Vol.2013-ARC-205 No.14
Vol.2013-OS-125 No.14
2013/4/26

SIMD 命令セットを用いた浮動小数点演算における精度低下検出

安仁屋 宗石[1]　吉田 浩章[2]　伴野 充[2]　北村 俊明[1]

概要：計算機の数値計算に用いられる浮動小数点演算では，桁落ち，情報落ち，丸めの影響により精度低下を引き起こす場合がある．現在，数値計算の浮動小数点演算規格として広く用いられる IEEE754 は，桁落ち，情報落ちに対する例外の定義が無いため，演算結果の信頼性保証が不十分である．
そこで本研究は，効率の良い精度低下検出を行うために，SIMD 命令セットを用いた精度低下検出手法を提案し，ソースコードを書き換えずに精度低下検出を行うシステムを，コンパイラインフラストラクチャである LLVM を用いて，精度低下検出コードをコンパイル時に組み込むことで実現した．その結果，SIMD 命令セットを用いた精度低下検出手法では，関連研究と比較して，精度低下検出を 24 倍高速化した．

1. はじめに

近年，計算機の著しい性能向上に伴い，幅広いの分野で計算機による数値計算が用いられている．計算機の数値計算に用いられる浮動小数点数は，殆どの場合において誤差を含んでいる．浮動小数点数の誤差は，浮動小数点演算を行う際に，桁落ち，情報落ち，丸めによる精度低下を引き起こす原因となる．現在，数値計算の浮動小数点演算規格として広く用いられる IEEE754[1] は，桁落ち，情報落ちに対する例外の定義が無いため，演算結果の信頼性保証が不十分である．また，ユーザーが精度低下に気付かずに演算を繰り返し行うことにより，実際とは全く異なった結果となる可能性がある．一方で，計算機で扱う計算の規模は大規模化の一途を辿り，精度低下が発生した箇所を特定することはより困難となっている．また，従来の精度低下検出は浮動小数点演算ごとに指数部の比較を行っており，あまり効率的ではなかった．

そこで本研究は，浮動小数点演算における精度低下検出を効率良く行うことを目的とし，SIMD 命令セットを用いて一度に複数の精度低下検出に伴う指数部比較をまとめて行う手法を提案する．また，コンパイラインフラストラクチャの一種である LLVM[2] を用いて，解析対象プログラムに SIMD 命令セットを用いた精度低下検出組み込むシステムを構築し，ソースコードを書き換えることなく，プログラムの実行時に精度低下検出が行えるようにした．さらに，どの程度効率良く精度低下検出が行えるか評価を行い従来手法と比較した．

[1] 広島市立大学
Hiroshima City University, 3-4-1 Ootsuka-higashi, Asami-nami, Hiroshima 739–2115, Japan
[2] Fujitsu Laboratories of America, Inc. 1240 E. Arques Ave. Sunnyvale, CA 94085 USA

2. 研究背景

計算機を用いて数値計算を行う場合，実数を有限な浮動小数点数を用いて表現するため，丸め等により殆どの場合誤差を含んでしまう．また，その誤差が原因となって，浮動小数点演算を行う際に精度低下が発生し，演算の結果が実際の値とは異なってしまう可能性がある．本節は，浮動小数点演算規格である IEEE754 について説明し，精度低下の要因となる桁落ち，情報落ち，丸め誤差について述べる．

2.1 浮動小数点演算規格

計算機では大抵の場合，数値を整数か浮動小数点数を用いて表現する．実数を扱う際には，一般的に浮動小数点数が利用され，仮数，基数，指数の 3 つで表現される．浮動小数点数を用いた演算は浮動小数点演算と呼ばれる．現在，最も広く用いられている浮動小数点演算標準である IEE754 は，基数を 2 とし，単精度，倍精度，4 倍精度等の形式と演算が定義されている．その中で，計算機を用いた数値計算において，最もよく用いられる倍精度の浮動小数点数で表現される値は式（1）で表すことができる．

$$(-1)^{exponent} \times 2^{exponent-1023} \times (1 + fraction) \quad (1)$$

指数部ではバイアス表現を用いるため，$\times 2^1$ を表現したい場合の指数部の値は 1024 となる．仮数部の整数部分は常に 1 と定められており，仮数部のビット幅は 52bit だが，

