Vol.2015-ARC-216 No.18
2015/8/5

自動メモ化プロセッサにおける近似的入力一致比較手法

佐藤 裕貴[1]　津村 高範[1]　津邑 公暁[1]　中島 康彦[2]

概要：計算の近似化によって実行時間や消費電力を削減する Approximate Computing が，ハードウェアからソフトウェアに至るまで様々な分野で盛んに研究されている．一方，我々は計算再利用技術に基づく自動メモ化プロセッサを提案している．自動メモ化プロセッサは，計算再利用の対象となる関数の入出力を再利用表に記憶し，それらを再利用することで，同一入力による同一命令区間の実行を省略し，高速化を図る手法である．本稿では，この自動メモ化プロセッサに Approximate Computing の考え方を適用し，併せて，Approximate Computing の対象とする関数や入力を指示するためのプログラミングフレームワークを設計することで，自動メモ化プロセッサをベースとした Approximate Computing 基盤を提案する．提案手法の有効性を検証するため，MediaBench の cjpeg を用いてシミュレーションによる評価を行った．その結果，既存の自動メモ化プロセッサと比較して，提案手法ではわずかな画質の低下で最大 22.3%の命令実行サイクル数を削減し，また，最大 29.5%の再利用率の向上を達成し，その有効性を確認した．

1. はじめに

計算の近似化によって実行時間や消費電力を削減する **Approximate Computing** が，プログラミング言語 [1] からトランジスタレベル [2] に至るまで様々な分野で盛んに研究されている．Approximate Computing とは，出力が知覚的に許容できる範囲であれば，完全に正確な計算ではなく近似的な計算を行うことを許容する，という考え方であり，画像処理や信号処理，機械学習などの分野での活用が期待されている．

一方，我々は計算再利用技術に基づいた**自動メモ化プロセッサ（Auto-Memoization Processor）**[3] を提案している．自動メモ化プロセッサは，関数を計算再利用可能な命令区間とみなし，実行時にその入出力を再利用表に記憶する．そして，それらの入出力を再利用することで，同一関数を同一入力を用いて再び実行しようとした際に，この関数の実行自体を省略する．本稿では，これまで我々が提案してきた自動メモ化プロセッサに Approximate Computing の考え方を適用し，併せて，Approximate Computing の対象とする関数や入力を指示するためのプログラミングフレームワークを設計することで，自動メモ化プロセッサをベースとした Approximate Computing 基盤を提案する．

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

2. 関連研究

Approximate Computing に関する研究として，例えば Raha ら [4] は，アプリケーションの実行中に近似度を動的に調節し，いくつかの演算処理をスキップすることにより，消費電力を削減する手法を提案している．また，Approximate Computing の考え方をハードウェアに対して適用する研究も行われている．例えば，Sasa ら [5] が提案するシステムでは，まずプログラマがコード内で，重要度が低く近似的な計算を行っても良い部分を指定する．そして，その情報を利用し，信頼度の低いハードウェアに重要度の低い命令を自動的に割り当て，その命令を実行させる．他にも，Shoushtari ら [6] は，Approximate Computing の考え方の適用により，SRAM 内の保護帯域の設定を緩和し，消費電力を削減している．

また，計算再利用に対して Approximate Computing の考え方を適用する研究も行われている．例えば，Álvarez ら [7] は命令単位で再利用を行う際に入力一致比較を近似化する手法を提案している．この手法ではまず，浮動小数点演算処理装置内に用意したルックアップテーブルに，浮動小数点演算命令のオペランド，オペレータおよび演算結果を登録する．その後，浮動小数点演算命令が検出されると，オペランドの仮数部の下位数ビットをマスクしてルックアップテーブル内のオペランドと一致比較を行う．比較の結果，一致するエントリが存在する場合，そのエントリに格納されている演算結果を使用し，近似的な浮動小数点演算を実現している．

© 2015 Information Processing Society of Japan

これらの既存研究に対し，本稿では関数単位で再利用を行う際に関数の入力を部分的にマスクし，入力一致比較を近似化する手法を提案する．なお，比較を行う対象を部分的にマスクする方法は，命令単位で再利用を行う際に一致比較を近似化する手法でも採用されている．一方，本提案手法では関数を計算再利用対象とし，その入力を部分的にマスクすることで，関数単位の計算再利用率の向上を図る．対象を関数単位とすることにより，命令単位の計算再利用を適用した場合より大きな効果が期待できるとともに，さまざまなプログラムに汎用的な枠組みで Approximate Computing を適用することが可能となる．本稿では，この Approximate Computing 適用のためのインタフェースについても提案する．

3. 自動メモ化プロセッサ

本章では，本稿で取り扱う自動メモ化プロセッサについて，その高速化の方針，アーキテクチャの構成，動作を概説する．

3.1 再利用機構の構成

計算再利用（Computation Reuse）とは，プログラム内の関数などの命令区間実行時に，その入力の組（入力セット）と出力の組（出力セット）を記憶しておき，再び同じ入力によりその命令区間が実行されようとした場合に，記憶された過去の出力を利用することで命令区間の実行自体を省略し，高速化を図る手法である．また，この手法を命令区間に適用することを**メモ化（Memoization）**[8] と呼ぶ．メモ化は元来，高速化のためのプログラミングテクニックである．しかし，メモ化を適用するためには，プログラムを記述し直す必要があり，既存のロードモジュールやバイナリをそのまま高速化することはできない．その上，ソフトウェアによるメモ化 [9] はオーバヘッドが大きく，限られたプログラムでしか性能向上が得られない傾向がある．

