドルを下に移動するのを妨げる役割を果す．

図 2 (a) の命令列のスケジューリングでは，何も書き換えなければ，命令間に依存があるため命令の並び順を入れ替えられず，したがって 2 行目と 4 行目で発生するストールを回避することもできない．しかしながら図 2 (a) の命令列にはレジスタを割り付け直す余地があり，具体的には 1 行目の命令が定義して 2 行目の命令が使用するレジスタ de を少なくともレジスタ bc に変更できる．レジスタ bc に変更できる理由は，図 2 (a) の 2 行目でレジスタ bc を使用せずに定義していることから，2 行目に到達した時点でレジスタ bc が空いていることが分かり，なおかつ RL78 マイコンの命令セットでは図 2 (a) に書き換える前の図 1 の 2 行目と 3 行目でレジスタ de の割り付いているオペランドにレジスタ bc を割り付けることが可能だからである．ただし図 1 の 3 行目にあるロード命令のオペランドをレジスタ bc に変更する場合，レジスタ bc にオフセットを加算するアドレシングモードしかないため，オフセット 0x0000 の追加が必要になり，その分，コードサイズが増える．

```
1:  movw de, [sp+0x04], ax⟨dead⟩
2:  movw bc, [de], ax⟨dead⟩ // ストール
3:  addw de, [sp+0x08], ax⟨dead⟩
4:  movw ax, [de] // ストール
```
(a) スケジューリング前

```
5:  movw bc, [sp+0x04], ax⟨dead⟩
6:  movw bc, 0x0000[bc], ax⟨dead⟩ // ストール
7:  addw de, [sp+0x08], ax⟨dead⟩
8:  movw ax, [de] // ストール
```
(b) レジスタの割付直し後

```
9:  movw bc, [sp+0x04], ax⟨dead⟩
10: addw de, [sp+0x08], ax⟨dead⟩
11: movw bc, 0x0000[bc], ax⟨dead⟩
12: movw ax, [de]
```
(c) スケジューリング後

図 2　バンドルにまとめたうえでのスケジューリング
Fig. 2　Scheduling by bundle.

ここでは図 2 (a) の 1 行目と 2 行目の命令が定義/使用するレジスタ de をレジスタ bc に割り付け直すものとし，割り付け直した結果を図 2 (b) に示す．割り付け直した後の命令列では，6 行目の命令にオフセット 0x0000 が追加になった分だけコードサイズは増えているものの，6 行目と 7 行目の命令を並べ替えることが可能になっており，並べ替えを実施した後の図 2 (c) の命令列ではストールを回避できている．

4. 命令単位でのスケジューリングとの関係

3 章で提案した，命令列をバンドルにまとめたうえでのスケジューリングは，バンドルにまとめない状態でのスケジューリング，すなわち命令単位でのスケジューリングと併用すべきものであり，命令単位でのスケジューリングを

```
1:  movw ax, [sp+8]
2:  movw [sp+4], ax
3:  movw de, ax
4:  movw ax, [de] // ストール
```
(a) スケジューリング前

```
5:  movw ax, [sp+8]
6:  movw de, ax
7:  movw [sp+4], ax
8:  movw ax, [de]
```
(b) スケジューリング後

図 3　バンドル内でのスケジューリング
Fig. 3　Scheduling in the bundle.

置き換えるものではない．併用すべきとする理由は，次のスケジューリングが必要となるためである．

- バンドル内でのスケジューリング
- バンドルの途中へのスケジューリング

それぞれのスケジューリングについて順次，詳述する．バンドルにまとめたうえでのスケジューリングと，命令単位でのスケジューリングの併用の方法については次章で述べる．

4.1　バンドル内でのスケジューリング

バンドルにまとめたうえでのスケジューリングでは，バンドル内の命令を並べ替えないが，バンドル内での並べ替えによりストールを回避できることもある．

たとえば**図 3** (a) の命令列について考える．バンドルにまとめたうえでのスケジューリングでは図 3 (a) の 1 から 3 行目の命令列をバンドルと見なし，その内部の命令の並び順を変更しないので，たとえばスケジュール対象の命令が図 3 (a) の 4 命令だけである場合，4 行目の命令で発生するストールを回避できない．これに対し，命令単位でのスケジューリングを併用すれば，図 3 (a) の 2 行目と 3 行目の命令の実行順序を入れ替えた図 3 (b) の命令列を得て，ストールを回避できる．