常に 1 となる暗黙の整数ビットを考慮すれば，その精度は 53bit である．これにより倍精度浮動小数点数は，実数を 53bit の有理数で近似して扱うことになる．この精度は 10 進数に換算すると，約 16 桁に相当する．また，単精度は，符号 1bit，指数部 8bit，仮数部 23bit から成る 32bit で表現され，4 倍精度は，符号 1bit，指数部 15bit，仮数部 112bit から成る 128bit で表現される．

2.2 桁落ち

桁落ちは，浮動小数点加減算において，演算結果の有効桁数が減少するこである．同符号で絶対値が非常に近い値同士の減算または，異符号で絶対値がほぼ等しい加算を行った場合，正規化によって有効桁数が減少することで発生する．桁落ちが発生する 10 進数の演算例を式（2）に示す．

$$1.003\times10^{10}-1.000\times10^{10} = 0.003\times10^{10} = 3.000\times10^{7} \quad (2)$$

演算前のオペランドは 4 桁の有効桁数を持つが，演算後は 1 桁しか有効桁数が無いことがわかる．また，左辺オペランドの「3」の桁が誤差を含む場合は，この桁落ちは精度低下を引き起こしている．しかし，3 の桁が正しく，それ以降の桁が全て 0 場合，精度低下は発生しない．

桁落ちによる精度低下は，誤差の影響が比較的少ない仮数部の下位ビットに位置していた不正確ビットが，桁落ちによって仮数部の上位方向にシフトされ，不正確ビットによる誤差の影響が大きくなることによるものである．

2.3 情報落ち

絶対値の大きさが極端に異なる値同士の加減算を行った場合，絶対値の小さい値が演算結果に反映されないことを情報落ちと呼ぶ．以下に 10 進数での有効桁数 5 桁式（3）と有効桁数 4 桁式（4）での例を示す．

$$2.0000 \times 10^4 + 1.0000 = 2.0001 \times 10^4 \quad (3)$$

$$2.000 \times 10^4 + 1.000 = 2.000 \times 10^4 \quad (4)$$

上記の通り，式（3）では正しい結果となるが，式（4）では，計算結果に絶対値の小さい値が反映されていないことが分かる．

浮動小数点加減算を行う場合，指数部の絶対値が大きい方に揃えて演算を行う．この時，絶対値の差が大きいと，小さい方の値は大きいく右シフトされ，仮数部の表現範囲から溢れてしまい，情報が欠落してしまう．これを回避する方法として，絶対値の大きく異なる加減算を行わないことが挙げられる．例として，値の総和を算出する際に，あらかじめソートを行い，絶対値が小さい値から計算する手法がある．

2.4 丸め誤差

丸め誤差は，10 進数では正確に表現できる値でも，2 進数では循環小数となってしまう値において，循環小数を端数処理することにより，仮数部の下位ビットが不正確になることにより発生する．IEEE754 に準拠しているシステムであれば，不正確例外処理によって丸め誤差を検出することができる．しかし，計算機が有限桁のフォーマットで数を処理する限り，丸め誤差自体を無くすことは不可能である．

3. 先行研究

近年，特に科学技術の分野において，より正確な数値計算の必要性が高まってきている．それに伴い，様々な浮動小数点演算における精度保証に関する研究が行われている．本節は，浮動小数点演算における精度低下検出に関連する研究を紹介する．また，精度低下を検出するだけでなく，自動で精度低下を回復する研究や多倍長精度演算についても簡単に紹介し，それらの研究と本研究を比較する．

3.1 動的桁落ち検出ツール

プログラマが容易に計算精度を見積もるために，プログラム実行時に桁落ちを検出する Lam らの研究 [3] がある．この中で提案されたツールは，Dyninst API library[4] を用いて実現する．

Dyninst API はオンラインモードとオフラインモードの 2 つのモードを持っている．オンラインモードは，ターゲットプロセス開始時に解析を開始し，目的の処理に置き換えたい箇所で一時停止する．そして，目的の処理を追加したヘルパー関数にジャンプするように置き換えたのち，プロセスをレジュームする．オフラインモードは，ターゲット実行ファイルを開き，処理の置き換えたい箇所を，目的の処理を追加したヘルパー関数にジャンプするように修正し，変更後の実行ファイルをディスクに書き戻す．