そこで，我々はハードウェアを用いて動的にメモ化を適用することで，既存のバイナリを変更すること無く高速実行可能な自動メモ化プロセッサ（Auto-Memoization Processor）を提案している．自動メモ化プロセッサの構成の概略を**図 1** に示す．自動メモ化プロセッサは，コアの内部に一般的な CPU コアが持つ ALU，レジスタ，1 次データキャッシュ（D$1）等を持ち，コアの外部に 2 次データキャッシュ（D$2）を持つ．また，自動メモ化プロセッサ独自の機構として，メモ化制御機構，再利用表（MemoTbl），および MemoTbl への書き込みバッファとして働く再利用バッファ（MemoBuf）を持つ．MemoTbl はサイズが大きく，コアからのアクセスコストが大きいため，入出力を検出する度に MemoTbl へアクセスするとオーバヘッドが大きくなる．そこで，このオーバヘッドを軽減するために，作業用のバッファである MemoBuf をコアの内部に設けている．

自動メモ化プロセッサは再利用対象命令区間に進入する

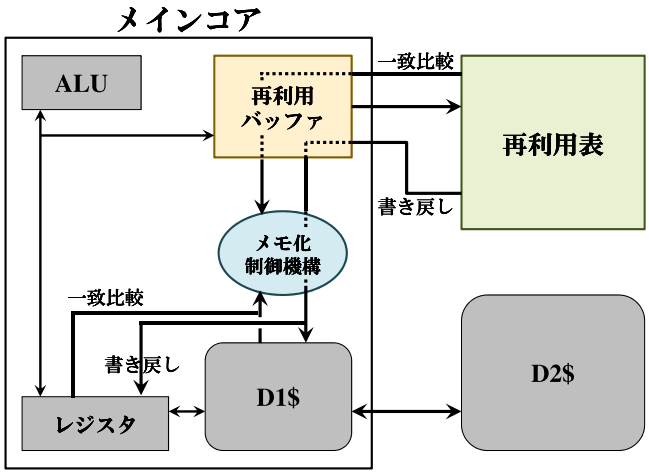

図 1 自動メモ化プロセッサの構成

```
1  int a = 3, b = 4, c = 8;
2  int calc(x){
3    int tmp = x + 1;
4    tmp += a;
5    if(tmp < 7) tmp += b;
6    else        tmp += c;
7    return(tmp);
8  }
9  int main(void){
10   calc(2);          /* x = 2, a = 3, b = 4 */
11   b = 5; calc(2); /* x = 2, a = 3, b = 5 */
12   a = 4; calc(2); /* x = 2, a = 5, c = 8 */
13   a = 3; calc(2); /* x = 2, a = 3, b = 5 */
14   return(0);
15 }
```

図 2 サンプルプログラム

際，MemoTbl を参照し現在の入力セットと MemoTbl に記憶されている過去の入力セットを比較する．もし，現在の入力セットが MemoTbl 上のいずれかの入力セットと一致する場合，メモ化制御機構は入力セットに対応する出力セットをレジスタやキャッシュに書き戻し，命令区間の実行を省略する．一方，現在の入力セットが MemoTbl 上のいずれの入力セットとも一致しない場合，自動メモ化プロセッサは当該命令区間を通常実行しながら，入出力を MemoBuf に格納し，実行終了時に MemoBuf の内容を MemoTbl に登録することで将来の再利用に備える．

なお，MemoBuf は複数のエントリを持ち，1 エントリが 1 入出力セットに対応する．各エントリは，どの命令区間に対応しているかを示すインデクス（FLTbl idx），その命令区間の実行開始時のスタックポインタ（SP），関数の戻りアドレス（retOfs），命令区間の入力セット（Read）および出力セット（Write）のフィールドを持つ．また，Read フィールドおよび Write フィールドは，アドレスもしくはレジスタ番号を記憶する addr/reg フィールドと値を記憶する value フィールドで構成される．