4.2　バンドルの途中へのスケジューリング

バンドルにまとめたうえでのスケジューリングでは，バンドルの外からバンドルの途中への並べ替えに対応しないが，アキュムレータを参照しない命令が存在する場合，バンドルの途中へのスケジューリングによってストールを回避できることもある．

たとえば RL78 マイコンの命令セットには，スタックポインタ sp の内容をアキュムレータ以外の汎用レジスタにコピーする命令がある．この命令を含む**図 4** (a) の命令列をスケジューリングする場合について考える．バンドルにまとめたうえでのスケジューリングでは図 4 (a) の 4 行目で発生するストールを回避できない．なぜなら，ストールを回避するには 3 行目の命令 `movw hl, sp` を移動する必要があるが，バンドルにまとめたうえでのスケジューリングでは 1，2 行目の命令をバンドルして図 4 (b) の 5 行目の命令にしてしまい，この命令が hl を使用するため，命令 `movw hl, sp` を移動できないからである．これに対し，命令単位のスケジューリングならば，図 4 (a) の 2 行目と 3 行目の命令の並び順を入れ替えた図 4 (c) の命令列を得ることができ，それによってストールを回避できる．

```
1:  movw ax, [hl]
2:  movw de, ax
3:  movw hl, sp
4:  movw ax, [hl] // ストール
```
(a) スケジューリング前

```
5:  movw de, [hl], ax⟨dead⟩
6:  movw hl, sp
7:  movw ax, [hl] // ストール
```
(b) スケジュール対象のバンドル

```
8:  movw ax, [hl]
9:  movw hl, sp
10: movw de, ax
11: movw ax, [hl]
```
(c) 命令単位でのスケジューリング後

図 4　バンドルの途中へのスケジューリング
Fig. 4　Scheduling to the middle of the bundle.

5.　実装

我々はバンドルにまとめたうえでのスケジューリングを，ルネサスエレクトロニクス製の RL78 マイコン向けコンパイラ製品 CC-RL [9] に実装し，その効果を評価した．CC-RL におけるコンパイル処理の流れは参考文献 [11] に示したとおりだが，命令スケジューリングは次に示す 3 つの段階にわけて順に実施している．

(1) レジスタプレッシャを軽減するスケジューラ
ルネサスエレクトロニクス製のコンパイラ製品 CC-RX/RH/RL は実装に llvm-2.3 をベースとするコンパイラプラットフォームを利用している．当該コンパイラプラットフォームでは，コンパイルの途中まで機種非依存の中間表現を利用し，レジスタ割付など機種依存な最適化を実施する前の段階で機種非依存な中間表現から機種依存な中間表現に変換する処理，すなわち命令選択を行うが，命令選択の際にレジスタプレッシャを軽減するスケジューリングもあわせて実施する [10]．このスケジューリングの実装は機種非依存なもので，個々の機種に固有な事情を考慮したものでない．

(2) 割付可能なレジスタを考慮したスケジューラ
RL78 マイコンでは命令のオペランドに任意のレジス

```
1:  bundleInstructions();
2:  eliminateAntiDependency();
3:  shceduleInstructions();
4:  unbundleInstructions();
5:  shceduleInstructions();
```

図 5　スケジューリングのアルゴリズム
Fig. 5　Scheduling algorithm.

タを割り付けられないことに着目し，同じ値を同じレジスタに収めて使い続けることができるよう命令を並べ替える [11]．レジスタ割付の直前に動作し，レジスタ割付の際に，値を別のレジスタに移しかえる命令を挿入するケースを減らす役割を果たす．スケジュールに際しては，アキュムレータへの代入を行う命令と，アキュムレータの内容を更新する命令，アキュムレータの内容を参照する命令を連続して配置することを最も優先し，結果として，本論文での提案技法におけるスケジュール単位であるバンドルを構成する役割も果たしている．

(3) ストールを回避するためのスケジューラ
レジスタ割付の後に動作し，基本ブロック内での命令の配置を，入口側から順に定めてゆくスケジューラである．ストールを回避するために，メモリを参照する命令の直前で参照先のアドレス計算につかうレジスタに代入を行っているか調べ，行っている場合には，代入を行う命令と，メモリを参照する命令の間に別の命令を配置する．

