情報処理学会論文誌 Vol.63 No.4 1019–1028 (Apr. 2022)

推薦論文

近似の積極性を動的制御可能なアーキテクチャのためのコンパイラフレームワーク

冨田 和孝[1,a)] 中村 朋生[1,b)] 小泉 透[1,c)] 出川 祐也[1,d)] 入江 英嗣[1,e)] 坂井 修一[1,f)]

受付日 2021年7月14日, 採録日 2022年1月11日

概要：Approximate Computing は，計算精度と引き換えに計算効率と電力効率の両方を改善する手法である．Approximate Computing では，誤差の本当の許容範囲は入力やユーザ状態によって変化するので実行時まで分からない．そこでプログラムの繰返し構造に着目して実行時に多段階に近似の積極度を調整可能な方式が提案されている．この方式では，ハードウェアとソフトウェアの協調が前提とされているが，様々なアプリケーションに適用するためのソフトウェア技術はまだ明らかにされていない．そこで本研究では，この方式を実際のベンチマークに適用する際の近似ルーチンの書き方，オーバヘッドの少ない実行を可能とするための詳細命令セット，その命令セットに対応したコンパイラアルゴリズムを提案する．複数のベンチマークプログラムを対象に評価を行い，提案コンパイラによりプログラマの意図どおりの近似手法を適用する実行バイナリが生成されることを確認した．また，シミュレーション評価により，動的に近似度を変更して精度と効率のトレードオフを変更可能であることを確認した．本研究により，確率的分岐命令を扱うコンパイラアルゴリズムが明らかになり，動的な近似制御を目的とした自由な近似ルーチンを高級言語により記述可能となった．

キーワード：アプロキシメートコンピューティング，コンパイラ，近似計算

A Compiler Framework for Architectures with Dynamically Controllable Aggressiveness of Approximation

Kazuki Tomida[1,a)] Tomoki Nakamura[1,b)] Toru Koizumi[1,c)]
Yuya Degawa[1,d)] Hidetsugu Irie[1,e)] Shuichi Sakai[1,f)]

Received: July 14, 2021, Accepted: January 11, 2022

Abstract: One of the paradigms to improve computational speed and power efficiency Approximate Computing, which aims to improve computational accuracy in exchange. In this method, it is necessary to keep the error caused by the decrease in computational accuracy within the range that users can tolerate. In this study, we focused on the fact that the acceptable range of error may vary depending on human subjectivity. We propose a compiler framework for using an architecture that can dynamically control the calculation accuracy and an approximation method that can change the calculation accuracy step by step. We measured the change in the number of instructions executed and the output error without approximation by varying the computation accuracy step by step for an application. The proposed framework enables the execution of the same program with different accuracy, different number of instructions, and different output errors, and enables run-time switching of the approximation accuracy.

Keywords: approximate computing, compiler, loop optimization

1. はじめに

Approximate Computing（AC）の適用は，計算を近似することでその精度と引き換えに消費電力削減と実行速度

[1] 東京大学
The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] tomida@mtl.t.u-tokyo.ac.jp
[b)] tomokin@mtl.t.u-tokyo.ac.jp
[c)] koizumi@mtl.t.u-tokyo.ac.jp
[d)] degawa@mtl.t.u-tokyo.ac.jp
[e)] irie@mtl.t.u-tokyo.ac.jp
[f)] sakai@mtl.t.u-tokyo.ac.jp

本論文の内容は 2020 年 9 月の FIT2020 第 19 回情報科学技術フォーラムで報告され，同プログラム委員長により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である．

の向上を同時に達成する [1], [2]．AC 技術を用いることで，プログラムから不必要な高精度計算が取り除かれ，サーバマシン上での大規模な消費電力削減や，エッジデバイス上の限られた計算資源下での高い実行性能を可能とする．

しかし，AC 技術を実アプリケーションに適用する際には，AC 技術による計算精度の低下をユーザが許容できるかが問題となる．その許容される計算精度の範囲はユーザの主観によって動的に変化することが知られている [3]．そのため，静的な計算精度の推定 [4] やコンパイル時に近似精度を決定するフレームワーク [5] では，安全を見込んで本来許容される範囲よりも限定的な近似を行わざるを得ないこと，一方でユーザの許容できない想定外の精度低下の可能性が排除できないこと，双方が課題となっている．

この課題の解決のために，近似の積極性を動的に制御するアーキテクチャ [6] が提案されている．このアーキテクチャは，既存の AC 技術とは異なり，アプリケーション実行中に近似の積極性を調整し，そのアプリケーションの出力をアプリケーションユーザの許容範囲に収めることを可能にするものであり，動画やゲームなどのように出力の精度が一定時間ごとに確認されるアプリケーションで特に有効である．この手法では，近似の積極性を特殊レジスタ[*1]に記憶して，CSR の値を利用してアプリケーションの動作を変化させる．CSR の値を外部からの入力によって動的に変更することで，実行時の動的な近似の積極性の制御を可能としている．これにより，プログラマは精度保証の課題から解放され，近似可能箇所の指定と近似ルーチンの記述に専念することができる．

当該アーキテクチャは，近似の積極性を動的に調整するための命令である**確率的分岐命令**を含む．この命令は，近似の積極性を示す CSR の値によって分岐先がランダムに変化する命令である．また，これを利用した近似手法として Loop Body Switching（LBS）というループへの近似が提案されている．ここで，LBS は元のループボディと近似されたループボディを確率的に切り替えて実行する手法である．ハードウェアは確率的分岐命令を特殊扱いすることで LBS が適用されたプログラムを効率良く実行することができる．

