デバッグ情報とシグナルを用いたメモリエラーの修復

濱田 槙亮[1,a)]　穐山 空道[2,b)]　並木 美太郎[1,c)]

概要：Approximate Computing は新しい省電力化手法であり、定められたハードウェアの処理時間や比率を変更することによって、通常では達成できないレベルでの電力削減が見込める。しかし、副作用としてデータのエラー率が上昇し、アプリケーションの実行に影響を及ぼし実行が中断されることもある。特に、エラーによって浮動小数点が NaN へ変化すると、実行が継続できなくなる要因となる。本研究は、Approximate Computing をメインメモリへ適用した環境を想定し、浮動小数点のデータに致命的なエラーが起きた場合に、実行が継続でき効果的にデータを修復する手法を提案する。NaN による例外をシグナルによって検知し、メモリやレジスタを書き換えて実行を継続させる。書き換える値には、デバッグ情報から得られるデータの型や論理的構造をもとに周辺データを取得して元のデータを推測する。提案システムを実装し、擬似的な Approximate Computing 適用環境の中で、エラーが起きても継続して実行ができることを確認した。またデバッグ情報を基に推測してデータを修復することで、計算結果への誤差を抑えることも確認できた。

Shinsuke Hamada[1,a)]　Soramichi Akiyama[2,b)]　Mitaro Namiki[1,c)]

1. はじめに

近年、HPC 分野や AI 分野の計算タスクの実行に大容量のメモリが要求される。他にもインメモリ KVS などの登場により大容量のメモリが要求され、サーバーマシンに搭載されるメインメモリの容量は増加している。メインメモリの容量の肥大化に伴い、サーバーマシンが消費する電力のうちメインメモリが占める割合も増加している。そのためメインメモリの消費電力を削減は、マシン全体の消費電力の改善が期待できる。メモリの消費電力を削減する研究は、従来の DRAM を最適化するものや [1], [2], [3]、不揮発性メモリに関する研究があり [4], [5], [6]。盛んに研究が行われている。

メモリの消費電力を削減する手法の一つに Approximate Computing が存在する。Approximate Computing は、本来決められたハードウェアの処理にかかる時間や比率を増加もしくは削減することによって消費電力の削減を図る手法である。計算結果の正確さや内容の一意性が保証される環境においては、通常では達成できないレベルでの消費電力削減が期待できる。しかし、Approximate Computing の副作用に、エラー率の上昇がある。メモリに対して Approximate Computing を適用した場合、エラー率が上昇しメモリのデータが突然化ける可能性も上昇する。メモリエラーが多発することは、通常のアプリケーションの実行では大きな問題であり、実行の中断や計算結果に大きな誤差が含まれる。しかし、HPC や AI 分野の数値計算アプリケーションは、繰り返し計算を行い計算結果が収束していく性質を持っており、メモリエラーによる計算誤差が混入しても収束によって誤差の影響を抑えられる。

メモリに対して Approximate Computing を適用すると、メモリのエラー率の増加を招き bit-flip などのメモリエラーが多発する。数値計算アプリケーションではデータに浮動小数点を扱うものが多く、浮動小数点がメモリエラーによって Not a Number (NaN) に化けることは大きな問題である。メモリエラーによって NaN が発生した場合、そのアプリケーションの実行結果は意味を失ってしまうか実行が継続できなくなる。

Approximate Computing の問題点を解決するために、メモリエラーによって NaN が混入しても継続して実行できる環境が必要である。また、エラーによる計算結果への影響を抑えるために、エラーによって失われたデータを復元する必要もある。

1 Tokyo University of Agriculture and Technology
2 The University of Tokyo
a) hamada@namikilab.tuat.ac.jp
b) akiyama@ci.i.u-tokyo.ac.jp
c) namiki@cc.tuat.ac.jp

$$\begin{vmatrix} a & NaN & c \\ d & e & f \\ g & h & i \end{vmatrix} = NaN$$

$$\begin{pmatrix} NaN & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{pmatrix} NaN & NaN & NaN \\ d & e & f \\ g & h & i \end{pmatrix}$$

図 1 行列演算での NaN の影響: NaN を含む行列の行列式の結果は NaN、行列演算では結果は伝搬する

本研究では、メインメモリに対して Approximate Computing を適用し、数値計算を行うアプリケーションを想定し、上記の問題の解決を図る。本稿では、OS のシグナルを用いて浮動小数点例外をキャッチしてメモリエラーの発生を検知し、実行の中断を抑制しながら実行を再開できるシステムと、本来あったデータをデータのアドレスやデバッグ情報などから推測して数値的誤差を抑えるシステムを提案する。