我々は，バンドルにまとめたうえでのスケジューリングを，3 つ目のスケジューラ，すなわちストールを回避するためのスケジューラに実装した．

実装後の 3 つ目のスケジューラの処理の全体像を**図 5** に示す．図 5 の 1 から 4 行目がバンドルにまとめたうえでのスケジューリングにあたり，5 行目が命令単位でのスケジューリングにあたる．バンドルにまとめたうえでのスケジューリングを，命令単位でのスケジューリングより前に実施する理由は，バンドルを構成するのが 2 番目のスケジューラであることから，バンドルの途中に他の命令が入り込む前に，バンドルにまとめたうえでのスケジューリングを実施する方がよいためである．

図 5 の各行で実施する処理について順次述べる．まず 1 行目でバンドルをマクロな命令に書き直す処理，たとえば図 1 の命令列を図 2 (a) の命令列に書き直す処理を行い，次に 2 行目で偽の依存を削除する．偽の依存の削除では，ベースレジスタへの代入を行い，ストールの原因となる命令（機械命令もしくは 1 行目で作成したマクロな命令）のうち，レジスタを割り付け直さない限りほかと順番を変更する余地がないものを求め，レジスタを割り付け直す余地があるなら，割り付け直し，たとえば図 2 (a) から図 2 (b) への書き換えを行うことで，順番の入れ替えを可能にする．続いて 3 行目で命令のスケジューリングを行う．スケジューリングでは，基本的には，命令を前にあるものから順に，順番を変更せずに並べてゆくが，ストールの原因となる，ベースレジスタへの代入命令がスケジュール可能になった場合には，それを優先してスケジュールする．また，ベースレジスタへの代入命令の直後には，ストールを引き起さない命令を優先してスケジュールする．4 行目では，1 行目で作成したマクロな命令を元の命令列に書き戻し，最後に 5 行目で命令単位のスケジューリングを行う．5 行目のスケジューリングの目的は 4 章で述べた，バンドル内でのスケジューリングや，バンドルの途中へのスケジューリングを実施することにあるが，スケジューリングの実施要領は 3 行目と同一である．

6. 評価

本章では，我々が実装した，バンドルにまとめたうえでスケジュールする機能の効果を示す．効果の測定には次に示す 4 つのベンチマークをもちいた．

- CoreMark [12], [13]
- dhrystone [14]
- FFT
- SHA256

ここで 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次のアルゴリズムを構成要素とするもので [15]，組込マイコンやコンパイラのベンダが性能指標として利用している [16], [17], [18], [19], [20], [21], [22].

- 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

また dhrystone はポインタの操作や代入といった一般的なプログラムが行う挙動を模した合成ベンチマークである．FFT は信号処理，SHA256 はセキュリティを必要とする通信の実現に必要なアルゴリズムである．FFT も SHA256 も RL78 マイコンの応用先でよく利用することから，それぞれの実装をルネサスエレクトロニクスが提供している [23], [24]．評価対象のベンチマークはいずれもマイコンのコアで計算をし続けるものであり，したがって本論文で示す評価結果は，コアで計算し続ける場合の性能に関する一定の指標になると考える．

ベンチマークのコンパイルに際しては，実行速度重視の最適化を行うため，コンパイルオプション `-Ospeed` を指定した．`-Ospeed` の指定時は，コードサイズを増やす代りに実行速度を上げる最適化を適用するので，たとえばレジスタ割付や偽の依存の削除の際に図 2 (a) から図 2 (b) への書き換えに示したオフセット 0x0000 の追加を実施することになる．

表 1　命令スケジューリングによる実行サイクル数の改善率
Table 1　Performance improvement by instruction scheduling.

ベンチマーク	ストール率 (%)	高速化率（%） 提案技法の利用 なし	高速化率（%） 提案技法の利用 あり	提案技法の効果 (%)
CoreMark	3.15	0.49	1.96	1.47
dhrystone	0.79	0.13	0.20	0.07
FFT	4.00	0.24	2.36	2.12
SHA256	1.29	0.57	0.85	0.28

表 2　代入の優先スケジュールおよび偽の依存の削除への影響
Table 2　Effect to priority scheduling and false dependency elimination.