LBS の枠組みと確率的分岐命令を含むコードを生成可能なコンパイラがあれば，AC を想定したアプリケーションの開発効率を飛躍的に高めることができる．しかし，確率的分岐命令は新しい振舞いの命令であり，既存のコンパイラの枠組みで取り扱うことが困難である．LBS が適用されたループや確率的分岐命令を含んだバイナリを生成するコンパイラアルゴリズムはこれまで明らかにされておらず，LBS を適用した確率的分岐命令を含む機械語プログラムを得るためには直接アセンブリ命令を記述する必要があった．

そこで我々は，確率的分岐命令を利用した機械語プログラムを生成可能なコンパイラを提案する．このコンパイラは pragma を用いる簡便なコード規約と，確率的分岐命令を利用した LBS をループに適用するためのコード変形を備えている．プログラマが近似計算の対象とするルーチンを含むループ構造の直前に *pragma* を加筆するだけで，LBS を適用するためのコード変形をコンパイラが自動的に行う．コンパイラは，プログラマが用意した近似ルーチンと無近似ルーチンとを両方コンパイルし，そのどちらを実行するか選択する分岐命令として確率的分岐命令を挿入するなど，適切なコード変形を行う．

本論文の貢献は以下のとおりである：

- Loop Body Switching（LBS）のための情報を伝える pramga を用いた簡便なコード規約を提案した．
- 確率的分岐命令を利用した機械語プログラムを生成するためのコード変更アルゴリズムを示し，これを含むコンパイラを開発した．
- 提案したコンパイラを用いて複数のベンチマークをコンパイルし，精度と実行速度のトレードオフ関係が成立することを確認した．

本論文の以降の構成は以下のとおりである．2 章で，近似の積極性を動的制御可能なアーキテクチャと LBS について述べる．3 章で，提案するコード規約とコンパイル技術について説明し，4 章でその実装と評価結果を示す．5 章で関連研究について述べ，6 章で本論文の結論を述べる．

2. 背景技術

本章では，本研究で対象としている近似積極性を動的制御可能なアーキテクチャ [6]，およびそのアーキテクチャ上で利用できる近似手法である Loop Body Switching と，アーキテクチャ技術である確率的分岐命令について述べる．

近似積極性を動的制御可能なアーキテクチャは，近似の積極性をハードウェアの内部ステートとして持ち，この積極性を通じて実行時の振舞いを変化させる．近似の積極性はプログラム中の命令や，外部割り込み，ハードウェアによる検知で内部ステートが書き換えられることによって変更される．

2.1 Loop Body Switching

この手法は，**図 1**，**図 2** に示すように近似ループボディを生成し，一部のイテレーションにおいて近似ループボディを元のループボディの代わりに実行するものである．ループボディをイテレーションごとに選択できるように，ループの変形を行う必要がある．

図 2 の変更後のループ構造は，正確なループボディと近似されたループボディ，ループ選択部分に分かれている．if 文の条件節では，近似の積極性に応じて確率的に近似ループボディを選択するようになっている．N は最大の近似の積極性である．

[*1] 以下では，当該アーキテクチャを提案した論文の記述に従い control state register（CSR）と記述する．

```
0 for( int i = 0; i < n; i++ ) {
1   a[i] = func(i,...);
2 }
```

図 1 近似対象のループの例
Fig. 1 Example of an approximate target loop.

```
0 for( int i = 0; i < n; i++ ) {
1   if( approx_level < rand() % N){
2     // Regular Loop Body
3     a[i] = func(i,...);
4   } else {
5     // Approximate Loop Body
6     a[i] = a[i-1]
7   }
8 }
```

図 2 変形されたループの例
Fig. 2 Example of a transformed loop.

```
approxbr <offset to Approximate Loop Body>
```

図 3 approxbr instruction
Fig. 3 approxbr instruction.

2.2 確率的分岐命令

近似の積極性を動的制御可能なアーキテクチャでは，オーバヘッドの少ないループボディの選択を行うために，Loop Body Switching のループ選択部分の処理をハードウェアで実行する．ループボディの選択は，内部ステートによって決定される．この内部ステートの値を用いてループボディを選択するための命令として確率的分岐命令が定義されている．確率的分岐命令は，**図 3** のように近似されたループボディのアドレスの差分を即値として持つものとして定義されている．

図 2 のループ選択部分では，変数 approx_level を用いてどちらのループボディを用いるか決定する．確率的分岐命令を活用して Loop Body Switching を行うためには，この approx_level を用いた分岐で生成される条件分岐命令を含む処理を確率的分岐命令に変更することが必要である．したがって，アセンブリを直接書き換えるなどの作業が必要であり，プログラマにかかる負担は大きい．

3. 提案手法

Loop Body Switching のプログラマの負担を減らし近似の積極性を動的制御するため，近似の積極性を動的制御可能なアーキテクチャと協調したコンパイラを提案する．近似の積極性を動的制御可能なアーキテクチャでは，2 章で述べたように，新たな分岐命令として確率的分岐命令を追加している．これは，意味的には条件分岐命令であるが，

```
0 #pragma approx
1 for( int i = 0; i < n; i++ ) {
2   // Loop Body
3 }
```

図 4 for 文に対する pragma の挿入の例
Fig. 4 Example of inserting a pragma for a for statement.