2. Approximate Computing/Memory

2.1 Approximate Computing/Memory の概要

Approximate Computing は消費電力を削減する手法の一つであり、正確な結果を要求する場合では達成できないレベルでの消費電力の削減を可能にする。メインメモリの消費電力削減は大きな効果が期待できるため、Approximate Computing をメモリに適用する研究は盛んに行われている [7], [8], [9]。

Approximate Computing は正確な結果を要求しないため、計算途中でデータが勝手に変化し結果に誤差が含まれる可能性がある。しかし、繰り返し計算を行い最終結果が収束していくような特性を持つアプリケーションであれば、エラーによる計算誤差は収束によって誤差が小さくなり、エラーの影響を比較的抑えることができる。そのため繰り返す計算を行う傾向が強い HPC 分野や AI 分野のアプリケーションが Approximate Computing に適している。

メモリに対する Approximate Computing の適用例の一つに、DRAM のリフレッシュレートを下げるものが存在する。文献 [10] では、DRAM のリフレッシュに必要な電力の 73%の削減に成功している。DRAM はデータを正常に保持するために定期的にリフレッシュを行う必要があり、リフレッシュレートを低下させることで、メモリセルの充放電にかかる電力を削減することができる。副作用としてリフレッシュが行われないためメモリセルのデータが正確に保持されている保証がなくなる。結果として bit-flips などのメモリのエラーの発生確率が増加し、メモリのデータを操作せずともデータが変化する可能性が高くなる。

2.2 NaN による問題

Approximate Computing は副作用としてエラー率の増加させる。データのエラーによってはアプリケーションに致命的な影響をもたらす可能性があり、数値計算を行う上で頻繁に用いられる浮動小数点は、メモリエラーによってアプリケーションの実行自体が中断する可能性がある。浮動小数点の表現の一つに Not a Number (NaN) があり、IEEE によって指数部のビットが全て 1 である場合の浮動小数点であると定義されている。NaN を用いた浮動小数点演算の結果は NaN になることも定義されており、計算途中で NaN が混入すると結果は NaN となる。図 1 は、NaN を含む行列の演算の影響を示している。図の上部は行列式を求めているが、NaN が 1 つでも存在すると結果は NaN になる。図の下部は、行列同士の乗算についてである。片方の行列に存在する 1 つの NaN が、計算結果の 1 行もしくは 1 列の値全てが NaN になる。1 つの NaN が複数の NaN となって伝搬していく可能性があり、計算結果から有意性が失われる。また意図しない NaN は例外を発生させ、アプリケーションの実行が中断する可能性がある。

浮動小数点を扱うアプリケーションにおいて、意図しない NaN が混入することは計算結果の有意性や実行自体の可否に関わる大きな問題である。このことから、Approximate Computing を適用する場合には、NaN が発生しても計算処理を継続できて計算結果への影響は数値的程度に留める実行環境が必要となっている。

3. 提案手法

本稿では、メモリエラーを検知して実行を継続でき、メモリエラーによって損なわれたデータを推測するシステムを提案する。提案するシステムは大きく以下の 2 つのサブシステムから構成され、1 つ目のサブシステムを「検知・実行継続システム」、2 つ目のサブシステムを「データ推測システム」と呼ぶ。

- シグナルハンドラを用いて NaN を検知し、レジスタやメモリの値を変更することで実行を継続するシステム (検知・実行継続システム)
- 実行ファイルの情報からデバッグ情報を得て、メモリエラーによって損失したデータの値を推測するシステム (データ推測システム)

3.1 検知・実行継続システムの基本方針

NaN を用いた浮動小数点演算は例外を起こし、その後の対処は OS・処理系によって決まっている。Linux では例外を発生させたアプリケーションに対して SIGFPE シグナルを発行し、アプリケーションのシグナルハンドラが処理を行う。この仕組みを利用してメモリエラーの検知を実現する。メモリエラーによって NaN が発生し計算に用いられた場合、浮動小数点例外を起こす。OS は SIGFPE シグナルをアプリケーションに対して発行するため、これをキャッチすることでメモリエラーの検知が行える。