ベンチマーク	代入の優先スケジュールの効果（%） 提案技法の利用 あり	代入の優先スケジュールの効果（%） 提案技法の利用 なし	偽の依存の削除の効果（%） あり	偽の依存の削除の効果（%） なし
CoreMark	0.04	0.04	0.75	0.07
dhrystone	0.07	0.00	0.00	0.00
FFT	0.68	0.24	0.55	0.00
SHA256	0.58	0.56	0.03	0.00

ベンチマークの実行にはシミュレータを利用した．シミュレータによる実行サイクル数の測定結果は一定で，実行ごとに変化することはない．

評価の結果を**表 1** に示す．表 1 のストール率は，ストールを回避するための命令スケジューリングを実施しない場合，どれだけストールが発生するかを示す比率で，ストールしたサイクル数を全実行サイクル数で除して求めた．ストール率は，ストールを回避するための命令スケジューリングから得られる性能向上率の上限とも解釈できる．

表 1 の高速化率は，ストールを回避するための命令スケジューリングによって実行サイクル数をどれだけ少なくできたかを示す比率で，提案技法を適用しなかった場合と，した場合の双方を測定した．ここで提案技法を適用しなかった場合とは，個々の命令単位でスケジュールのみを行うこと，すなわち，図 5 の 5 行目の処理のみ実施したことを意味する．また，提案技法を適用した場合とは，本論文で提案したバンドルにまとめたうえでのスケジューリングを併用したこと，すなわち，図 5 の処理をすべて実施したことを意味する．表 1 の提案技法の効果は，提案技法ありの場合と，なしの場合との高速化率の差を示している．

表 1 に示した提案技法の効果の欄から，バンドルにまとめたうえのスケジューリングを併用すると最大で 2.12%実行を高速化できることが分かる．

次に，ストールを回避するためのスケジューラの 2 つの機能，すなわち代入の優先スケジュールと偽の依存の削除の効果に提案技法が与える影響を調査した結果を**表 2** に示す．表 2 に示した，提案技法ありの場合の代入の優先スケジュールの効果とは，提案技法ありの状況下で代入の優先スケジュールの有無が実行サイクル数に及ぼす影響，すなわち次に示す式の値を測定した結果を表す．

$$\frac{\text{代入の優先スケジュールなしの場合の実行サイクル数}}{\text{代入の優先スケジュールありの場合の実行サイクル数}} - 1$$

表 2 の測定にあたっては，測定対象でない方の機能はつねにありとし，たとえば代入の優先スケジュールの効果の測定にあたっては，偽の依存の削除をつねにありとした．

表 2 の測定結果から，2 つの機能の効果は提案技法によって大きくなっていることが分かる．大きくなった理由は，提案技法によって命令移動の自由度が増え，結果として 2 つの機能を適用できる機会が増えたためと考える．

最後に，提案技法を適用してなおストールする原因と，その対策について述べる．表 1 のストール率と高速化率の比較から，バンドルにまとめたうえでのスケジューリングを併用しても，ストールが発生することが分かる．残りのストールの発生原因を，評価したベンチマークのうち最も規模の大きな CoreMark を対象に調査した結果，原因はいずれも移動可能な命令がないことであった．移動可能な命令がない理由のうち，最も大きかったのは，真のデータ依存によるもので，これがストールの発生回数全体の 98.9%を占めていた．残り 1.1%の原因は偽の依存で，空きレジスタがなかったため，3 章で示したレジスタの割り付け直しでは除去できなかったものである．

真のデータ依存によるストールを除去する既存の最適化に，ループ展開 [25], [26] や if 変換 [27] がある．しかしながら，ループ展開を利用するには多くのレジスタが，if 変換を利用するには述語が必要であり，そのどちらも保持しない RL78 マイコンでは，これら既存の最適化によって真のデータ依存によるストールを除去することはできない．RL78 マイコン向けに利用可能な最適化技法として，ストールの原因を調査する過程で見つかったものに，次の 4 つがあった．

- ベースレジスタの入替
- 共通コードの上方集約
- 命令 pop によるスタックフレームの解放
- 演算順序の入替

これらの技法で除去できるストールは，それぞれ CoreMark の残りのストール全体の 2.0/2.0/1.6/0%であり，すべて除去すると CoreMark のスコアを 0.07%改善できる．