命令の見た目としては無条件分岐命令である．命令を見ただけでは，どちらに分岐するか分からない．このことは，1 命令では表しきれない情報を持っていることになる．

提案するコンパイラはプログラマにかかる負担の削減のために以下の特徴を持つ．

- pragma を用いた Loop Body Switching を行うループの決定
- ループ構造の変更

コンパイラがこれらの項目を自動で行えるようになれば，プログラマはアセンブリの編集という労力のかかる作業を行うことなく，ソースコード中に近似されたループボディを追加するだけで様々なアプリケーションに LBS を適用できる．

3.1 Loop Body Switching を行うループの決定

近似したいループをプログラマが指定するための手法として，本研究では pragma を用いる．その理由は，プログラマができる限り少ないソースコードへの変更で動的な近似の積極性の制御によるプログラムの出力の調整を可能にするためである．本コンパイラに新しく追加した pragma は 2 種類ある．1 つ目は for 文に対するものであり，近似ルーチンとして何も行わないものを生成する．2 つ目は if 文に対するものであり，近似ルーチンをプログラマが書くことができる方法である．

3.1.1 for 文に対する pragma

図 4 に pragma の書き方の例を示す．

このように書かれた場合，確率的近似命令はもとのルーチンと何も行わない近似ルーチンを選択することになる．つまり，イテレーションが確率的に飛ばされることになる．

3.1.2 if 文に対する pragma

if 文に対する pragma は，指定された if 文によって生成される条件分岐命令を対象のアーキテクチャで実装されている approxbr 命令に変更を行う．適用の例を**図 5** に示す．pragma の下の if 文の条件節には，それと分かる定数表現を用いる．コンパイル時には条件節は無視され，approxbr 命令が生成される．else 節が近似の場合に選択されるループボディとなる．

3.2 命令の定義と変形後のループ構造

approxbr 命令を用いるためにループ構造に対して**図 6** のような変形を行う．これは，前述の 2 種類の pragma どち

```
0 for (i=0; i < N; i++) {
1       #pragma approx branch
2       if(1){
3           //regular routine
4       } else {
5           //approximate routine
6       }
7 }
```

図 5 if 文に対する pragma の挿入の例
Fig. 5 Example of inserting a pragma for an if statement.

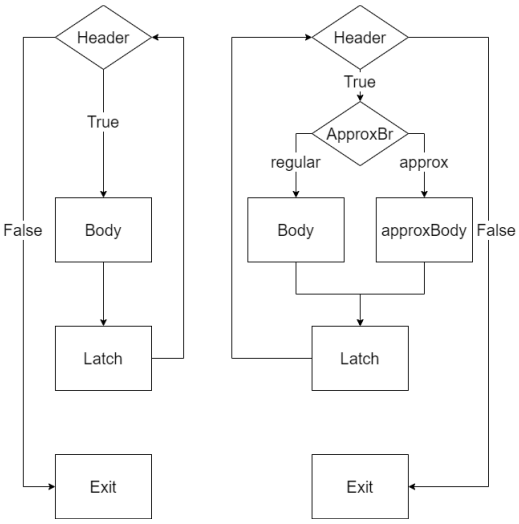

図 6 通常のループ構造（左）と変形後のループ構造（右）
Fig. 6 Regular loop structure (left) and transformed loop structure (right).

```
approxbr <ref to approximate loop body> <ref to regular loop body>
```

図 7 approxbr instruction IR
Fig. 7 approxbr instruction IR.

らを用いてもこのような構造の変換が起きる．本研究ではこの変形を LLVM のミドルエンドにおける最適化パスを用いて行う．そのため，LLVM IR における approxbr の中間表現を定義する．LLVM IR においては，ある BasicBlock から遷移する BasicBlock への参照を命令が持っていなくてはならない．そのため，**図 7** のように第 1 引数に近似されたループボディへの参照，第 2 引数に正確なループボディへの参照を持つものとする．

今回のループボディの変形のアルゴリズムのために以下のようにいくつかの言葉を定義する．

- ループのそれぞれの BasicBlock の名前
  - Header：HeaderBB
  - Body：exBodyBB
  - Latch：LatchBB
  - Exit：ExitBB
- 新しく生成される BasicBlock の名前
  - approxbr 命令を含む BasicBlock（ApproxBr）：AxBrBB
  - ループボディを近似したもの（approxBody）：axBodyBB

3.3 ループ構造の変更

ループ構造の変更時には，approxbr 命令と，近似したループボディの生成，挿入が行われる．

正確なループボディと近似されたループボディへの分岐命令である approxbr 命令は，それらループボディの直前に挿入すればよい．そのようにすれば，近似対象のループを決定するだけで，自動的に approxbr 命令の挿入位置が決定する．

近似されたループボディは，図 5 で if 文の else 節に記述されたものを用いる．

以下にループ構造の変形のアルゴリズムを示す．

(1) axBodyBB を生成
(2) axBodyBB に axBodyBB から LatchBB への無条件分岐命令を追加
(3) AxBrBB を生成
(4) AxBrBB に exBodyBB と axBodyBB への分岐を持つ approxbr 命令を追加
(5) HeaderBB の exBodyBB と ExitBB への条件分岐命令を削除
(6) HeaderBB に AxBrBB と ExitBB への条件分岐命令を追加