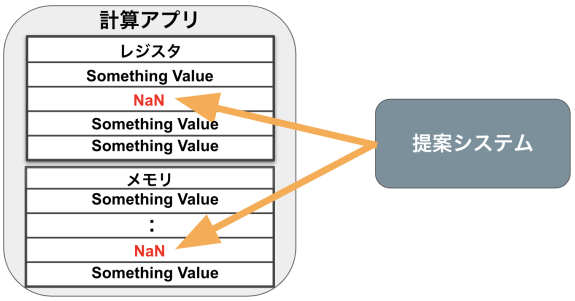


図 2 浮動小数点例外を引き起こした NaN は、レジスタとメモリにあるため、それらを書き換える。

実行を継続させるためには、例外の原因となった NaN を書き換える必要がある。例外を引き起こした NaN はレジスタに格納されている。実行の継続自体は、レジスタの NaN を書き換えて実行を再開すれば、アプリケーションは例外が起きたことを知らずに計算を再開する。しかし、メモリエラーが根本的な原因であるためレジスタの NaN は元々はメモリからロードされたものであり、メモリエラーによって生じた大元の NaN はメモリに存在する。メモリの NaN はそのままであるため、再度レジスタへロードされて計算に利用されると例外を起こしてしまう。これを防ぐために、メモリにある NaN も妥当な浮動小数点に置き換える必要がある。置き換えの対象はレジスタとメモリの NaN の 2 つとなる。提案システムは、対象アプリケーションの外側からメモリとレジスタを調べ、NaN を書き換える。(図 2)

3.2 データ推測システムの基本方針

検知・実行継続システムでは、メモリエラーの検知とレジスタ・メモリにある NaN の書き換えを行う。具体的にどのような値に書き換えるかはデータ推測システムが決定する。データ推測システムでは、メモリエラーで変化した値がアプリケーションでどのように扱われていたか推測し、他のメモリの値を用いて本来の値を推測値を計算する。

置き換える値として 0 などの定数があるが、アプリケーションによっては 0 除算になり問題である。他に置き換える候補として有力だと考えられる数値に 1 があるが、アプリケーションの特性を無視した定数では、計算結果に大きな数値的誤差が生まれる可能性がある。データ推測システムは、他の正常なデータを参照してエラーの修復を行い、計算誤差の抑制を狙う。

エラーが起きたメモリに定数を入れた場合と、エラーが起きたアドレス周辺の値を使用した値を入れる場合の差を、物理シミュレーションを可視化して確かめた。図 3 は、電磁波が空間を伝搬していく様子のシミュレーション結果を可視化したものであり、その 535 ステップ目の状態である。中央に電流源を配置し、そこから電磁波が放射状に拡がっていくシミュレーションを行う。(a) は、正常に実行した

表 1 文献 [11] の Table の Table 1 より引用。探索対象の命令群。

arithmetic	addss, addpd, addsd, addps mulss, mulpd, mulsd, mulps, subss, subpd, subsd, subps, divss, divpd, divsd, divps
mov	movss, movsd, movd

ものである。(b) は、200 ステップ目にエラーが入り、エラーを 0 で書き換えたものである。(a) と比較すると、画像の中央から左の位置に値を書き換えた影響が見られる。シミュレーション結果をアニメーションにすると、書き換えの影響が見られる位置に常に影響が出ている。(c) は、(b) と同様に 200 ステップ目でエラーが入り、エラーを書き換えている。書き換えた値は、エラーが起きた値のアドレスの前後 2 つの平均値である。単純な定数では計算結果の誤差が大きくなる可能性があり、他のデータや情報を利用して値を予測することで誤差を抑えられる可能性がある。

そのためには、データの利用方法を解析し、アプリケーションに合った値を入れる必要がある。アプリケーション内でどのように扱われているかは、実行ファイルに付属しているデバッグ情報を利用する。デバッグ情報には、アプリケーションで使用する関数や変数の情報が格納されている。変数の型や配列の大きさや次元数を得ることでき、構造体もメンバについての情報も格納されている。データ推測システムでは、デバッグ情報を解析しメモリエラーが起きた値の用途を推測する。例えば、メモリエラーが起きた値が配列の一部であった場合では、配列の他の値を用いることで元の値に近い値を計算できる可能性がある。

4. 実装

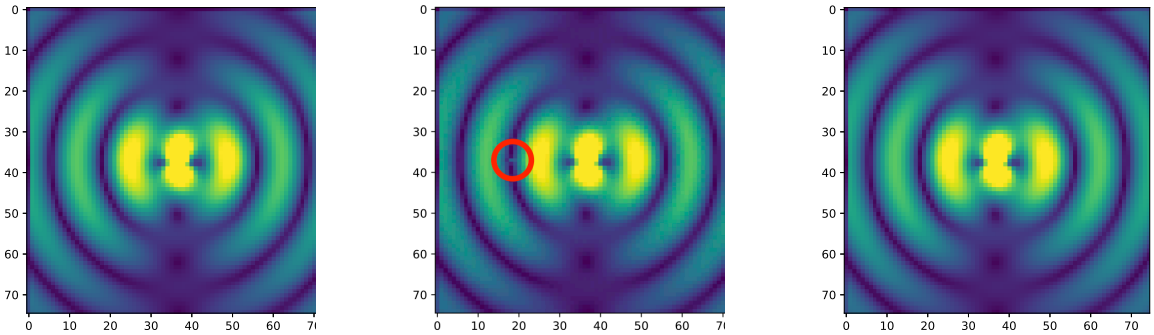
提案システムには、シグナルのキャッチやレジスタやメモリの値の読み書きが必要であるため、gdb の機能を利用する。Approximate Computing を適用する対象のアプリケーションは、gdb がアタッチした状態で実行される。gdb のアタッチによる実行時間のオーバーヘッドだが、シグナルをキャッチしないと gdb が割り込まないためエラーが起きないとオーバーヘッドは存在しない。