さて，一般に命令区間内では，複数の入力が順に参照され使用される．しかし，同じ命令区間でも，その入力アドレス

© 2015 Information Processing Society of Japan

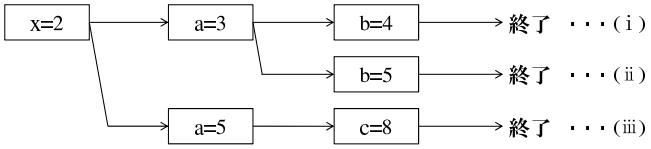

図 3　入力パターンの木構造

図 4　MemoTbl の構成

の列は実行毎に異なる場合がある．例えば，条件分岐命令の実行直後に参照される入力アドレスはその条件分岐命令の分岐結果によって変化する．そしてその分岐命令の結果は，入力の値によって変化する．このように，ある命令区間の入力アドレスの列はその入力値によって分岐していくため，自動メモ化プロセッサは，全入力パターンを木構造で表現し，MemoTbl に格納する．例えば，自動メモ化プロセッサが**図 2** に示すサンプルプログラムを実行する場合の，関数 calc における全入力パターンを表現した木構造を**図 3** に示す．なお，図 3 のノードは命令区間の入力を，エッジは入力と次に参照される入力との対応関係を示している．また，入力セット (i)，(ii)，(iii) はサンプルプログラムの 10 行目，11 行目，12 行目における関数呼び出しにそれぞれ対応する．この例において，プログラムの 10 行目および 11 行目の関数呼び出しにより命令区間を実行する際，それぞれ図 3 の入力セット（i），入力セット（ii）に示す順で入力が参照される．この場合，どちらも 3 番目に変数 b が参照される．これに対し，12 行目の関数呼び出しにより命令区間を実行する際，入力セット (iii) に示す順で入力が参照され，3 番目に変数 c が参照される．これは，2 番目に参照される変数 a の値が異なることにより，プログラムの 5 行目における条件分岐の結果が変化し，次入力アドレスが変化したためである．このように，自動メモ化プロセッサでは入力の参照順の変化に対応するために，入力パターンを木構造で管理している．

次に**図 4** に MemoTbl の具体的な構成を示す．この MemoTbl は以下の 4 つの表から構成される．

FLTbl: 各命令区間の開始アドレスを記憶する表
InTbl: 命令区間の入力を記憶する表
AddrTbl: 命令区間の入力アドレスを記憶する表
OutTbl: 命令区間の出力を記憶する表

FLTbl，AddrTbl，OutTbl は RAM で実装し，InTbl は高速な連想検索が可能な汎用 3 値 CAM（Content Addressable Memory）で実装する．以下では，MemoTbl の各表の構成について説明する．

FLTbl は各行に再利用対象となる各命令区間を記憶する．命令区間の判別には，命令区間の開始アドレス（addr）を用いる．

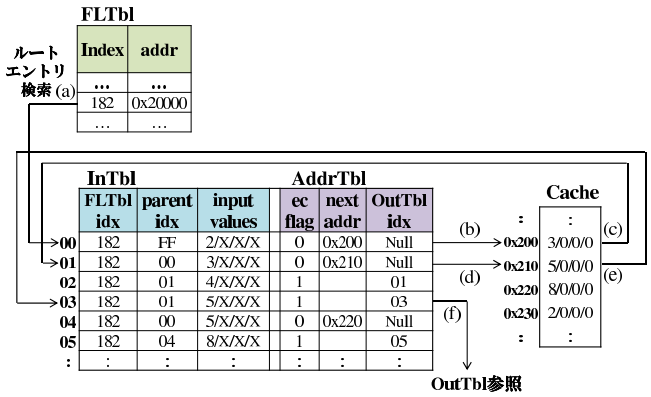

図 5　MemoTbl の検索手順

InTbl の各行は FLTbl の行番号 Index に対応する FLTbl idx を持ち，どの命令区間の入力を記憶しているかを区別する．また，InTbl は入力を記憶するための input values を持ち，記憶すべき入力が含まれるキャッシュライン全体を 1 エントリとして登録する．その際，当該キャッシュライン中の，対象入力以外の情報を含む部分については 3 値 CAM のドントケア値を用いて表し，検索の際に一致比較の対象とならないようにする．また，InTbl では命令区間の全入力パターンを木構造で管理するため，親エントリのインデクス（parent idx）を持つ．

AddrTbl の各行は入力検索のために，次に参照するべきアドレス（next addr）を持つ．この AddrTbl は InTbl と同数のエントリを持ち，各エントリは 1 対 1 に対応する．また，AddrTbl はそのエントリが入力セットの終端であるか否かを示すフラグ（ec flag）に加えて，入力検索に成功した際に，出力を記憶している表である OutTbl のエントリを参照するためのインデクス（OutTbl idx）を持つ．

OutTbl の各エントリは命令区間の出力先のアドレス（output addr）と出力値（output values）を持つ．また，OutTbl は 1 つの出力セットを複数エントリを用いてリスト構造により管理するため，次に参照すべきエントリのインデクス（next idx）を持つ．

3.2　再利用機構の動作

本節では，自動メモ化プロセッサが命令区間に計算再利用を適用する際の動作について説明する．図 3 に示した入力セットが InTbl と AddrTbl に登録されている様子を**図 5** に示す．図 5 中の (a)〜(f) は，図 2 の 13 行目における関数呼び出し時の MemoTbl 検索フローを示している．また，図中の InTbl における X はドントケアを表しており，3.1 節で述べたように，キャッシュライン中の入力とは無関係な部分をドントケアで表している．なお，図中の InTbl の parent idx における FF は親エントリが存在しないことを表している．

再利用対象区間の実行開始アドレスが検出されると，input values が現在のレジスタ上の入力と一致し，かつ parent idx が FF であるようなルートエントリが検索される．検索の