これによって，図 6 の変形を行うことができる．exBodyBB は，HeaderBB の持つ分岐先の BasicBlock である．したがって，axBodyBB に制御を移すことで，ループボディ全体の書き換えが可能になる．

4. 評価

4.1 評価方法

本研究で対象としているプロセッサはプログラムの実行中に近似の積極性を変更してプログラムの実行速度と，出力の精度を変更する．実行されるプログラムのバイナリは単一のバイナリであり，そのバイナリは近似の積極性によって動作が変わる．近似の積極性によって実行サイクル数，出力の精度が変化する．近似の積極性が大きくなるほど，サイクル数は小さくなり，出力の精度は悪くなる．近似の積極性を変化させるのは出力の精度に対するユーザからのフィードバックなどである．提案コンパイラは対象のプロセッサに対して 2 つの機能を提供する．1 つ目は，単一のバイナリで様々な近似の積極性での実行を可能にすることである．2 つ目は，生成したバイナリは近似の積極性の変化によって実行速度と出力の精度が変化することである．

このことを確認評価するために，複数のベンチマークを対象として実験を行う．実験を行うベンチマークは，ユー

ザからのフィードバックによって近似の積極性が変化する時間に比べて実行時間は十分短いので，実行中の近似の積極性は一定と見なして実験を行った．近似の積極性ごとにプログラムを実行し，近似の積極性ごとに実行サイクル数，出力の誤差の変化を測定した．

4.1.1 提案コンパイラの実装

提案コンパイラは，最適化コンパイラ基盤である LLVM のバージョン 10.0.0 をベースとして開発した．独自に定義した pragma を解釈するため，C 言語フロントエンドである clang のパーサーに変更を加えた．また，pragma で指定されたループに LBS を適用するコード変更は，ミドルエンドの最適化パスとして実装した．

ソースコードに含まれる pragma の情報は，LLVM に含まれる Metadata と呼ばれる仕組みを用いて受け渡すようにした．pragma は clang のパーサーが解釈し，近似したいループに Metadata が付与された中間表現（LLVM IR）を出力する．ミドルエンドは LLVM IR を読み込み，プログラマが指定したループを当該 Metadata が付与されているかで判別する．判別されたループはミドルエンドの最適化パスで，3 章で述べたようなコード変形を受ける．

4.1.2 ベンチマークプログラム

近似計算用のベンチマークスイートである Axbench と Phoenix から**表 2** に示した 5 つのアプリケーションを選択した．これらのベンチマークは，すべて C 言語で記述されている．

ベンチマークプログラムに提案手法を適用するため，2 通りの方法でベンチマークコードを一部改変した．

1 つ目は，補完なし近似ルーチンを記述するための改変である．**図 8** に，ベンチマークプログラム histogram に変更を加えた後のコードを示す．このように，補完なし近似ルーチンを記述するためには，for 文の直前に pragma を挿入するだけでよい．

2 つ目は，補完あり近似ルーチンを記述するための改変である．本実験では，補完あり近似ルーチンとして，1 つ前の正しく実行されたイテレーションの結果を使用して計算処理を削減するものを採用した．**図 9** に，ベンチマークプログラム histogram に変更を加えた後のコードを示す．このように，補完あり近似ルーチンを記述するためには，ダミーの if 文を挿入し，その直前に pragma を挿入する．ここで，then 節側が元のループボディであり，else 節側が近似ループボディである．その他，1 つ前の正しく実行されたイテレーションの結果を使用するため，正しく実行されたイテレーションやループ外にも適宜コード追加が必要となる．

これらの変更はベンチマークプログラムごとに適切な箇所に適用する必要があるが，従来はアセンブリコードを記述したりバイナリを直接書き換えたりするなどの手法が必要であったのと比べると，プログラマへの負担がはるかに軽減されている．

表 1 実験に用いたアーキテクチャのパラメータ
Table 1 Parameters of the architecture used in the experiment.

Modules	Parameter
Issue Width	2 way
Order	InO
Scheduler Size	64 entries
Branch Predictor	gshare

Memory	Capacity	Cache line Size	associativity	Latency
L1I Cache	8 KiB	64B	2-way	2 cycles
L1D Cache	8 KiB	64B	4-way	3 cycles
L2 Cache	64 KiB	64B	4-way	9 cycles
Main Memory	Infinite	-	-	120 cycles

表 2 実験に用いたベンチマーク
Table 2 Benchmarks used in the experiments.

Name	Suite	Input	Output
Histogram	Phoenix	256 x 256 RGB image	256 x 3 histogram
Matrix multiply	Phoenix	200 x 200 matrix	200 x 200 matrix
Linear regression	Phoenix	252K points	Y = AX + B
Principal Component Analysis	Phoenix	200 x 200 matrix	200 x 200 matrix
Jpeg	Axbench	512 x 512 RGB image	512 x 512 jpeg image

```
0  #pragma approx
1  for (i=*data_pos; i < finfo.st_size; i+=3) {
2    unsigned char *val =
3                  (unsigned char*)&(fdate[i]);
4    blue[*val]++;
…    …
7    val = (unsigned char*)&(fdate[i]);
8    red[*val]++;
9  }
```

図 8 for 文に対する pragma の適用例
Fig. 8 Example of applying pragma to a for statement.

```
0  unsinged char *reg_b, *reg_g, *reg_r;
1  for (i=*data_pos; i < finfo.st_size; i+=3) {
2      #pragma approx branch
3      if(1){
4          //regular routine
5          reg_b = (unsigned char*)&(fdate[i]);
6          blue[*reg_b]++;
7          …
8          reg_r = (unsigned char*)&(fdate[i]);
9          red[*reg_r]++;
10     } else {
11         //approximate routine
12         blue[*reg_b]++;
13         green[*reg_g]++;
14         red[*reg_r]++;
15     }
16 }
```

図 9 if 文に対する pragma の適用例
Fig. 9 Example of applying pragma to an if statement.