このツールはプログラム実行時に全ての浮動小数点加減算を桁落ちを検出する機能を追加したヘルパー浮動小数点加減算へジャンプするように修正する．ヘルパー関数では，通常の浮動小数点加減算を行うと同時に，演算引数と演算結果の引数を抽出し，演算引数の指数部の大きい方から演算結果の指数部を減算した差が 10 ビットを超えたら桁落ちと判定する処理を行う．精度低下検出対象のプログラム実行時に桁落ちが発生した場合，精度低下情報が書き込まれたログファイルが生成され，GUI を用いてユーザーに桁落ちした桁数，桁落ちが発生した命令アドレス，桁落ちが発生した演算のソースコードレベルにおける行数を通知することができる．またソースコードが無いバイナリのみのアプリケーションも精度低下解析を行うことができる．

しかし，プログラム実行時に，全ての浮動小数点加減算がヘルパー関数にジャンプするように置き換えられているため，関数呼び出しの回数が増加し，精度低下検出処理とは関係の無い処理で，実行速度が低下していると考えら

れる．このことから，本研究でも使用する，SPEC CPU 2006 に含まれる SPECfp2006 の 470.lbm を用いたベンチマークにおいて，70 倍の実行時間スローダウンがある．本研究は，コンパイル時に精度低下検出コードを組み込むため，プログラム実行時に精度低下検出に伴う，関数呼び出し等を行わないので，少ないスローダウンで精度低下検出が行えると考えられる．

3.2 動的な精度低下解析と精度低下回復

プログラム実行中に浮動小数点演算の精度低下を解析し，自動で精度低下を回復する Benz らの研究 [5] がある．この研究で提案されたシステムは，動的解析ツールのフレームワークである Valgrind[6] を活用して精度低下解析と精度低下回復を行う．このシステムは，Valgrind の監視下で実行される解析対象バイナリ中の全ての浮動小数点演算において，通常精度の演算を実行すると共に，より高精度な変数を使用した演算を並列に実行する．2 つの演算結果を比較して演算精度が低下していないか判定する．2 つの演算結果が異なっていた場合，通常精度演算結果を高精度演算結果に置き換える．これにより，通常精度の演算において精度低下が発生した場合，高精度演算の結果に置き換えることができ，自動的に精度低下を回復することができる．またこのシステムは，Valgrind 上で実行可能なプログラムであれば，精度低下解析を行うことができる．従って，解析対象のソースコードを書き換える必要は無い．

しかし，全ての浮動小数点演算を 2 重に行い，高精度演算は汎用プロセッサの浮動小数点演算器では行えず，さらに精度低下検出も行うため実行時間スローダウンが大きい．SPEC CPU 2006 に含まれる SPECfp2006 の 470.lbm を用いたベンチマークで，303 倍の実行時間スローダウンを伴う．本研究は，精度低下回復はできないが，SIMD 命令セットを用いた精度低下検出により，1 度に複数回分の精度低下検出を行うため，精度低下検出の実行時間スローダウンは少ない．

3.3 PC エミュレータを活用した精度低下検出

我々の研究室で行った PC エミュレータを活用した精度低下検の研究 [7] では，数多くのプロセッサアーキテクチャをエミュレートすることが可能な PC エミュレータである QEMU[8] に精度低下検出機能を追加した．

QEMU は TCG（Tiny Code Generator）と呼ばれる命令変換機構を持っており，ゲスト CPU の命令をホスト CPU の命令へ動的に変換する事でエミュレートを実現している．TCG は命令変換を行う過程で，一旦ゲスト命令を中間表現に変換する．TCG はゲスト命令と中間表現を動的にマッピングする．その際，1 つのゲスト命令に対して複数の中間表現で対応する場合がある．

この研究では，x86_64 アーキテクチャを対象に，浮動小数点加減算の中間表現に精度低下検出機能を追加した．これにより QEMU 上で解析対象プログラムを実行することで，ソースコードを改変することなく精度低下を検出し，QEMU の強力なサポートによって，多くの精度低下に関する情報をユーザーに通知することができる．また，QEMU 上でゲスト OS を動作させることができるため，ゲスト OS 上で動作するプログラムであれば，エクセル等のソースコードが入手不可能なソフトでさえ精度低下検出を行うことができる．