© 2015 Information Processing Society of Japan

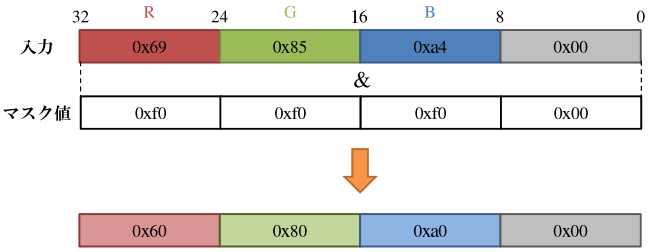

図 6　入力のマスク例

結果，インデクス番号が 00 であるエントリがルートエントリであると判明する (a)．いま，そのエントリに対応する AddrTbl の next addr が 0x200 番地を指しており，そのアドレスに対応するキャッシュラインに，次に参照する input values の値が格納されていることがわかるため，そのキャッシュラインを参照する (b)．次に，キャッシュから得られた値である 3 を input values として持ち，parent idx が 00 であるエントリを InTbl から検索する (c)．以降，同様に検索を続け (d)(e)，インデクス番号が 03 であるエントリにおいて，AddrTbl の ec flag がセットされているため，入力セットの終端に達したことがわかる，つまり入力の検索に成功したことがわかるので，この時，検索の結果到達した AddrTbl エントリの OutTbl idx に格納されているインデクス番号が指す OutTbl エントリを参照する (f)．そして，エントリから読み出した出力値をレジスタやキャッシュに書き戻すことで命令区間の実行を省略することができる．

4. 自動メモ化プロセッサをベースとする Approximate Computing 基盤

本章では，関数を単位として Approximate Computing を適用可能とする，自動メモ化プロセッサをベースとした計算基盤について述べる．提案するシステムは，近似的な入力一致比較を可能とする自動メモ化プロセッサと，Approximate Computing の適用対象である関数および関数の入力をプログラマが指示可能なプログラミングフレームワークから成る．

4.1 提案する計算基盤の全体構成

本稿では，自動メモ化プロセッサをベースとした Approximate Computing 基盤を提案する．このシステムでは，自動メモ化プロセッサの入力一致比較を近似化することで，従来の自動メモ化プロセッサの再利用率および命令実行サイクル削減率を向上させるとともに，プログラマは関数単位で容易に Approximate Computing を適用可能となる．

自動メモ化プロセッサにおける近似的入力一致比較の具体的な方法として，関数の入力を部分的にマスクして一致比較を行うという方法を採用する．なお，計算再利用において一致比較対象となる入力の一部をマスクする方法は，2 章で述べたように命令単位の計算再利用に対しては既に提案されて

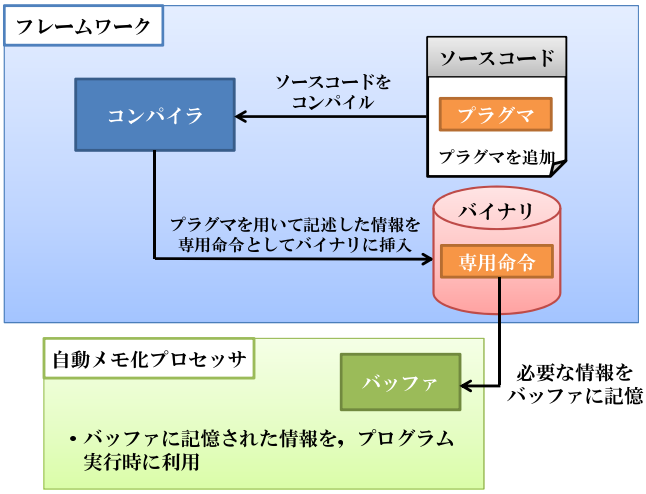

図 7　自動メモ化プロセッサをベースとした Approximate Computing 基盤の全体構成

いる．これに対し本提案手法では，関数を計算再利用の単位とし，その入力一致比較において部分的不一致を許容する．これにより関数単位での処理の削減に寄与するとともに，さまざまなプログラムに対して汎用的な枠組みで Approximate Computing を適用可能となる．ここで，入力を部分的にマスクする方法について説明する．例えば画像処理では，一般にピクセル値が処理関数の入力とされることが多く，そのピクセル値は 32 ビット表現の場合，R，G，B の各色がそれぞれ 8 ビットずつで表現される．そのようなピクセル値入力を近似化する際，例えばピクセル値における RGB のそれぞれ上位 4 ビットのみで一致比較を行う場合，**図 6** に示すように，0xf0f0f000 という値でピクセル値をマスクする．また，ピクセル値における RGB のそれぞれ上位 2 ビットのみで一致比較を行う場合，0xc0c0c000 という値でピクセル値をマスクする．提案手法では入力をマスクし，自動メモ化プロセッサの入力の一致比較を近似化するために，まずプログラマが指定した入力とマスク値の情報を MemoTbl に登録する．そして，この情報を利用することで，一致比較の際に現在の入力と過去の入力のビットが全て一致していなくとも当該命令区間に計算再利用の適用を許可する．このような入力の近似化は，画像処理以外にも，例えば音声や動画の圧縮処理を行うマルチメディアアプリケーションにおいて，入力として与えられるデータを量子化により変換する際の処理などに適用可能である．他にも，入力に対する出力を推定する機械学習において，類似する入力データをグループ化するクラスタリング処理を行う際などにも有効であると考えられる．なお，我々は既に，特定のアプリケーションにおいて入力一致比較を近似化することで自動メモ化プロセッサの性能が向上することを確認しているが [10]，本稿ではこれらさまざまなプログラムに適用範囲を広げるため，入力一致比較の近似化を適用可能とするための汎用的なプログラミングフレームワークを提案する．