4.1.3 シミュレーション実行環境

提案したコンパイラによって生成したバイナリを用いて，シミュレーション評価を行った．鬼斬弐 [7] をベースとして，LBS をサポートするシミュレータを作成した．評価に用いたプロセッサパラメータは，ARM Cortex A53 をベースとして決定した．その詳細を**表 1** に示す．

近似の積極性には，0〜31 の 32 段階で示される値を用いた．近似の積極性 0 は近似ループボディをまったく使用しない，つまり近似をまったく行わないことを意味する．また，近似の積極性 31 は最大限の近似を行うことを意味する．これらの近似の積極性（level）は，実行時に以下の処理で利用される．

- 近似ループボディの実行確率：$level/32$
- 正規のループボディの実行確率：$1 - (level/32)$

これを実現するため，approxbr 命令の処理の際，ハードウェア内の擬似乱数生成器の値と CSR の値を用いて，正規のループボディと近似されたループボディを確率的に選択した．

実行したバイナリの性能を示す指標として，シミュレーションにかかる実行サイクル数と出力値の真の値に対する誤差を用いた．誤差を示す指標として，Normalized Mean Square Error（NMSE）を用いた．出力が 2 次元配列として得られるベンチマークである histogram，matrix multiply，principal component analysis，jpeg では，近似をしない実行結果とそれぞれの近似積極性での実行結果に対して NMSE を計算した．出力自体が回帰直線である linear regression では，入力座標群と出力された回帰直線に対して NMSE を計算した．

4.2 シミュレーション実行結果

本シミュレーションは，次の 2 つを確認することを目的とする．1 つ目は，提案コンパイラが出力した単一のバイナリが複数の近似の積極性で実行可能であること．2 つ目は，与える近似度の値の大小と，実際に行われる近似実行の効果の大小が一致すること．すなわち，近似度を保持するレジスタの値がより大きくなると，より実行速度があがり，出力誤差はより大きくなる傾向があること．実行速度を表す指標として実行サイクル数，出力誤差を表す指標として NMSE を用いた．

4.2.1 実行サイクル数

近似の積極性ごとの実行サイクル数の変化を**図 10** に示す．実行サイクル数は近似の積極性によって変化した．すべてのベンチマークにおいて，近似の積極性が大きくなると実行サイクル数は減少した．補完ありの場合はどのプログラムにおいても近似の積極性を上げるにつれ実行サイクル数が線形に減少した．補完を行わずにスキップを行った場合も，補完ありと同じような変化を示した．

4.2.2 誤差評価

結果を**図 11** に示す．近似の積極性が増加すると，誤差は大きくなる傾向にある．しかし，誤差の変化は実行サイクルのように線形ではない．近似によって変化する値が誤差に直接関係するものではないため，非線形な変化になっている．補完ありの場合，スキップした場合，どちらも近似の積極性が大きくなると誤差は大きくなり，近似の積極性が小さくなれば誤差は小さくなる．

4.2.3 実行速度と誤差の関係

近似の積極性に応じた実行サイクル数，誤差の変化を確認した．積極的な近似によって実行サイクル数が小さくなると，計算誤差は大きくなる．したがって，この結果からプログラムの出力のより大きな誤差を許容するほど，プログラムの実行速度は速くなることを確認した．

4.3 考察

4.3.1 実行サイクル数

実行サイクル数は，近似の積極性に応じておおよそ線形に変化している．この変化の仕方には近似ルーチンの有無が大きくかかわっている．近似ルーチンによってはhistogram のように差が開いていく場合もあるし，jpeg などのように変化がほぼない場合もある．

4.3.2 誤差評価

誤差のグラフは，実行サイクル数とは異なり線形な特徴を示さないことが多い．また，近似ルーチンとして何も行わない場合と，前回のイテレーションの結果を利用する場合は，大きく異なる．もともとスパースなデータを入力とするものは近似ルーチンとして何も行わないならば，データをいくつか抜くだけにとどまる．逆に前回のイテレーションを利用すると，データの集まりに偏りが生まれ誤った結果を生じる可能性がある．画像などのように局所的にみると同じようなデータが並んでいる場合は，前回のイテレーションを利用することで誤差の変化は緩やかなものになる．たとえば，histogram では，近似ルーチンを用意した場合に誤差が大きく減少した．このアプリケーションでは，何も近似ルーチンで行わない場合は，空白の画素がカウントされる．また，前回のルーチンを利用すれば隣り合う画素を利用する．そのため，画像の局所性などが利用でき，誤差が大きく減少したと考えられる．

4.3.3 提案コンパイラによる最適化への影響

提案コンパイラの最適化への影響は，提案手法の近似積極度 0 の結果と提案手法なしの結果の比較により確認できる．今回のコンパイラでは，近似対象のループ部分での最適化を適用していない．したがって，最適化をしていない場合のバイナリの振舞いから最適化への影響を確認する．補完なしの場合は，提案手法なしと比べて実行サイクル数は大きな差はない．そのため，補完なしの場合の最適化への影響は少ないことが分かる．一方，一部のベンチマーク