提案システムの実行の流れは以下の通りである。 (図 4)

(1) gdb がアタッチして対象アプリケーションの実行を開始
(2) メモリエラーの発生を検知・実行継続システムが検知
(3) データ推測システムがデバッグ情報から値を推測し、レジスタとメモリを書き換える
(4) 対象アプリケーションの実行を再開する (メモリエラーが起きた場合は (2) に戻る)

4.1 検知・実行継続システムの実装

gdb がシグナルをキャッチした時点のアプリケーションの状態は、例外を引き起こした時の状態である。そのた



(a) 正常な実行結果　(b)0 で書き換え　(c) エラー周辺の平均値で書き換え

図 3 電磁場シミュレーション 535 ステップ目の状態 (b,c には 200 ステップ目で値を書き換え)。赤丸にメモリエラーの影響が見られる。

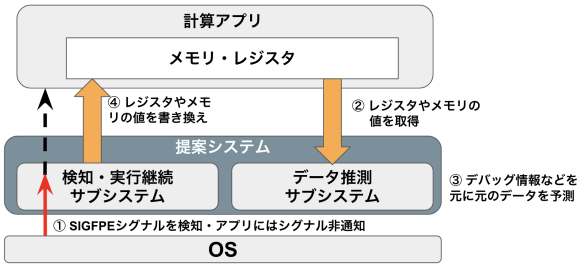


図 4 提案システム: 1. OS からの SIGFPE シグナルを検知する。対象のアプリにはシグナルは非通知にする。 2. レジスタやメモリの値を取得する。 3. デバッグ情報などを基にデータを推測する。 4. 推測された値でレジスタやメモリを書き換える。

め命令レジスタは例外を起こした命令を指しており、命令オペランドから NaN が入っているレジスタを特定できる。図 5 は、SIGFPE を検知して実行が中断された状態での、命令レジスタ (rip レジスタ) が指している機械語命令付近である。浮動小数点例外を起こした命令は、4 行目の **mulsd** 命令であり、これはオペランドでの積算を行う命令である。そして、オペランドには **xmm0** レジスタの値と **r9+rcx*8** で指すメモリの値が使用されている。オペランドのそれぞれの値を確認することで、どちらに NaN が存在するか特定できる。

本研究では、メモリに Approximate Computing の適用を想定しており、メモリのエラー率が上昇している。そのため、エラーによって発生した NaN はメモリに存在する。オペランドのレジスタに NaN が存在した場合、メモリエラーによって生じた大元の NaN が存在したメモリのアドレスを特定する必要がある。レジスタを使用する直前で、メモリからレジスタへロードする命令が実行される可能性が高く、機械語命令をバックトレースして遡ることでロード命令を発見することができる。機械語のバックトレースの方法については、[11] で提案している。図 5 の 1 行目の命令は、**movsd** 命令であり **xmm0** レジスタへメモリから値をロードしている。この時のオペランドの値から、メモリエラーによって生じた大元の NaN が存在したメモリアドレスを得られる。図 5 の場合では、**r10+rsi*8** で指す先からロードしているため、**r10** レジスタと **rsi** レジスタの値からアドレスを計算できる。

機械語命令を遡ることによってメモリエラーによって生じた大元の NaN を特定できる確率は、SPEC CPU 2006 のベンチマークアプリケーションで調査したところ 95%以上であることを確認した。図 6 は、浮動小数点を演算で使用する命令と、その命令で使用する値をメモリからロードする **mov** 命令を探し、対応する **mov** 命令が見つかった確率を示している。探索の対象とした命令は、表 1 の通りである。四則演算で使用する命令を対象としている。同種の **add** 命令が複数存在するのは、計算を行うときのレジスタ幅が異なるためである。

この手法で高い確率で、メモリエラーによって生じた大元のメモリの NaN を特定でき、NaN を修正することが可能であり、例外が発生しても、アプリケーションの実行を継続できる。メモリアドレスを特定できない場合は、レジスタの NaN のみを書き換えて実行を継続する。

4.2 データ推測システムの実装

データ推測システムは実行ファイルのデバッグ情報を使用する。デバッグ情報は DWARF というフォーマットで格納されており、これを解析することで変数の情報を得られる。**gcc** では**-g** オプションを付与することで、デバッグ情報を持った実行バイナリをコンパイルできる。デバッグ情報の解析には DWARF を解釈できるライブラリである **elfutils** の **eu-readelf** コマンドを使用した。