© 2015 Information Processing Society of Japan

```
1 #pragma approx(img, 0xf0f0f000)
2 int col2gray(int img, float y){
3   int R, G, B;
4   R = img>>16;
5   G = (img&0x0000ff00)>>8;
6   B = img&0x000000ff;
7   return pow(R, y)+pow(G, y)+pow(B, y);
8 }
```

図 8 プラグマを追加したソースコードの例

```
1                :
2   20000 call   30000 <col2gray>
3                :
4                :
5 00030000 <col2gray>:
6   30000 approx   r0, 0xf0f0f000
7   30004 sub      sp, sp, #16
8   30008 str      r1, [sp, #8]
9   3000c str      r2, [sp, #12]
10               :
11  30024 ret
```

図 9 専用命令を挿入したアセンブリコードの例

ここで，提案する計算基盤の全体構成を**図 7** に示す．まず，プログラマは関数の計算を近似化するために，近似化の対象となる関数入力，および入力に適用するマスク値をプラグマを用いてソースコードに記述する．次に，このソースコードをコンパイルする際，コンパイラはプラグマを用いて記述された情報を，近似的入力一致比較のための専用命令としてバイナリに挿入する．自動メモ化プロセッサは，この，コンパイラが生成した専用命令を含むバイナリを実行する際に，専用命令で指定された情報をバッファに記憶する．そして，バッファに記憶した情報を利用することで，当該関数に対して近似的入力一致比較を用いた計算再利用を適用する．4 章の以下の節では，近似化対象となる関数入力をプログラム中で指定する方法，およびコンパイラの拡張について述べ，当該コンパイラによって生成されたバイナリを実行する際の自動メモ化プロセッサの動作について説明する．

4.2 関数選択・近似度設定のためのプログラム記述

本稿で提案する計算基盤では，Approximate Computing を適用する対象となる関数，および，その近似化対象となる関数の入力を，プログラマがプラグマを用いて指定する方法を採用する．記述方式にプラグマを採用した理由は，拡張性の高さと，構文解析が容易であるという点からである．提案するプラグマの書式は以下の通りである．

#pragma approx (app_input, app_mask)

approx は，近似的入力一致比較を行うことをコンパイラに示すための記述である．また，近似化対象の入力変数名（app_input）とマスク値（app_mask）を，続く括弧内で指定する．このプラグマをソースコード中の，Approximate Computing の適用対象となる関数を定義している箇所の直前に記述する．ここで**図 8** に，近似的入力一致比較のためのプラグマを追加したソースコードの例を示す．図 8 は，関数 col2gray の入力変数 img に対して近似的入力一致比較を適用する場合の，ソースコードへのプラグマの記述例を表している．図 8 では，2 行目から関数 col2gray が定義されている．そのため，この関数の直前である 1 行目にプラグマを記述している．この枠組みにより，Approximate Computing の対象とする関数や，近似化の対象となる関数入力を，プログラマが柔軟に設定することができる．

4.3 コンパイラの拡張

提案手法により，近似化の対象となる関数入力に対して近似的一致比較を適用するために，プラグマを用いて記述された近似化の対象となる関数入力，およびマスク値の情報を自動メモ化プロセッサは利用する必要がある．そこで，自動メモ化プロセッサの命令セットに専用命令を追加する．コンパイラは，プラグマをこの専用命令へと変換し，プラグマで指定されたパラメータを，当該専用命令のオペランドとしてエンコードする．なお，コンパイラは専用命令を，Approximate Computing の対象とする関数内部の処理の先頭に挿入する．この拡張したコンパイラにより生成された専用命令を含むバイナリを自動メモ化プロセッサで実行する際，専用命令が検出されると，当該関数が Approximate Computing の適用対象であることがわかる．ここで，拡張したコンパイラによって図 8 のソースコードをコンパイルした際に生成されるアセンブリコードの例を**図 9** に示す．なお，近似化の対象となる関数入力である img は，レジスタ r0 にマッピングされているとする．図 9 では，Approximate Computing の対象となる関数である col2gray が，5 行目から定義されている．そのため，専用命令 “approx r0, 0xf0f0f000” を，関数 col2gray の先頭である 6 行目に挿入する．このように，近似的入力一致比較のための専用命令をバイナリに挿入するようコンパイラを拡張する．

4.4 提案手法における自動メモ化プロセッサの動作

本節では，提案手法によりプログラムを実行する際，どのように入力を MemoTbl に登録し，どのように MemoTbl から入力を検索するかについてそれぞれ説明する．