しかし，多彩な精度低下情報をユーザーに提供できる一方で，この手法の弱点として QEMU のオーバーヘッドが大きいため実行速度が遅い点が挙げられる．QEMU ではプログラム実行時に TCG を用いた複雑な命令変換を動的に行うため，精度低下検出処理とは関係の無い処理でオーバーヘッドが伴う．これにより，第 6 節で紹介する多次元積分においてユーザーモードエミュレーションで約 7 倍，フルシステムエミュレーションで約 9 倍の実行時間スローダウンに繋がった．本研究は，ユーザーに提供する精度低下情報は少ないが，コンパイル時に精度低下検出コードを解析対象のプログラムに組み込むことで，プログラム実行時に命令変換等を行わないため，精度低下検出以外の処理によるスローダウンは少ないと考えられる．また，SIMD 命令セットを用いた精度低下検出によって指数部の比較回数が削減されるため実行速度の面で有利だと考えられる．

3.4 有効桁数を追跡するシステム

浮動小数点数の有効桁数を追跡して，ユーザーに通知する鈴木らの研究 [9] がある．浮動小数点演算における精度低下は，浮動小数点演算ごとに桁落ち，情報落ち，丸めによる有効桁数の変化を追跡することができれば，演算結果の誤差を見積もることが可能である．しかしこの研究では，浮動小数点演算ごとに有効桁数を追跡するコストが非常に高いため，より深刻な誤差につながる桁落ちのみに着目している．

このシステムは桁落ち検出機能を，C++言語のオペレーターオーバーロードという演算子に対して演算以外の処理を定義することができる機能を活用して実現している．加算と減算の演算子にオペレーターオーバーロードを用いて，演算引数の指数部を抽出し，演算結果の指数部と比較する桁落ちを監視処理を追加し，演算を行うごとに，各変数が持つ有効桁数情報から演算結果の有効桁数を算出することで，有効桁数を追跡する処理を実現した．

しかし，このシステムの解析対象は C++で書かれたソースコードのあるプログラムに限られる，これはオペレーターオーバーロードを使用するために，解析対象のソースコードを書き換えてコンストラクタを追加する必要があるためである．また，加減算の多い解析対象ではオペレーターオーバーロードを多用するため，数十倍の実行時間を

Vol.2013-ARC-205 No.14
Vol.2013-OS-125 No.14
2013/4/26

要することが分かっている．本研究は，コンパイル時に精度低下検出コードを組み込むため，ソースコードを書き換える必要がない．また，SIMD 命令を用いて 1 度に複数回の精度低下検出処理を行えるので，加減算の回数が増加しても実行速度が大幅に遅くならないと考えられる．

4. 提案手法

浮動小数点演算の精度低下検出を行うには，浮動小数点加減算ごとに演算引数と演算結果の指数部を比較する必要がある．しかし，この比較はソフトウェアで記述すると処理量が多く，浮動小数点加減算ごとに行っていては精度低下検出による実行時間のスローダウンが大きくなってしまう．そこで提案手法では，指数部の比較回数を削減するために SIMD 命令セットを用いて，複数個の指数部を 1 つの SIMD レジスタにパッキングし，1 度の操作で複数回分の精度低下検出に当たる指数部比較を行う．さらに，精度低下検出対象のソースコードを書き換えること無く，精度低下検出を行うために，コンパイラインフラストラクチャの一種である LLVM を用いて，コンパイル時に精度低下検出コードを組み込むことにより，精度低下検出対象プログラムの実行時に自動で精度低下検出が行える機構を構築した．

4.1 SIMD

SIMD（Single Instruction Multiple Data-stream）は単一の命令で，複数のデータを処理できる並列アーキテクチャである．近年，メディア処理の高速化等を目的として，汎用プロセッサの多くが SIMD 拡張命令セットとして備えるようになった．固定長のベクトルレジスタを，8bit や 16bit 等のサブレジスタに分割し，ベクトルレジスタ同士の対応するサブレジスタ間で演算や操作を並列に行うものである．例として，Intel 社の SSE（Streaming SIMD Extensions）や 256bit のベクトルレジスタを用いる AVX(Advanced Vector eXtensions)[10] がある．

4.2 SIMD 命令セットを用いた精度低下検出

本節はまず，精度低下検出の処理量を示すために従来の精度低下検出手法について説明し，次に SIMD 命令セットを用いた 1 度に複数の精度低下検出が行える手法について述べる．

4.2.1 従来手法

従来の精度低下検出方法では，浮動小数点加減算ごとに，演算引数と演算結果の指数部を比較する必要がある．桁落ち検出では，演算引数の大きい方の指数部から演算結果の指数部を減算する．その差がユーザーが定義する桁落ちビット数を超えていないか比較し，超えていた場合は桁落ちと判定する．また，情報落ちでは，演算引数の指数部の差が 52 を超えると情報落ちと判定する．