まず、検知・実行継続システムで既にメモリエラーが起きたアドレスが得られている。そのアドレスがアプリケーション内のどの変数か特定する必要がある。デバッグ情報にある変数の情報には、変数のアドレスも格納されているため、これと検知・実行継続システムで得られたメモリアドレスを比較していく。配列は次元数やその次元の大きさ

```
   0x5555555549ff <calculate()+79>:     movsd  xmm0,QWORD PTR [r10+rsi*8]
   0x555555554a05 <calculate()+85>:     add    edx,edi
   0x555555554a07 <calculate()+87>:     cmp    eax,r8d
=> 0x555555554a0a <calculate()+90>:     mulsd  xmm0,QWORD PTR [r9+rcx*8]
```

図 5 文献 [11] の Figure 3 より引用。浮動小数点例外を起こし SIGFPE シグナルを受け取った時の計算アプリの実行コンテキスト

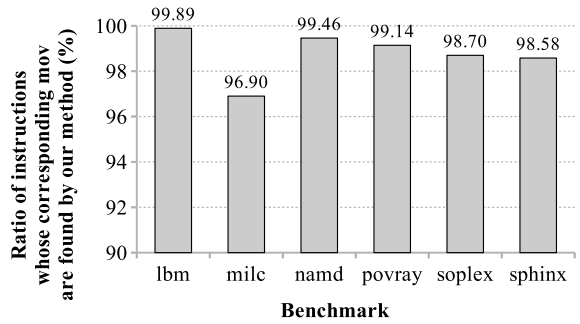


図 6 文献 [11] の Figure 6 より引用。浮動小数点を使用するベンチマークそれぞれにおいて、浮動小数点演算命令に対応する mov 命令が見つかった確率を示している。

```
[   303]    array_type
            type                    (ref4) [   319]
            sibling                 (ref4) [   319]
[   30c]      subrange_type
              type                    (ref4) [    86]
              upper_bound             (data1) 9
[   312]      subrange_type
              type                    (ref4) [    86]
              upper_bound             (data1) 19
[   319]    base_type
            byte_size               (data1) 8
            encoding                (data1) float (4)
            name                    (strp) "double"
[   320]    variable
            name                    (strp) "sample_matrix"
            decl_file               (data1) 1
            decl_line               (data1) 3
            type                    (ref4) [   303]
            external                (flag_present)
            location                (exprloc)
             [   0] addr 0x601060 <sample_matrix>
```

図 7 デバッグ情報の解析結果の一部。配列の sample_matrix についての情報が載っている。

も得られるため、配列のメモリ空間内でエラーが起きた場合でも特定は可能である。

図 7 は、**sample_matrix** という配列に関するデバッグ情報の一部である。左側の括弧に囲われた数字は、オフセットを示している。オフセット 320 に、**sample_matrix** の情報があり、細かい情報などはオフセットで指定した参照先に存在している。また一部の情報は入れ子になっており、**subrange_type** は **array_type** の子情報となっている。図の例では、**sample_matrix** の先頭アドレスは、**0x601060** となっている。**sample_matrix** の type をたどると、**array_type** になっており **sample_matrix** は配列である。また、子情報の **subrange_type** 情報が 2 つあることから 2 次元配列であることと、一つ目の次元の大きさは **upper_bound** から 10 であること、2 つ目の次元の大きさは 20 である。配列の型は **base_type** 情報から得られ、8byte のサイズの浮動小数点となっている。

表 2 評価実験の動作環境とソフトウェアのバージョン

OS	Ubuntu 16.04 (Linux 4.4.0-92-generic)
CPU	Intel Core i7 870 (2.93GHz)
gcc	5.4.0
gdb	7.11.1
python	2.7.12

これらの配列の型情報と次元数、各次元の大きさから、その配列のメモリ空間がどこまで計算できる。メモリエラーのアドレスが、その空間内にある場合はその配列のデータの一部であると判断できる。データ推測システムでは、これらの情報を活用してデータ構造を推測していく。

アプリケーションで使用するデータは、単一の変数や配列、malloc などで動的に確保したものがある。デバッグ情報を用いた解析では、単一の変数や静的に宣言された配列、ローカル変数にある配列が対応できる。しかし、malloc などで確保した場合はデバッグ情報のみでは対応できない。提案システムが得られる情報は、例外発生時のコンテキストとメモリエラーのアドレス、デバッグ情報のみである。アドレスだけでは、どの型もしくは構造体で malloc を実行したのか分からない。データ推測システムは、静的変数やローカル変数の変数や配列には対等できるが、malloc によって確保された領域でメモリエラーが起きた場合は対応できていない。