4.4.1 入力登録時の動作

提案手法では，自動メモ化プロセッサが専用命令を実行し，近似化の対象となる関数入力およびマスク値を記憶する必要がある．そこで，これらの情報を記憶するために，MemoBuf を拡張する．MemoBuf に，近似化の対象となる関数入力を記憶するための approx. input フィールド，および入力に適用するマスク値を記憶するための approx. mask フィールドを追加する．そして，専用命令の実行により，これらフィールドにオペランドで指定された情報を記憶する．

© 2015 Information Processing Society of Japan

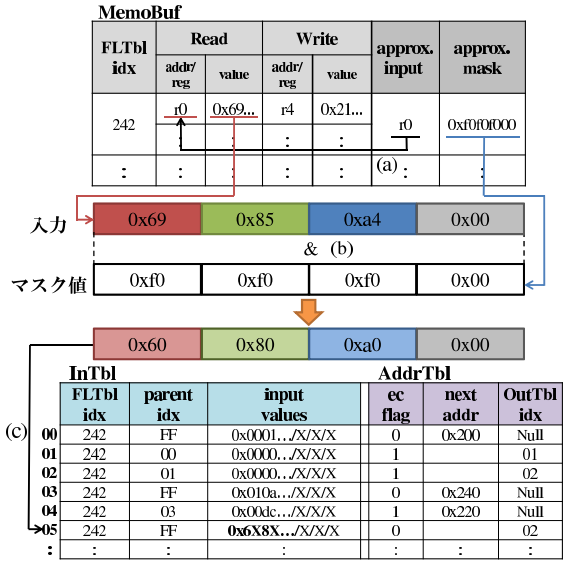

図 10 入力登録時の手順

ここで，提案手法において自動メモ化プロセッサがプログラムの実行を開始してから MemoTbl へ入出力を登録するまでの動作を図 9 と**図 10** を用いて説明する．図 10 の上の表は，提案手法により拡張した MemoBuf を表し，下の表は MemoTbl 内の，InTbl と AddrTbl を表す．なお，図 10 では InTbl の input values を 16 進数で表現している．まず，図 9 に示すプログラムが先頭から実行され，2 行目の関数呼び出し命令が実行されると，現在の入力セットと MemoTbl に登録されている過去の入力セットが比較される．比較の結果，一致する入力エントリが存在しなかった場合，5 行目から順に関数内の命令を通常実行する．関数内の命令を実行する際，4.3 節で述べた 6 行目の専用命令がまず検出され，実行中の関数が Approximate Computing の対象であることがわかる．この専用命令が実行されると，近似化の対象となる関数入力である img を格納しているレジスタ番号 r0 およびその入力に適用するマスク値である 0xf0f0f000 を，MemoBuf の approx. input フィールドおよび approx. mask フィールドにそれぞれ記憶する．専用命令を実行した後，通常の自動メモ化プロセッサと同様に関数内の命令を順に実行しながら，入出力を MemoBuf へ記憶していく．その後，11 行目で return 命令を検出すると，自動メモ化プロセッサは MemoBuf の内容を MemoTbl へ書き込む．この時，MemoBuf の approx. input フィールドを参照し，レジスタ番号 r0 を MemoBuf の addr/reg フィールド上で検索する（図 10(a)）．図 10 では，addr/reg フィールドにレジスタ番号 r0 が存在するため，approx. mask フィールドを参照し，addr/reg フィールドのレジスタ番号 r0 と対応する value フィールドの値に対してマスク値 0xf0f0f000 を適用する（図 10(b)）．なお 3 章で述べたように，従来の自動メモ化プロセッサでは入力セットをキャッシュライン単位で登録しており，キャッシュライン中の入力とは無関係な部分をドントケアで表している．提案手法ではこれと同様に近似化の対象となる入力値の内，検索の際に一致比較対象とならないビットをドントケアとして MemoTbl 内で記憶する（図 10(c)）．

表 1 評価環境

MemoBuf	64 KBytes
MemoTbl CAM	128 KBytes
Comparison (register and CAM)	9 cycles/32 Bytes
Comparison (Cache and CAM)	10 cycles/32 Bytes
Write back (MemoTbl to Reg./Cache)	1 cycle/32 Bytes
D1 cache	32 KBytes
line size	32 Bytes
ways	4 ways
latency	2 cycles
miss penalty	10 cycles
D2 cache	2 MBytes
line size	32 Bytes
ways	4 ways
latency	10 cycles
miss penalty	100 cycles
Register windows	4 sets
miss penalty	20 cycles/set

4.4.2 入力検索時の動作

近似的入力一致比較の対象となる関数入力を，MemoTbl から検索する際，従来の検索手順を変更することなく，近似的入力一致比較を実現できる．これは，4.4.1 項で述べたように，近似化の対象となる関数入力については，比較する必要のないビットをドントケアとして MemoTbl に登録しているためである．関数が呼び出されると，現在のレジスタ上の入力を InTbl に記憶されている input values のエントリから検索する．そして，入力を比較する際．input values のエントリに登録されている入力のビットの内，ドントケアではないビットに対して一致比較を行う．これにより，近似的入力一致比較を実現する．

5. 評価

以上で述べた提案システムを既存の自動メモ化プロセッサシミュレータに対して実装した．また，提案手法の有効性を確認するために，ベンチマークプログラムを用いてサイクルベースシミュレーションにより評価を行った．