疑似コード 1 に桁落ち，疑似コード 2 に情報落ちを検出する際の疑似コードを示す．

Algorithm 1 桁落ち検出の疑似コード

```
if exp_op1 > exp_op2 then
  big_exp ⇐ exp_op1
else if exp_op1 < exp_op2 then
  big_exp ⇐ exp_op2
else
  big_exp ⇐ exp_op1
end if
cancellation_digits ⇐ big_exp − exp_result
if cancellation_digits > USER_DEFINITION_VALUE
then
  桁落ち検出
end if
```

Algorithm 2 情報落ち検出の疑似コード

```
exp_difference⇐ exp_op1 − exp_op2
exp_difference⇐ abs(exp_difference)
if exp_difference> 52 then
  情報落ち検出
end if
```

このように，精度低下を検出する処理量は少なくなく，浮動小数点加減算ごとにこの処理を行うと，実行時間スローダウンが大きくなるのは明らかである．

4.3 提案手法

精度低下検出処理を高速化するために，従来は浮動小数点加減算ごとに行っていた指数部の比較を，SIMD 命令セットを用いて 1 度の操作で複数回分の指数部比較を行う手法を提案する．提案する精度低下検出コードは x86_64 アーキテクチャ向けとし，Intel 社の AVX 拡張命令セットを用いて実現した．

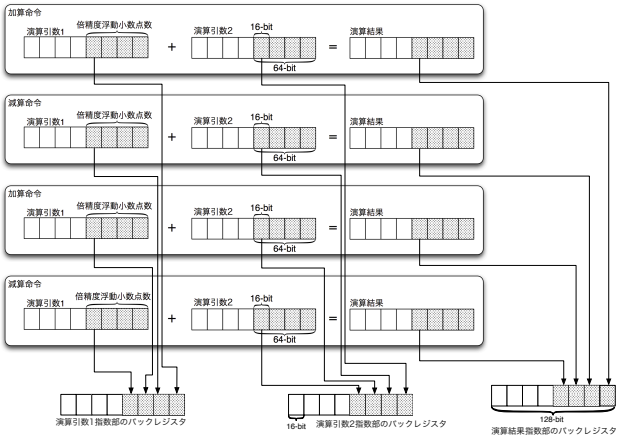

図 1 指数部をパックする様子

提案手法は，演算引数と演算結果の指数部を，それぞれ 1 つの SIMD レジスタに複数個まとめることによって，1 度で複数回分の精度低下検出に伴う指数部の比較を行うこ

とができる．まず，浮動小数点加減算ごとに，演算引数と演算結果の指数部を 1 つの SIMD レジスタにパックする必要がる．今回の実装では，倍精度の指数部が 11bit で表現されることから，指数部をパックするレジスタは 128bit の SIMD レジスタを 16bit 区切りとしたものを用いる．図 1 に示すように，まず，1 番最初の加算命令に着目し，演算引数 1，演算引数 2，演算結果の 3 つが格納されている SIMD レジスタから，それぞれの指数部を含む 16bit を抽出する．演算引数 1 から抽出したデータは演算引数 1 パックレジスタに，演算引数 2 から抽出データは演算引数 2 パックレジスタに，演算結果から抽出したデータは演算結果パックレジスタに格納する．次の減算命令も先の加算命令と同様に，指数部を含む 16bit のデータを，それぞれのパックレジスタにパッキングしていく．指数部は最大 8 個まで格納することができる．

次に各指数部レジスタから，指数部のみを抽出する必要がある．そこで，指数部のみを取り出すために，サブレジスタに対する 0x7FF0 が格納されたマスクレジスタと各指数部のパックレジスタに SIMD 論理積を使用して，複数の指数部を 1 回の操作で抽出する．このとき抽出された指数部は，実際の指数部より 16 倍大きくなってしまう．これは 16bit 単位でデータ転送を行った影響で，下位 4bit が不要になってしまうためである．

指数部を抽出した後は，桁落ちビット数を計算するために，演算引数の指数部の大きい方を取得する必要がある．そこで，各サブレジスタの組を比較して大きい方を格納する vpmaxsw 命令を用いた．各演算引数の指数部パックレジスタ内の互いのサブレジスタを比較するとこによって，大きい方の値だけ集まった演算引数パックレジスタを生成する．