データ推測システムでは、そのデータがアプリケーションでどのように使用されているかを特定するまでに留めている。実際にどうやって推測するかはプログラマに委ねることにする。推測の方法として、平均値や最小二乗法による近似式の導出などが考えられる。実際にどうやって推測するかは、アプリケーションの特性を熟知した人間が記述することによって、最も効果的な値の修正が実現できる。本提案システムの動作は、python で記述されている。同様に推測部分についても python で記述できる。

5. 評価

提案手法を適用した状態で、アプリケーションの実行時間への変化を確認した。Approximate Computing は消費電力の削減を狙った制御であり、提案手法を適用することによる実行時間の増大は消費電力の増加となる。提案手法

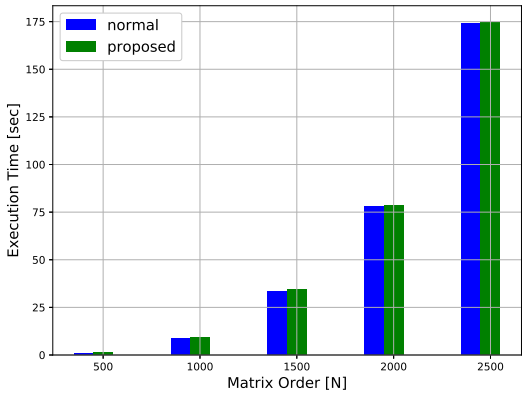


図 8 実行時間の計測結果: 青が通常の実行、緑が提案手法適用

によるオーバーヘッドがどの程度の時間なのか確認するために、提案手法の適用の有無による実行時間の差の比較を取った。また、提案手法の効果の一例として、物理シミューレションを行うアプリケーションに提案手法を適用し計算結果の数値的誤差を確認した。データの修復を複数の方法で試し、計算結果に与える影響について観測した。

Approximate Computing によるメモリエラーを再現するために、今回の評価ではソースコードを改変して意図的に NaN をデータ内に入れた。

以上の評価実験を表 2 に示す環境で実施した。

5.1 オーバーヘッドの計測

オーバーヘッドを計測するために用いたアプリケーションは、2 つの 2 次元正方行列で掛け算を行う。アプリケーション内では、正方行列は 2 次元配列として定義した。また、NaN の意図的な混入は一度だけ行った。計測実験は、行列の大きさを変えながら複数回計測した平均を取った。

最初に比較対象として、提案手法を適用せず意図的な NaN の混入もない行列同士の掛け算の実行時間を計測した。次に、提案手法を適用した状態の実行時間を計測した。データ推測システムでは、実際に得られたメモリアドレスとデバッグ情報から配列の構造を把握し、2 次元配列の上下左右の 4 つの値の平均値を出すようにした。レジスタとメモリの NaN は平均値で書き換えて、アプリケーションの実行を再開させる。

5.2 オーバーヘッドの計測結果

2 種類の状態での実行時間の計測結果は、図 8 である。グラフの横軸は行列の大きさを示し、縦軸は実行時間である。**normal** が通常状態での実行時間で、**prposed** は提案手法を適用した状態での実行時間である。2 つの計測結果の差が、提案手法によって発生したオーバーヘッドであり、表 3 に示す。

オーバーヘッドは、行列の大きさとは独立して一定であり、今回の計測では平均値は 0.6 秒だった。実行時間は行列の大きさに対して自乗で増えているが、オーバーヘッドは一定であるため、行列が大きくなるほど相対的にオーバーヘッドが実行時間のうちで占める割合は少なくなる。このオーバーヘッドは、検知・実行継続システムが gdb でシグナルをキャッチしてからアプリケーションの実行が再開されるまでの時間であり、再度メモリエラーが発生した場合、同程度の時間のオーバーヘッドが発生する。NaN のメモリアドレスを特定できずにレジスタのみを書き換えている場合は、再度メモリからロードされて使用されると浮動小数点例外が起きるため、オーバーヘッドが発生する。

表 3 オーバーヘッドの計測結果: 通常の実行と提案手法適用時の実行時間の差

Matrix Order	normal (sec)	proposed (sec)	ovehead (sec)
500	0.610	1.363	0.753
1000	8.758	9.201	0.443
1500	33.556	34.245	0.689
2000	77.990	78.295	0.305
2500	173.797	174.617	0.820

提案手法の適用によるオーバーヘッドは、メモリエラーのたびに発生し、0.6 秒程度である。提案手法のプロセスが 1 度実行される時のオーバーヘッドは計算時間に対して十分小さく無視できる程度であった。