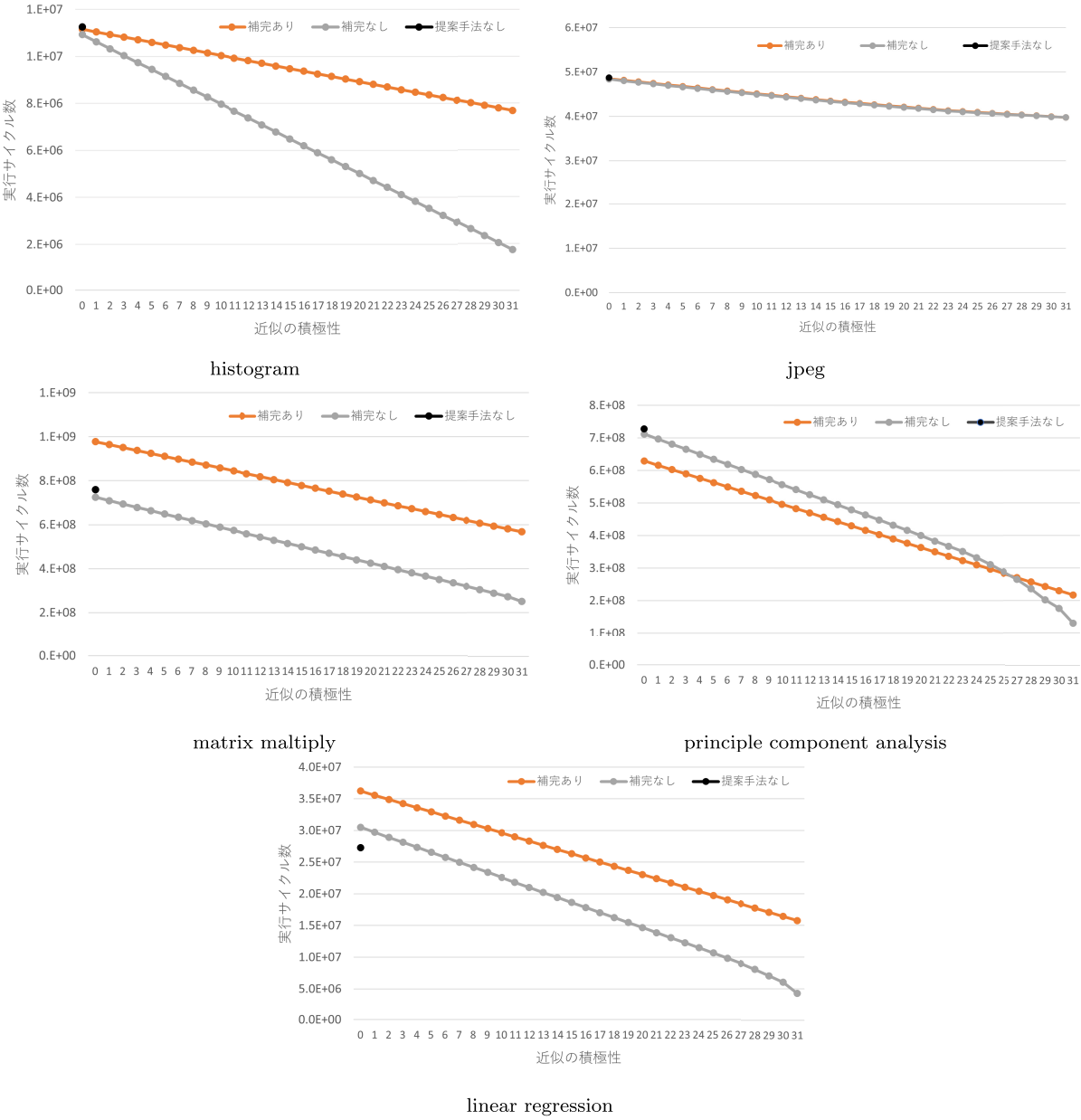

図 10 近似の積極性ごとの実行サイクル数の変化

Fig. 10 Change in the number of execution cycles for each aggressiveness of approximation.

において補完ありの場合は提案手法なしの場合と比べて大きな差が生じる．これは，図 9 に示すように近似ルーチンで使用するために変数を追加で用意していることが原因である．追加した変数への代入処理が多量のスピルを引き起こしており，実行サイクル数を増加させている．

5. 関連手法

Loop perforation [8] や Approximate Loop Unrolling [9] といったループに着目したコンパイル時に計算精度を調整する手法が提案されている．これらの手法は，プログラマが定めた出力品質となるように，ループのイテレーション回数の削減やアンロールを行う．ループだけでなく汎用的なルーチンを対象とした手法として，ACCEPT [4] が提案されている．ACCEPT は，ユーザの設定した計算精度での近似を自動でコンパイルするフレームワークである．ACCEPT を使う際，プログラマは ACCEPT の記法に従って，近似するルーチン，計算精度，出力のメトリクスを選択する．ACCEPT は，コンパイル時に各近似手法と計算精度を組み合わせて検査し，出力のメトリクスを満たすバイナリを生成する．これらの手法はコンパイル時に単一の計算精度を持つバイナリを出力するため，実行時に計算精度を変更することはできないが，出力品質を満たすコーディングという実装コストを大きく軽減する．

anytime automaton [3] は，繰り返すごとに精度が上昇する収束計算，焼きなましといったアプリケーションを対象とした近似計算フレームワークである．この手法は，実行