ベースレジスタの入替は，ストールの原因になるレジスタとは別に，同じ値を保持しているレジスタがあるなら，そちらをベースレジスタにする技法であり，たとえば**図 6** (a) の命令列を図 6 (b) の命令列に書き換える．ただし，図 6 の (a) から (b) への書き換えは，コードサイズの増加をともなうので，コードサイズの削減の方が性能向上より重要な局面では実施すべきでない．コードサイズが増加する理由は，RL78 マイコンがレジスタ sp をベースレジスタとする相対参照命令に，ディスプレイスメントのない，省サイ

```
1:   movw hl, sp
2:   movw ax, [hl] // ストール
```
(a) 最適化前

```
3:   movw hl, sp
4:   movw ax, [sp+0x00]
```
(b) 最適化後

図 6　ベースレジスタの入替
Fig. 6　Base register replacement.

```
1:       bz BB1
2:   BB0:
3:       movw de, ax
4:       movw ax, [de] // ストール
5:   ···
6:   BB1:
7:       movw de, ax
8:       movw ax, [de+2] // ストール
```
(a) 最適化前

```
 9:      movw de, ax
10:      bz BB1
11:  BB0:
12:      movw ax, [de]
13:  ···
14:  BB1:
15:      movw ax, [de+2]
```
(b) 最適化後

図 7　共通式の上方集約
Fig. 7　Hoisting common expressions.

ズの命令を提供しないためである．

共通式の上方集約は，制御フローの分岐先にある同一の命令を分岐元に引き上げて集約する最適化であり，たとえば**図 7** (a) の 3 行目と 7 行目にある同一の命令 `movw de, ax` を引き上げて図 7 (b) の命令列に書き換える．

命令 `pop` によるスタックフレームの解放は，スタックポインタ `sp` を参照する際のストールを除去するためのものである．RL78 マイコンの提供する関数呼び出しや呼出元に戻るための命令は戻り番地をスタックに書いたり，スタックから読み込んだりするので，その直前で `sp` の値を更新するとストールする．ただし，`sp` の更新に関しては命令 `push` もしくは `pop` で実施すればストールしないという規則があるので，この規則を利用して，たとえば**図 8** (a) の命令列を図 8 (b) の命令列に書き換えるとストールしなくなり，実行効率を改善できる．

演算順序の入替は結合則を満たす演算に適用可能な技法で，演算の順番を入れ替えることでストールを回避する．たとえば**図 9** (a) の命令列では 1 行目でレジスタ `hl` に代入し，その直後の 2 行目でレジスタ `hl` をメモリ参照のベースレジスタとして使っているためストールするが，2 行目

```
1:   movw ax, [sp+0x00]
2:   addw sp, 4
3:   ret // ストール
```
(a) 最適化前

```
4:   pop ax
5:   pop hl
6:   ret
```
(b) 最適化後

図 8　命令 `pop` によるスタックフレームの解放
Fig. 8　Stack frame destruction by `pop` instructions.

```
1:   movw hl, sp
2:   addw ax, [hl+0x00] // ストール
3:   incw ax
```
(a) 最適化前

```
4:   movw hl, sp
5:   incw ax
6:   addw ax, [hl+0x00]
```
(b) 最適化後

図 9　演算順序の入替
Fig. 9　Changing operation order.

と 3 行目で実施する演算がともに加算であるため，演算の順序を入れ替えることが可能であり，入れ替えを行って図 9 (b) の命令列にすればストールがおきなくなる．

7. 関連研究

命令スケジューリングに関する研究はこれまで数多くあるが，アキュムレータマシンのストールの解消を目的とした研究は，我々が調査した限りでは過去に存在しない．その理由としては，命令スケジューリングでストールを抑止できるアーキテクチャが RL78 マイコンのみであることを指摘できる．ローエンドマイコンの中にはアキュムレータマシンが多くあり [1], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], その一部は RL78 マイコンと同様にパイプライン処理を行うが [1], [28], [29], [31], [34], [37]，先行する命令に応じてストールするか否かが変化するものは RL78 マイコンしかない．

また本論文では，ストールを回避するためのスケジューラに実装した 2 つの機能，すなわち，代入の優先スケジュールと偽の依存の削除の効果に，バンドルにまとめたうえでのスケジューリングが与える影響を定量的に評価し，その結果を示した．代入の優先スケジュールや偽の依存の削除は既知の最適化だが，当該最適化の RL78 マイコンでの有効性や，当該最適化がバンドルにまとめたうえでのスケジューリングから受ける影響について論じた論文は過去にない．複数の命令を一まとめにしてスケジューリングする