5.3 データ推測による数値的誤差

物理シミュレーションを行うアプリケーションでメモリエラーが発生した場合を想定し、書き換える値によって計算結果がどうなるか確認した。

対象のアプリケーションは、時間領域差分法 (Finite-Difference Time-Domain method: FDTD 法) による電磁場のシミュレーションを行う。アプリケーションでは、空間の中央に電流源を設置して交流電流を流し、空間に電磁場が拡がっていく様子をシミュレーションする。電場と磁場のそれぞれで x 方向、y 方向、z 方向の値を配列で保持していて、配列は静的変数として定義している。シミュレーションは 1000 ステップ分を計算し、200 ステップ目で意図的にデータに NaN を入れ込む。メモリエラーが起きた箇所に修正値として書き込む値は、以下の 5 種類とした。

- 定数の 0
- メモリエラーのアドレスの前後 2 つの値の平均値
- メモリエラーのアドレスの前後 4 つの値の平均値
- 配列の上下左右の 4 つの値の平均値
- 配列の上下左右の 8 つの値を使った Linear Regression

結果は、図 9 と図 10 である。縦軸は、正常なシミュレーション結果との計算誤差をシミュレーション空間全てで足し合わせたものをステップごとに求めたもので、横軸はステップである。横軸が進むほどにシミュレーションが進んでいる。メモリエラーが発生し修正を行っている 200 ス

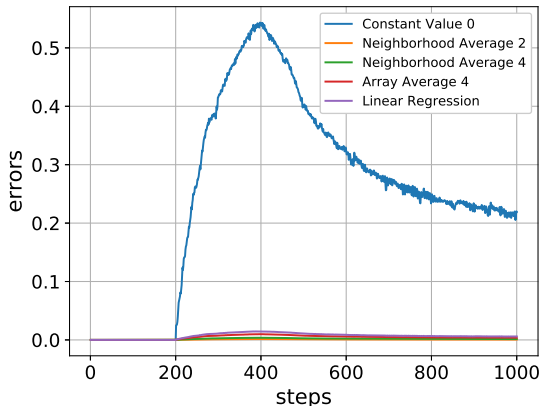


図 9　意図的なエラー混入を修正した時の、シミュレーション結果の誤差 (エラーを入れた位置は、電流源の位置)

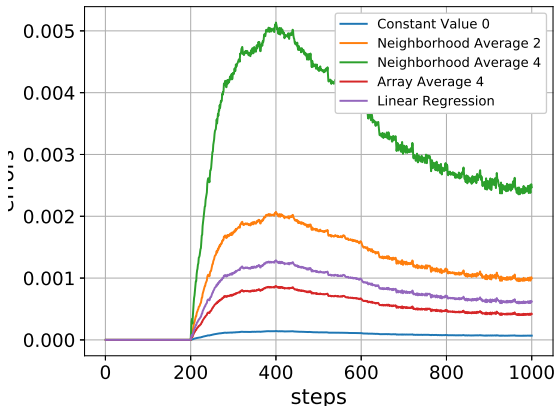


図 10　意図的なエラー混入を修正した時の、シミュレーション結果の誤差 (エラーを入れた位置は、電流源から x 軸方向にずらした位置)

テップ目で、計算誤差が発生してステップが進むごとにエラーは一定の値に収束している。2 つの図では、エラーの混入する位置が異なっており、図 9 は電流源の位置にエラーを混入した。図 9 では、電流源から x 軸方向にずらした位置にエラー混入した。

結果を比較すると、メモリエラーの位置によってどの修正が適しているかは異なっている。しかし、図 9 では、定数 0 で書き換えると他の方法と比較して計算誤差が大きくなっている。図 10 では、0 で書き換えても計算誤差は他の方法と同程度であるが、これは元の値が 0 に近いためだ。対して、デバッグ情報から得られた情報を使用して得られた値で修復した方法では、多少の差はあるが誤差は同程度となった。

定数での書き換えは、大きな計算誤差となる可能性があり、デバッグ情報を駆使して配列の情報を使った予測の方が誤差が抑えられる結果となった。

6. 議論

6.1 アプリケーションに合わせた値の修復方法の選択

本稿では、データ推測システムで実際に推測する部分をプログラマに委任している。例外発生時のコンテキストと実行ファイルから得られる情報のみで、一意に最適な推測方法を決定できないためである。またデータは、アプリケーションに依ってセンシティブな情報であるため、断片的な情報での一方的な推測では計算誤差を増加させる可能性がある。そのため、本研究では特定のアプリケーションに依存する方向は避けて、依存箇所についてはそれぞれのプログラマや実行者によって記述でき、推測方法は簡単に記述できる。

アプリケーションの特性に依存した箇所を人間に任せずに、既存の情報のみで解決できることが理想であるが、今後の課題として取り組む。

6.2 Approximate Computing の適用方法に依るエラーの特性

Approximate Computing は、適用方法が数多く存在する。DRAM の Approximate Computing に関する研究では、DRAM のリフレッシュレートの低下や DRAM のセルの充電時間を短縮するものが存在する。適用方法によっては、ビットが 0 もしくは 1 のどちらかに偏っているや、エラーがバーストして発生する、といった特徴があることが考えられ、適用方法のエラーの特徴をデータ推測の手掛かりにできる可能性がある。