従来手法では，桁落ちの判定を行う際に，桁落ちビット数とユーザー定義桁落ちビット数の比較を行ったが，提案手法では，桁落ちビット数とユーザー定義桁落ちビット数を比較する処理の代わりに，まず，あらかじめ演算結果パックレジスタの各指数部 E にユーザー定義桁落ちビット数を加算する．次に，演算結果パックレジスタから，演算引数の大きい方の指数部を持つ演算引数パックレジスタで飽和減算を行う，これによりユーザー定義桁落ちビット数を超える桁落ちが発生する場合は，飽和減算結果が 0 になることによって桁落ちを判定することができる．

最後に，飽和減算結果のレジスタに対して，そのレジスタ内の最も小さいワード（16bit）を検索する vphminposuw 命令を適応し，その結果が 0 になるサブレジスタがあれば桁落ちと判定する．更に vphminposuw 命令は，最小の値が格納されていたレジスタ内の位置も検出するので，桁落ちが発生した演算が，後述する Detection Window 内のどの演算かを判定できる．

この手法を用いることで，精度低下検出処理を 7 命令で実現することができる．更に，精度低下検出に伴う指数部の比較回数を最大で 1/8 回に削減することができ，実行時スローダウンを抑えることが可能である．また，コードを挿入する際に，GCC の拡張インラインアセンブラ構文を使用して，他の SIMD レジスタを使用する命令が，指数部パックレジスタを破壊するようなコードを生成しないようにしている．

5. コンパイラ補助による精度低下検出の自動化

本研究は，ソースコードを書き換えずに，精度低下検出を行うための手段として，コンパイラインフラストラクチャである LLVM を使用した，コンパイラ補助を採用した．コンパイラ補助は，コンパイル時に目的の処理を行うコードを組み込む手段である．精度低下検出を自動化するにあたって，関連研究で紹介した，Dyninst API を用いた，プログラム実行時に目的の処理を修正することや，Valgrind や QEMU 等のツールの監視下でプログラムを実行することが，実行時間の大きなスローダウンに繋がると考えた．その点，コンパイル補助であれば，あらかじめ解析対象プログラムに精度低下検出コードを組み込めるので，実行時に処理の置き換えをすることなく精度低下検出が可能であることから，実行時間のスローダウンを減少させることができると考えられる．

コンパイル補助による精度低下検出の実現にあたって，LLVM (Low Level Virtual Machine) を用いた．LLVM は，コンパイラインフラストラクチャの一種であり，仮想機械をターゲットとした独自の中間表現を持っている．LLVM の最適化は中間表現に対して行われるため，言語やアーキテクチャとは独立して行うことができる．

C 言語で書かれたソースコードから，ターゲットアーキテクチャ固有のアセンブリコードが生成するにはまず，LLVM の C，C++言語用のフロントエンドである clang によって，C ソースコードから LLVM 中間表現のバイトコードに変換される．LLVM は C,C++以外にも FORTRAN や Objective-C，Ada 等の様々なプログラミング言語用のフロントエンドを持っている．次に LLVM オプティマイザーで LLVM バイトコードレベルでの様々な最適化を行う．LLVM オプティマイザーは自分で作成したパスを追加して LLVM バイトコードに適応することもできる．最後に llc という LLVM バイトコードをターゲットアーキテクチャ固有のアセンブリコードに変換するツールによって，アセンブリコードが生成される．

本研究では，精度低下検出対象のソースコードを書き換えることなく，精度低下検出を行うために，LLVM オプティマイザーに精度低下検出を解析対象プログラムに組み込むパスを実装した．LLVM オプティマイザーでは LLVM バイトコードに対して，モジュール，関数，基本ブロック

Vol.2013-ARC-205 No.14
Vol.2013-OS-125 No.14
2013/4/26

の 3 つのレベルで最適化を行うことができる．モジュールは LLVM に入力される 1 つのソースコードファイルが 1 モジュールという区分である．

精度低下検出コードを挿入する処理は基本ブロック単位で行う．これは，分岐をまたいで精度低下検出を行うと，どの演算が原因で精度低下が発生したのか特定できなくなるためである．精度低下検出は Detection Window 単位で行う，これは SIMD レジスタにパックできる指数部の個数が決まっているためである．今回の実装では最大 8 個の浮動小数点数加減算を対象とすることができる．Detection Window は浮動小数点加減算カウンターの値が，8 で剰余算をした結果が 1 の場合に形成を開始する．浮動小数点加減算を検出したら，演算引数と演算結果の指数部を指数部パックレジスタにパックするコードを挿入する．また浮動小数点加減算カウンターが 8 で割り切れる値になるか，基本ブロックの末尾に到達すると Detection Window を閉じて，Window 内の最後の演算命令の後に精度低下検出コードを挿入する．これにより，解析対象のプログラムは，実行時に浮動小数点演算の置き換えをすることなく，自動で精度低下を検出することができる．