スケジューラは，過去にも存在し，たとえば 6 章で示した llvm のレジスタプレッシャを軽減するスケジューラもその一例である．しかしながら，当該スケジューラが複数の命令を一まとめにするのは，連続して実行すべき命令列の途中に他の命令をスケジュールしないようにするためであり，ストールを回避するためではない．

8. 結論

アキュムレータマシンにおいて，アキュムレータへの依存を超えて命令をスケジュール可能にすることを目的として，アキュムレータの定義から最後の使用までの命令列をまとめてスケジューリングする技法を提案した．提案技法をルネサスエレクトロニクスの RL78 マイコン向けコンパイラ製品 CC-RL に実装し，その効果を CoreMark や FFT，SHA256 などのベンチマークによって評価したところ，最大で 2.12%，実行を高速化できることが分かった．

参考文献

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

[2] Goodman, J.R. and Hsu, W.-C.: Code Scheduling and Register Allocation in Large Basic Blocks, *Proc. 2nd International Conference on Supercomputing*, *ICS '88*, pp.442–452, ACM (online), DOI: 10.1145/55364.55407 (1988).

[3] Bradlee, D.G., Eggers, S.J. and Henry, R.R.: Integrating Register Allocation and Instruction Scheduling for RISCs, *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS IV*, pp.122–131, ACM (online), DOI: 10.1145/106972.106986 (1991).

[4] Pinter, S.S.: Register Allocation with Instruction Scheduling, *Proc. ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, *PLDI '93*, pp.248–257, ACM (online), DOI: 10.1145/155090.155114 (1993).

[5] Berson, D.A., Gupta, R. and Soffa, M.L.: Integrated Instruction Scheduling and Register Allocation Techniques, *Proc. 11th International Workshop on Languages and Compilers for Parallel Computing*, *LCPC '98*, pp.247–262, Springer-Verlag (1999) (online), available from ⟨http://dl.acm.org/citation.cfm?id=645676.663774⟩.

[6] Inagaki, T., Komatsu, H. and Nakatani, T.: Integrated Prepass Scheduling for a Java Just-In-Time Compiler on the IA-64 Architecture, *Proc. International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, *CGO '03*, pp.159–168, IEEE Computer Society (2003) (online), available from ⟨http://dl.acm.org/citation.cfm?id=776261.776279⟩.

[7] Shobaki, G., Shawabkeh, M. and Rmaileh, N.E.A.: Preallocation Instruction Scheduling with Register Pressure Minimization Using a Combinatorial Optimization Approach, *ACM Trans. Archit. Code Optim.*, Vol.10, No.3, pp.14:1–14:31 (online), DOI: 10.1145/2512432 (2013).

[8] Lozano, R.C.N., Carlsson, M., Blindell, G.H. and Schulte, C.: Register Allocation and Instruction Scheduling in Unison, *Proc. 25th International Conference on Compiler Construction*, *CC 2016*, pp.263–264, ACM (online), DOI: 10.1145/2892208.2892237 (2016).

[9] 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⟩.

[10] Sethi, R. and Ullman, J.D.: The Generation of Optimal Code for Arithmetic Expressions, *J. ACM*, Vol.17, No.4, pp.715–728 (online), DOI: 10.1145/321607.321620 (1970).

[11] 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).

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

[13] The Embedded Microprocessor Benchmark Consortium: eembc/coremark (2018), available from ⟨https://github.com/eembc/coremark/⟩.

[14] Weicker, R.P.: Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules, *SIGPLAN Not.*, Vol.23, No.8, pp.49–62 (online), DOI: 10.1145/47907.47911 (1988).

[15] Gal-On, S. and Levy, M.: Exploring CoreMark — A Benchmark Maximizing Simplicity and Efficacy (2009), available from ⟨https://www.eembc.org/techlit/coremark-whitepaper.pdf⟩.

[16] ARM Limited: CoreMark Benchmarking for ARM Cortex Processors (2013), available from ⟨http://infocenter.arm.com/help/topic/com.arm.doc.dai0350a/DAI0350A_coremark_benchmarking.pdf⟩.

[17] Green Hills Software: Green Hills Software Announces Record EEMBC CoreMark Benchmark Results (2012), available from ⟨https://www.ghs.com/news/201201031_ARM_TechCon_EEMBC.html⟩.