5.1 評価環境

評価には，計算再利用のための機構を実装した単命令発行の SPARC V8 シミュレータを用いた．評価に用いたパラメータを**表 1** に示す．なお，キャッシュや命令レイテンシは SPARC64-III[11] を参考とした．MemoTbl 内の InTbl に用いる CAM の構成は MOSAID 社の DC18288[12] を参考にし，サイズは 32Bytes 幅 ×4K 行の 128KBytes とした．な

© 2015 Information Processing Society of Japan

```
1                    :
2  void forward_Q(JSAMPLE sample...){
3                    :
4    for (i = 0; i < DCTSIZE; i++) {
5      qval = div[i];
6      temp = work[i];
7      if (temp < 0) {
8        temp = -temp;
9        temp += qval>>1;
10       DIVIDE_BY(temp, qval);
11       temp = -temp;
12     } else {
13       temp += qval>>1;
14       DIVIDE_BY(temp, qval);
15     }
16     out[i] = (JCOEF) temp;
17   }
18                   :
19 }
20                   :
```

図 11　書き換え前のプログラム

お，プロセッサのクロック周波数は 128KBytes の CAM のクロック周波数の 10 倍と仮定して検索オーバヘッドを見積もっている．

5.2　評価結果

提案手法の有効性を確かめるため，メディア系ベンチマークである MediaBench から，画像圧縮を行うプログラムである cjpeg を用いて実行サイクル数および再利用率を評価した．入力には 256 × 256 ピクセルの画像を使用した．cjpeg の画像圧縮は，RGB/YUV 変換，DCT（離数コサイン変換），量子化，ハフマン符号化の各処理から成る．本稿ではこれらの処理の内，量子化を行う処理に対して提案手法を適用し，評価を行った．提案手法を用いるためにプログラマは，Approximate Computing が適用可能な部分である，量子化計算の部分を関数として切り出して定義し，その関数に対してプラグマを指定する．今回の評価にあたり行った，具体的なプログラムの書き換えを**図 11** および**図 12** に示す．図 11 は書き換え前，図 12 は書き換え後のプログラムを表している．図 11 の 2 行目から定義されている forward_Q が，量子化処理のための関数であり，その中で量子化計算は 7 行目～16 行目の部分で定義されている．この量子化計算の部分を別関数 q_loop として切り出し，forward_Q 内で q_loop を呼び出すよう書き換えたプログラムが図 12 である．この書き換え後のプログラムにおいて，DCT 係数値を持つ入力である temp に対してマスクを適用するため，10 行目にプラグマを追記している．以上のようにプログラムを書き換え，提案手法の評価を行った．この結果を**図 13** に示す． 図 13 では，左から順に

(N)　プログラム書換前・メモ化なし（ベースライン）
(N')　プログラム書換後・メモ化なし
(M)　プログラム書換前・メモ化あり
(M'0) プログラム書換後・メモ化あり

```
1                    :
2  void forward_Q(JSAMPLE sample...){
3                    :
4    for (i = 0; i < DCTSIZE; i++) {
5      out[i] = q_loop(work[i], div[i]);
6    }
7                    :
8  }
9                    :
10 #pragma approx(temp, 0xfffffff0)
11 JCOEF q_loop(temp, qval){
12   if (temp < 0) {
13     temp = -temp;
14     temp += qval>>1;
15     DIVIDE_BY(temp, qval);
16     temp = -temp;
17   } else {
18     temp += qval>>1;
19     DIVIDE_BY(temp, qval);
20   }
21   return (JCOEF) temp;
22 }
23                   :
```

図 12　書き換え後のプログラム

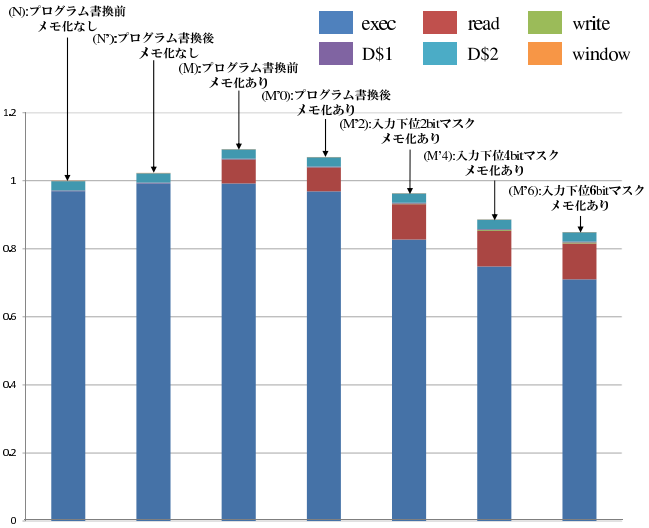

図 13　評価結果:命令実行サイクル数

(M'2) 入力下位 2 ビットマスク・メモ化あり
(M'4) 入力下位 4 ビットマスク・メモ化あり
(M'6) 入力下位 6 ビットマスク・メモ化あり

が要した総実行サイクル数を表しており，プログラム書換前・メモ化なし (N) の場合を 1 として正規化している．なお，上述したプログラムの書き換えによって，書き換え前と比較して関数 q_loop の呼び出しオーバヘッドによる速度低下が発生すると考えられる．プログラムを書き換えた上で入力をマスクしない場合についても計測しているのは，この影響を確認するためである．