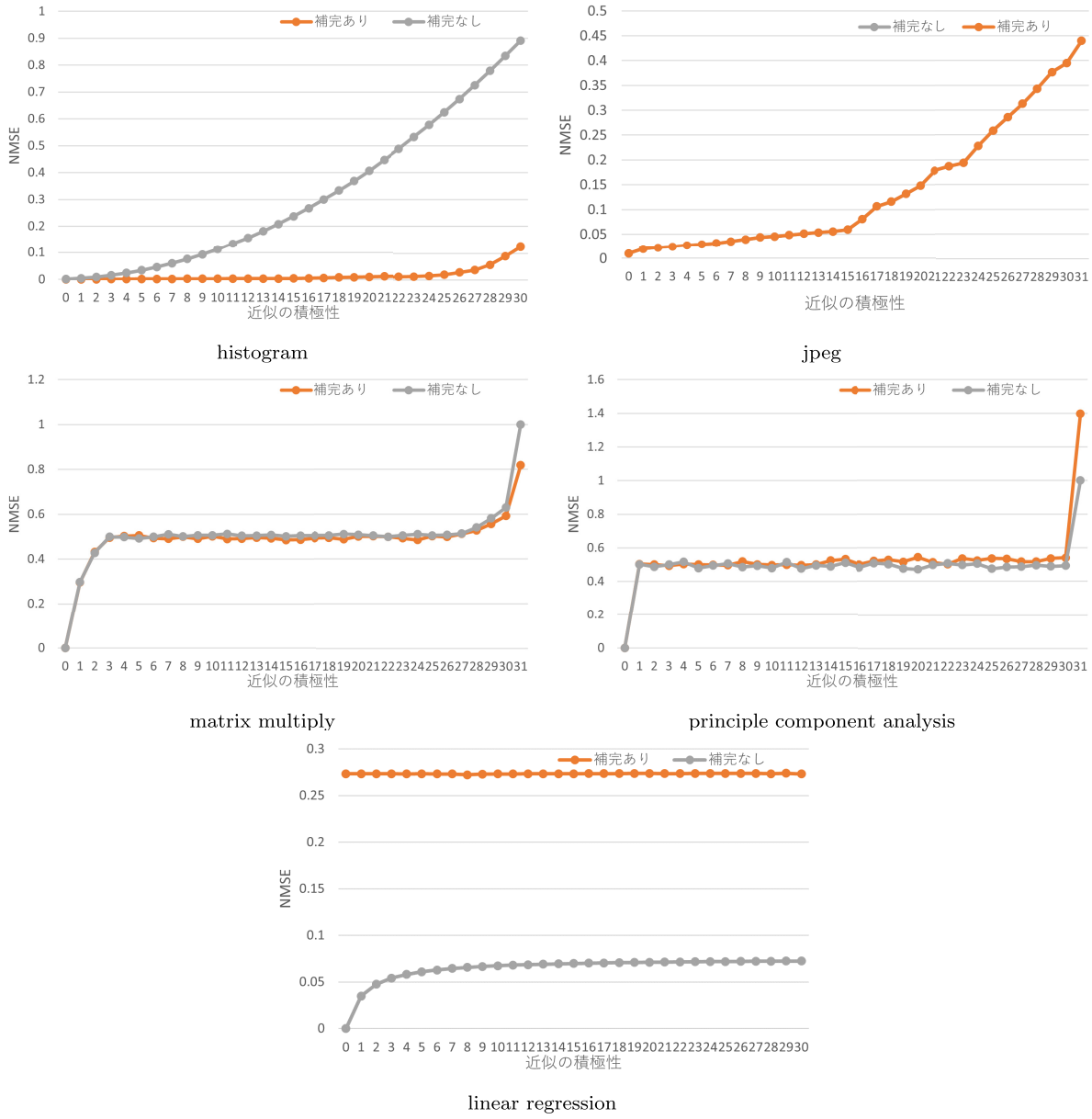

図 11 近似の積極性ごとの NMSE の変化

Fig. 11 Change in NMSE for each aggressiveness of approximation.

をユーザの意思で途中でやめることで精度と実行時間を可変とする．適用できるアプリケーションの種類は制限されているものの，この手法を用いることでアプリケーションの出力品質を保証することができる．

6. 結論

Approximate Computing（AC）の適用は，計算を近似することでその精度と引き換えに消費電力削減と実行速度の向上を同時に達成する．その一方で，AC 技術による計算精度の低下が，プログラムを実行するユーザの許容範囲を超え，実行結果を損なう場合がある．これを解決するため，本論文では，ユーザに応じた近似積極性を動的制御可能なアーキテクチャ，および，Loop Body Switching（LBS）と呼ばれる近似手法に着目した．当該アーキテクチャ上で LBS を利用するために従来ユーザが行っていた機械語命令の記述作業を簡素化すべく，機械語プログラムを生成可能なコンパイラを提案した．提案したコンパイラは，必要な命令の生成と挿入位置の決定，ループ構造の最適化を行う．命令の挿入位置は pragma によってプログラマからコンパイラに伝えることができる．5 つのベンチマークプログラムを用いて，提案コンパイラが当該アーキテクチャで実行可能なバイナリを生成できることを確認した．加えて，そのバイナリをプロセッサシミュレータ上で実行することで，単一のバイナリのみの実行における近似積極性が制御できることを確認した．今後の課題として，if pragma での実行例で用いた 1 つ前のイテレーション結果の利用だけでなく，プログラムとデータの特性ごとに適した近似ルーチンの適用があげられる．