[18] IAR Systems: IAR Systems earns best-ever CoreMark scores on STMicroelectronics' STM32 ARM Cortex-M4 (2012), available from ⟨https://www.iar.com/about-us/newsroom/press/?releaseId=1082053⟩.

[19] Microchip Technology Inc.: SAM S MCUs (2018), available from ⟨http://www.microchip.com/design-centers/32-bit/sam-32-bit-mcus/sam-s-mcus/⟩.

[20] MIPS: Benchmarks (2017), available from ⟨https://www.mips.com/develop/tools/benchmarks/⟩.

[21] RENESAS Electronics：高性能かつ低消費電力を実現する産業機器向け大容量メモリ搭載マイコン「RX64M グループ」を開発 (2014)，入手先 ⟨https://www.renesas.com/jp/ja/about/press-center/news/2014/news20140226.html⟩.

[22] TEXAS INSTRUMENTS: CC2652R SimpleLink Multi-protocol 2.4-GHz Wireless MCU (2018), available from ⟨http://www.ti.com/lit/ds/symlink/cc2652r.pdf⟩.

[23] RENESAS Electronics：FFT ライブラリ (2014)，入手先 ⟨https://www.renesas.com/jp/ja/products/software-tools/software-os-middleware-driver/dsp-fft/fft-library.html⟩.

[24] RENESAS Electronics：暗号ライブラリ (2016)，入手先 ⟨https://www.renesas.com/jp/ja/products/software-tools/software-os-middleware-driver/security-crypto/crypto-library.html⟩.

[25] Dongarra, J.J. and Hinds, A.R.: Unrolling Loops in Fortran, *Software-Practice and Experience*, Vol.9, No.3, pp.219–226 (1979).

[26] Domagala, L., van Amstel, D., Rastello, F. and Sadayappan, P.: Register Allocation and Promotion Through Combined Instruction Scheduling and Loop Unrolling, *Proc. 25th International Conference on Compiler Construction*, *CC 2016*, pp.143–151, ACM (online), DOI: 10.1145/2892208.2892219 (2016).
[27] Allen, J.R., Kennedy, K., Porterfield, C. and Warren, J.: Conversion of Control Dependence to Data Dependence, *Proc. 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, *POPL '83*, pp.177–189, ACM (online), DOI: 10.1145/567067.567085 (1983).
[28] Freescale Semiconductor, Incorporated: M68HC16 CPU16 reference manual (1997), available from ⟨http://www.nxp.com/docs/en/reference-manual/CPU16RM.pdf⟩.
[29] セイコーエプソン株式会社：SIC63000 コア CPU マニュアル (2001)，入手先 ⟨http://www.epson.jp/prod/semicon/pdf/id001212.pdf⟩.
[30] STMicroelectronics: ST7 family programming manual (2005), available from ⟨http://www.st.com/resource/en/programming_manual/cd00004607.pdf⟩.
[31] Freescale Semiconductor, Incorporated: HCS08 family reference manual (2007), available from ⟨http://www.nxp.com/docs/en/reference-manual/HCS08RMV1.pdf⟩.
[32] Maxim Integrated: MAXQ family user's guide (2008), available from ⟨http://pdfserv.maximintegrated.com/en/an/AN4811.pdf⟩.
[33] 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⟩.
[34] Fujitsu Microelectronics Limited: F^{2}MC-16FX 16-bit microcontroller programming manual (2010), available from ⟨http://www.fujitsu.com/downloads/MICRO/fma/mcu/prog16fx-cm44-00203-3e.pdf⟩.
[35] The Western Design Center, Incorporated: W65C02S 8-bit microprocessor (2016), available from ⟨http://www.westerndesigncenter.com/wdc/documentation/w65c02s.pdf⟩.
[36] The Western Design Center, Incorporated: W65C816S 8/16-bit microprocessor (2016), available from ⟨http://www.westerndesigncenter.com/wdc/documentation/w65c816s.pdf⟩.
[37] Silicon Laboratries Inc.: EFM8 Busy Bee family EFM8BB3 reference manual (2017), available from ⟨https://www.silabs.com/documents/public/reference-manuals/efm8bb3-rm.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．

大石 圭一

1985 年生．2010 年東京工業大学大学院総合理工学研究科知能システム科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．

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．