グラフの凡例はサイクル数の内訳を示しており，exec は命令実行サイクル数，read は MemoTbl との比較に要したサイクル数（検索オーバヘッド），write は MemoTbl から出力をレジスタやメモリに書き戻す際に要したサイクル数（書き戻

© 2015 Information Processing Society of Japan

表 2　評価結果:再利用率

	(M)	(M'0)	(M'2)	(M'4)	(M'6)
再利用率	0.019%	0.029%	16.8%	24.7%	29.5%

図 14　出力結果

しオーバヘッド)，D$1 および D$2 は 1 次および 2 次データキャッシュミスペナルティ，window はレジスタウインドウミスペナルティである．また，メモ化を行ったそれぞれの場合における，関数全体の再利用率を**表 2** に示す．

(M'2)，(M'4)，(M'6) の結果より，マスクする入力のビットを増やすに連れ，read および write がわずかに増加しているが，exec は大きく減少しており，全体の性能は向上していることがわかる．また，表 2 より，マスクする入力のビットを増やすに連れ，再利用率が向上していることがわかる．これらの結果から，入力を部分的にマスクすることで入力の一致比較を近似化し，関数単位の計算再利用率の向上および命令実行サイクル数を削減できていることがわかり，提案手法による期待通りの効果が得られていることを確認できた．

次に，入力をマスクしなかった場合の出力結果と入力をマスクした場合の出力結果を，**図 14** に示す．図 14 より，(M'2)，(M'4) では，入力をマスクしなかった場合の出力結果と比較してほとんど画質の低下は見られないことがわかる．また，(M'6) では，入力をマスクしなかった場合の出力結果と比較してわずかな画質の低下が見られるが，アプリケーションによっては十分許容範囲内の画質低下に収まっていると考えられる．以上の結果から，命令実行サイクル数の削減および再利用率の向上を達成し，また，出力の精度低下が知覚的に許容できる範囲内であることを確認できた．

6. おわりに

本稿では，これまで我々が提案してきた自動メモ化プロセッサに Approximate Computing の考え方を適用し，併せて，Approximate Computing の対象とする関数や入力を指示するためのプログラミングフレームワークを設計することで，自動メモ化プロセッサをベースとした Approximate Computing 基盤を提案した．MediaBench から，画像圧縮を行う cjpeg を用いて評価を行った結果，非常に簡単なプログラムの書き換えにより，既存の自動メモ化プロセッサと比較して，最大 22.3%の命令実行サイクル数削減，および最大 29.5%の再利用率向上を達成し，提案手法の有効性を確認することができた．今後の課題として，許容できる出力の誤差率をプログラマに指定させ，その誤差率の範囲内で最大のパフォーマンスが得られるよう，自動メモ化プロセッサ が動的にマスク値を調整する機構の検討が挙げられる．

参考文献

[1] Hadi, E., Adrian, S., Luis, C. and Doug, B.: Architecture Support for Disciplined Approximate Programming, *Proc. 17th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, pp. 301–312 (2012).

[2] Vaibhav, G., Debabrata, M., Anand, R. and Kaushik, R.: Low-Power Digital Signal Processing Using Approximate Adders, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 32, No. 1, pp. 124–137 (2013).

[3] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp. 245–250 (2007).

[4] Raha, A., Venkataramani, S., Raghunathan, V. and Raghunathan, A.: Quality Configurable Reduce-and-Rank for Energy Efficient Approximate Computing, *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pp. 665–670 (2015).

[5] Sasa, M., Michael, C., Sara, A., Qi, Z. and C., R. M.: Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels, *Proc. ACM Int'l Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*, pp. 309–328 (2014).

[6] Shoushtari, M., BanaiyanMofrad, A. and Dutt, N.: Exploiting Partially-Forgetful Memories for Approximate Computing, *IEEE Embedded Systems Letters*, Vol. 7, No. 1, pp. 19–22 (2015).

[7] Álvarez, C., Corbal, J., Salamí, E. and Valero, M.: Initial Results on Fuzzy Floating Point Computation for Multimedia Processors, *Computer Architecture Letters*, Vol. 1, No. 1, pp. 1–4 (2002).

[8] Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).

[9] Huang, J. and Lilja, D. J.: Exploiting Basic Block Value Locality with Block Reuse, *Proc. 5th Int'l Symp. on High-Performance Computer Architecture (HPCA-5)*, pp. 106–114 (1999).

[10] 津邑公暁，清水雄歩，中島康彦，五島正裕，森眞一郎，北村俊明，富田眞治：ステレオ画像処理を用いた曖昧再利用の評価，情報処理学会論文誌コンピューティングシステム，Vol. 44, No. SIG 11(ACS 3), pp. 246–256 (2003).

[11] HAL Computer Systems/Fujitsu: *SPARC64-III User's Guide* (1998).

[12] MOSAID Technologies Inc.: *Feature Sheet: MOSAID Class-IC DC18288*, 1.3 edition (2003).

© 2015 Information Processing Society of Japan