6. 評価

本節は，提案する SIMD 命令セットを用いた精度低下検出手法が，従来の浮動小数点加減算ごとに指数部の比較を行う検出方法と比較して，どの程度効率よく精度低下検出が行えているかについて評価と考察について述べる．

評価の際に用いた環境を表 1 に示す．提案する SIMD 命令セットを用いた精度低下検出コードを AVX 拡張命令セットを用いて実装したため，CPU は AVX に対応した Ivy Bridge アーキテクチャの Intel Core i7 を使用した．

表 1　評価に用いたマシンのスペック

コンパイラ	Apple clang version4.1 + LLVM3.2
OS	OS X 10.8.2
CPU	Intel Core i7 2.9GHz
メモリ	8GB

ワークロードとして，高エネルギー加速器機構から提供された式（5）で表現される素粒子物理学における理論数値計算で現れる多次元積分を評価対象とした．この式は，ファインマンループ積分を変形したもで，素粒子反応の散乱断面積の計算に用いられる．この式に類似する 2 次元の式の各パラメータを特定の値に固定し，X-Y 平面でこの関数をプロットすると図 2 となる．この図が示すように，積分領域には絶対値のほぼ等しい正値部分と負値部分がある．この積分領域の加減算に当たる箇所で大量の桁落ちが発生する．

$$I = \int_0^1 \int_0^{1-x} \int_0^{1-x-y} \frac{1}{D^2} dz dy dx \qquad (5)$$

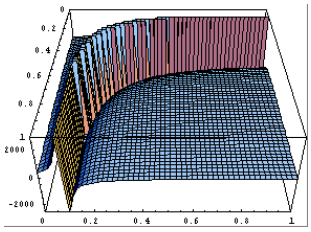

図 2　多次元積分のデータプロット

$$\begin{aligned} D = & -xx * yy * s - tt * zz * (1.0 - xy - zz) \\ & + (xx + yy) * ramda2 \\ & + (1.0 - xy - zz) * (1.0 - xy) * fme2 \\ & + zz * (1.0 - xy) * fmf2 \end{aligned}$$

2 つめのワークロードとして関連研究 [3] でも用いられた，大規模ベンチマークである SPEC CPU2006 に含まれる SPECfp2006 の 470.lbm を使用し，データセットは test を選択した．470.lbm は流体力学分野における 3 次元の非圧縮性流体のシミュレーションを格子ボルツマン法を用いて実装したものである．

精度低下検出は，桁落ちを対象に，ソースコードに SIMD 命令セットを用いた精度低下検出コードを手動で組み込んだ場合と，提案するシステムを用いて，自動で組み込んだ場合の 2 つの方法で行った．

7. 結果と考察

多次元積分の実行時間を表 2 に示す．*original* はワークロードをそのまま実行した実行時間で，*hand* は手動で精度低下検出コードを挿入した実行時間，*proposal* は提案手法を用いた実行時間である．*proposal* スローダウンは 4.7 倍であった．同じワークロードを対象とした QEMU を用いて精度低下検出を行った [7] は実行時間スローダウンが約 7 倍であることから，提案手法が効率よく精度低下検出を行うことを確認した．これは，SIMD 命令を用いて精度低下検出を行ったことにより，指数部の比較回数が削減されたことに加え，提案手法では，QEMU の命令変換のような，精度低下検出と関係のないオーバーヘッドが無かったためだと考えられる．また，470.lbm の実行時間を表 3 に示す．比較の参考として Dyninst API を用いて，プログラム実行時に浮動小数点加減算を，精度低下検出機能付きのヘルパー関数に修正して実行する *related[3]* を加えた，実行時間は *original* を 70 倍したものである．*proposal*

表 2　多次元積分の実行時間

name	*time*	slowdown
original	0.650 sec	1
hand	2.215 sec	3.4 倍
proposal	3.057 sec	4.7 倍