謝辞 本論文の研究は一部，JST さきがけ JPMJPR20M1，共同研究（株式会社富士通研究所）「Approximate Computing 自動適用技術の研究」，JSPS 科研費 JP21J11687 による．

参考文献

[1] Xu, Q., Mytkowicz, T. and Kim, N.S.: Approximate computing: A survey, *IEEE Design & Test*, Vol.33, No.1, pp.8–22 (2015).
[2] Mittal, S.: A survey of techniques for approximate computing, *ACM Computing Surveys* (*CSUR*), Vol.48, No.4, pp.1–33 (2016).
[3] Miguel, J.S. and Jerger, N.E.: The anytime automaton, *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture* (*ISCA*), pp.545–557, IEEE (2016).
[4] Sampson, A., Baixo, A., Ransford, B., Moreau, T., Yip, J., Ceze, L. and Oskin, M.: Accept: A programmer-guided compiler framework for practical approximate computing, *University of Washington Technical Report UW-CSE-15-01*, Vol.1, No.2 (2015).
[5] Baek, W. and Chilimbi, T.M.: Green: A framework for supporting energy-conscious programming using controlled approximation, *Proc. 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.198–209 (2010).
[6] 道上和馬，中村朋生，小泉 透，入江英嗣，坂井修一：近似レベルを動的制御可能なアーキテクチャの提案，*IPSJ SIG Technical Report* (2020).
[7] 塩谷亮太，五島正裕，坂井修一：プロセッサ・シミュレータ「鬼斬弐」の設計と実装．先進的計算基盤システムシンポジウム SACSIS2009，Vol.2009, No.4, pp.120–121 (2009).
[8] Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H. and Rinard, M.: Managing performance vs. accuracy trade-offs with loop perforation, *Proc. ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*, pp.124–134 (2011).
[9] Rodriguez-Cancio, M., Combemale, B. and Baudry, B.: Approximate loop unrolling, *Proc. ACM International Conference on Computing Frontiers*, pp.94–105 (2019).

推薦文

本論文は，今後のポストムーア時代に必要とされる Approximate Computing（AC）に関する独創的な発想に基づくコンパイラフレームワークを提案するものであり，評価を通してその有効性を定量的に示している．本提案手法は，AC で特に重要とされる演算性能と演算精度のトレードオフを動的に制御でき，様々なアプリケーションへの適用が可能である．「計算の質と性能のバランシング」は今後のコンピュータ・アーキテクチャ研究において解決すべき重要課題の 1 つであり，本論文はその直接的な解を示す価値あるものである．以上より，本論文を推薦する．

（FIT2020 第 19 回情報科学技術フォーラム
プログラム委員長 長 健太）

冨田 和孝 （学生会員）

1997 年生．2020 年東京大学工学部電気電子工学科卒業．2020 年同大学大学院情報理工学系研究科修士課程在学中．

中村 朋生 （学生会員）

1995 年生．2018 年東京大学工学部電気電子工学科卒業．2020 年同大学大学院情報理工学系研究科修士課程修了．同年より同博士課程在学中．プロセッサアーキテクチャに関する研究に従事．

小泉 透 （学生会員）

1993 年生．2016 年東京大学理学部化学科卒業．2020 年同大学大学院情報理工学系研究科修士課程修了．同年より同博士課程在学中．プロセッサアーキテクチャに関する研究に従事．IEEE 学生会員．

出川 祐也

1996 年生．2019 年東京大学工学部電子情報工学科卒業．2021 年同大学大学院情報理工学系研究科修士課程修了．同年より同博士課程在学中．プロセッサアーキテクチャに関する研究に従事．IEEE 学生会員．

入江 英嗣 （正会員）

1999 年東京大学工学部電子情報工学科卒業．2004 年同大学大学院情報理工学系研究科博士課程修了．博士（情報理工学）．JST 研究員，東京大学理学部助教，電気通信大学准教授を経て 2015 年より東京大学大学院情報理工学系研究科准教授．コンピュータシステム，特に計算機アーキテクチャ，ヒューマン・コンピュータ・インタラクション，セキュアプロセッサ等に興味を持つ．電子情報通信学会シニア会員，IEEE，ACM 各会員．

坂井 修一 （正会員）

1958 年生．1981 年東京大学理学部卒業．1986 年同大学大学院工学系研究科修了．工学博士．電子技術総合研究所，MIT，RWC，筑波大学を経て，1998 年東京大学助教授．2001 年より同大学大学院情報理工学系研究科教授．情報システムおよびその応用の研究に従事．現在，東京大学副学長・附属図書館長．情報処理学会研究賞（1989），情報処理学会論文賞（1991），IBM 科学賞（1991），市村学術賞（1995），IEEE Outstanding Paper Award（1995），Sun Distinguished Speaker Award（1997），大川出版賞（2012），電子情報通信学会業績賞（2018）等受賞．電子情報通信学会フェロー．人工知能学会，ACM，IEEE 各会員．本会フェロー．