適用方法によるエラーの特徴を確認するために、実際に Approximate Computing を適用して確認し検討する必要がある。

7. 関連研究

提案手法は、エラーによる例外を検知して、エラーを受動的に修正している。似た仕組みを持つものとして、故障が起きてもシステムが動作し続けることが求められる fault tolerance がある。　fault tolerance に関する研究では、アプリケーションが中断されてから復帰させるためにコンテキストを変更する方法を用いているものがある。

LetGo[9] は、ポインタ変数がエラーによって化けた場合に対処する方法について提案している。エラーによってポインタ変数の指す先が不正なものになった場合、アプリケーションに対して SIGSEGV シグナルが送られる。LetGo はこのシグナルをキャッチし、アプリケーションのコンテキストとレジスタを修正して実行を継続させる。我々の提案手法もシグナルをキャッチして、メモリやレジスタを書き換えており、似た点がある。しかし、デバッグ情報を用いることでより多くの情報を利用している。ポイ

ンタを上手く修正する方法は、数値データの修正の参考になる可能性がある。

Jolt[12] は、エラーによってアプリケーションが無限ループに陥った時に、ループから脱出させる手法である。ループの反復ごとに、アプリケーションの状態を記録し、同様の状態が検出された場合に無限ループに陥ったと判断する。提案手法は、AI や HPC 分野の数値アプリケーションを想定しており、扱う数値データは巨大であり逐次保存していない。しかし正常なデータは、元のデータの推測に大きく役立つ情報である。アクセス頻度の高いデータを識別し保存しておくことで、**Jolt** の手法を参考にできる。

8. おわりに

Approximate Computing は、定められたハードウェア処理の時間や比率を増加もしくは削減することによって、計算結果や保存されたデータの一異性を失う代わりに消費電力の削減を図る手法である。浮動小数点を扱うアプリケーションでは、メモリエラーによって浮動小数点が NaN へ突然化けることは、アプリケーションの実行中断か計算結果の有意性の損失となる。

本稿では、OS のシグナルを用いて浮動小数点例外をキャッチしてメモリエラーの発生を検知し、実行の中断を抑制しながら実行を再開できるシステムと、本来あったデータをデータのアドレスやデバッグ情報などから推測して数値的誤差を抑えるシステムを提案した。提案手法の適用による実行時間のオーバーヘッドを計測し、オーバーヘッドは無視できる程度であることを確認した。また、デバッグ情報を用いたデータの推測による値の修復によって定数や単調な推測よりも計算エラーを抑えられることを確認した。

参考文献

[1] Tao Zhang, Ke Chen, Cong Xu, Guangyu Sun, Tao Wang, and Yuan Xie. Half-dram: A high-bandwidth and low-power dram architecture from the rethinking of fine-grained activation. In *International Symposium on Computer Architecture (ISCA)*, pp. 349–360, 2014.

[2] Yabin Lee, Hyeonggyu Kim, Seokin Hong, and Soontae Kim. Partial row activation for low-power dram system. In *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 217–228, 2017.

[3] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R. Alameldeen, Donghyuk Lee, and Onur Mutlu. Detecting and mitigating data-dependent dram failures by exploiting current memory content. In *International Symposium on Microarchitecture (Micro)*, pp. 27–40, 2017.

[4] Atsushi Koshiba, Takahiro Hirofuchi, Soramichi Akiyama, Ryousei Takano, and Mitaro Namiki. Towards write-back aware software emulator for non-volatile memory. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, 2017.

[5] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 256–267, 2013.

[6] Yuliang Sun, Yu Wang, and Huazhong Yang. Energy-efficient sql query exploiting rram-based process-in-memory structure. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, 2017.

[7] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 213–224, 2011.

[8] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. In *International Symposium on Computer Architecture (ISCA)*, pp. 1–12.

[9] Bo Fang, Qiang Guan, Nathan Debardeleben, Karthik Pattabiraman, and Matei Ripeanu. Letgo: A lightweight continuous framework for hpc applications under failures. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pp. 117–130, 2017.

[10] A. Raha, H. Jayakumar, S. Sutar, and V. Raghunathan. Quality-aware data allocation in approximate dram*. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 89–98, Oct 2015.

[11] Shinsuke Hamada, Soramichi Akiyama, and Mitaro Namiki. Reactive nan repair for applying approximate memory to numerical applications. In *Workshop on Systems for Multi-core and Heterogeneous Architectures (SFMA)*, 2018.

[12] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *European Conference on Object-oriented Programming*, ECOOP, pp. 609–633, 2011.