のスローダウンは 2.9 倍であった．今回の評価と同じく，桁落ち検出を対象とした関連研究 [3] では，470.lbm の実行時間スローダウンが 70 倍であることから，提案手法では 470.lbm のスローダウンが 3 倍以内に抑えられており，効率良く精度低下検出が行えているとがわかる．これは，Dyninst API を用いた手法では，浮動小数点加減算ごとに精度低下検出処理を行うのことに比べて，提案手法では複数の精度低下検出処理を 1 度に実行するため，実行スローダウンが削減できたと考える．また，提案手法はコンパイル補助により，コンパイル時に精度低下検出コードを埋め込むので，実行時には命令変換等を行わないため高速に処理できたと考えられる．

表 **3** 470.lbm の実行時間

name	*time*	slowdown
original	5.53 sec	1
hand	15.02 sec	2.7 倍
proposal	16.52 sec	2.9 倍
related[3]	約 387 sec	70 倍

この評価結果から，提案手法の SIMD 命令セットを用いた精度低下検出が，実行時間スローダウンの削減に有効であることが分かった．これは，精度低下検出の際に行う，指数部比較の回数が SIMD 命令セットを用いることにより，削減されたことと，LLVM を使用したコンパイル補助によって精度低下検出コードを，解析対象プログラムに組み込むことによって，プログラム実行時に浮動小数点演算を置き換える必要が無くなったためだと考えられる．

しかし，LLVM によって精度低下検出コードを自動挿入すると，$a = b + (-c)$ のような演算を行う際に，$c = 0.0 - c$ といった精度低下検出の対象としなくても良い減算まで指数部をパックしてしまうため，*hand* と *proposal* の差があると考えられる．また現在の実装では，精度低下が発生した際にユーザーに提供する情報が，プログラム内で発生した全精度低下回数と，精度低下が発生した演算が Detection Window 内の何番目であるか，という 2 つしかない．このため，精度低下が発生した関数名や変数名，アドレス等の他の情報をユーザーに提供できるようにしたいと考えている．さらに，将来的には精度低下検出だけでなく，精度低下が発生した場合に，Detection Window 内の演算を高精度な変数を用いて再計算する等して精度低下の回復が行えるようにしたいとも考えている．

8. まとめ

従来は浮動小数点加減算ごとに行っていた精度低下検出に伴う指数部の比較を，SIMD 命令セットを用いることによって指数部の比較回数を最大 1/8 に削減した．また LLVM を使用したコンパイル補助によって，コンパイル時に解析対象プログラムに精度低下検出コードを組み込むことにより，関連研究 [3][5] とは異なり，実行時に浮動小数点演算の置き換えを行わないことで，実行時間スローダウンを大きく削減することができた．今後は，精度低下検出コードを手動で挿入した場合と自動で挿入した場合の実行時間差の改善が必要である．また，現在の実装では精度低下が発生した場合にユーザーに提供できる情報が，精度低下が発生した回数と，Detection Window 内での位置しか無いため，提供できる情報の種類を増やしていきたいと考えている．

参考文献

[1] IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic IEEE Standard 754 (2008).
[2] The LLVM Project: The LLVM Compiler Infrastructure, , available from ⟨http://llvm.org⟩ (accessed 2013-02-01).
[3] Lam, M. O., Hollingsworth, J. K. and Stewart, G. W.: Dynamic floating-point cancellation detection, *WHIST 11* (2011).
[4] Buck, B. and Hollingsworth, J. K.: An api for runtime code patching, *The International Journal of High Performance Computing Applications*, pp. 317–329 (2000).
[5] Benz, F., Hildebrandt, A. and Hack, S.: A dynamic program analysis to find floating-point accuracy problems, *SIGPLAN Not.*, Vol. 47, No. 6, pp. 453–462 (online), DOI: 10.1145/2345156.2254118 (2012).
[6] Nethercote, N. and Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation, *PLDI'07*, pp. 89–100 (2007).
[7] 松田稔彦，北村俊明：計算精度低下を検出する PC エミュレータの開発，*2011-ARC-197*, Vol. 24, pp. 1–6 (2011).
[8] Bellard, F.: About - QEMU, , available from ⟨http://wiki.qemu.org/Index.html⟩ (accessed 2013-02-01).
[9] 鈴木弘，大岩元：浮動小数点演算における精度見積もりアルゴリズムとその評価，*1994-HPC-53*, Vol. 94, pp. 27–33 (1994).
[10] Intel: Intel Advanced Vector Extensions Programming Reference, , available from ⟨http://software.intel.com/en-us/avx/⟩ (accessed 2013-02-01).