strictfp の実装

首藤 一幸
村岡研究室, 早稲田大学

strictfp とは
言語仕様改訂の経緯
x86 の挙動,問題
strictfp の実現手法
実装法
実験
残されている問題

strictfpとは

- クラス,メソッド,インタフェース修飾子。
  - 例) `strictfp void aMethod(...) { ...`
- **strictfp**の文脈では、ある浮動小数点演算に対して
  どの **Java**仮想マシンもまったく同じ結果を返す。
- **Java 2** から言語仕様に採り入れられた。
  - JDK 1.2 の `javac` は予約語 '`strictfp`' を認識する。

strictfpの実装 – この応募について

- **Golliver**氏提案手法を **JIT**コンパイラ **shuJIT** に実装。
  - 世界初の strictfp 対応...ではなかった。(後述)
  - 手法が有効であることを確認。
- 実装法についての考察、ペナルティを減らす工夫
  - 丸め精度の設定回数を削減
  - メモリ – レジスタ間のトラフィック削減
- いまだ残されている問題を指摘
  - strictfpではない単精度数の演算にも実はケアが要る。
  - JDK 1.1, 1.0 のセマンティクスは？
  - Javaコンパイラの strictfp 対応。
- 対象
  - 数値計算に興味がある方， x86用 Java実行系の開発者， ...

実行例

- 文脈 **(default or** `strictfp`**)** によって結果が異なる演算
  - 特にこの例は store-reload だけでなく 2度丸め対策 (scaling) も必要とする。

```
 % java StrictfpTest
  shuJIT  for Sun JVM/IA-32
1.1128085447148447E-308 (0x0008008000000000)
  * 1.0000000000000002 (0x3ff0000000000001)
default : 1.1128085447148447E-308 (0x8008000000000)
strictfp: 1.1128085447148447E-308 (0x8008000000001)
2.225073858507201E-308 (0x000fffffffffffff)
  / 0.9999999999999999 (0x3feffffffffffff)
default : 2.2250738585072014E-308 (0x10000000000000)
strictfp: 2.225073858507201E-308 (0xfffffffffffff)
```

背景

言語仕様改訂の経緯 (1)

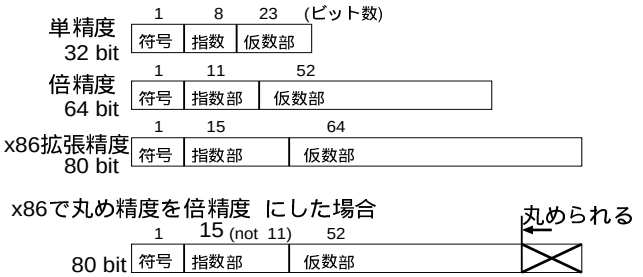
- □**SPARC(Sun)** と **x86(Intel)** の挙動が若干異なる。
  - ○どちらも IEEE 754-1985 (浮動小数点演算規格) 準拠だが...
- □**x86** の仕様が **Java**の(旧)言語仕様と合致しない。
  - ○言語仕様に従おうとすると、数倍の性能低下。
  - ○他のプロセッサより不利。
- □それぞれの思惑　**(想像)**
  - ○Javaを高性能計算に利用したい人々
    - ○x86の性能も活かしたい。
  - ○Intel
    - ○x86 が Java に向かないのでは困る。
  - ○Sun
    - ○言語仕様と JVM の実装の不一致は困る。
    - ○Intel 他につつかれて困る。

言語仕様改訂の経緯 (2)

- □**Intel, SuperCede** が **Sun** に要求
  - ○x86, PowerPC の性能を活かせるように
    - ○x86内部の拡張精度 (80bit 浮動小数点数) の許容
    - ○PowerPC の fmac (fused multiply-accumulate) 命令の許容
- □**Sun** が拡張案を提示**, public review (1998**年 **5**月**)**
  - ○strictfp (1), widefp
- □**Java Grande Forum** の **Numerics Working Group (JGNWG)** による対案 **(9,10**月**)**
  - ○strictfp (2), (default: 指定なし), associativefp
- □**Sun**、**JGNWG** 案を若干修整して採用
  - ○strictfp (2), (default)
  - ○注) strictfp の (1)と(2) は若干異なる。

x86 の挙動

- □浮動小数点数は常に、拡張精度 **(80bit)** で保持される。
- □演算毎に仮数部のみが指定の丸め精度に丸められる。
- □指数部は常に **15 bit (**拡張精度**)**。この設計が元凶。
- □この設計も **IEEE 754** 準拠。
  - ○x86 の設計を許容するために、許容範囲を広げた？



x86 の問題

□指数部が常に **15 bit (not 11,8 bit)** あるため、
起こるべき **overflow, underflow** が起きない。

指数部が表現できる範囲
$(-1)^s \cdot 2^E \cdot 1.b_1b_2b_3\ldots b_{p-1}$ として

	ビット数	E の範囲		
単精度	8 bit	-126	~	+127
倍精度	11 bit	-1022	~	+1023
x86 拡張精度	15 bit	-16382	~	+16383

○例) 倍精度数に対して 次の演算を行った場合

$$(1) register = 2^{+1023}$$
$$(2) register += 2^{+1023}$$
$$(3) register -= 2^{+1023}$$

○SPARC の挙動 (2) の時点で overflow が起きて +∞

○x86 の挙動 overflow は起きずに 2 ** +1023

□**underflow** については **2**度丸めという問題もある。

strictfp の実現手法

□**Roger A. Golliver (Intel)** が提案。

○store-reload
○scaling (down and up)

□完全に言語仕様 **(and SPARC** の挙動**)** に沿える。

○cf. store-reload のみの旧手法。2度丸めを防げない。

□比較的ペナルティが小さい。

○cf. SPARC の挙動をエミュレーション。10倍以上遅い。

□**Java Grande Forum** の **Numerics Working Group (JGNWG)** が推奨。

strictfp の実現手法 (contd.)

□**store-reload**

○演算のたびに結果を倍(単)精度数としてメモリにストアする。

○ストアによって、指数部は 11(8) bit で表現される。

○レジスタ上ではどうしても 15 bit。

○加減乗除算で必要。

store-reload だけでは、演算時とストア時の **2**回丸めが起き、
きちんと**1**度で丸められた場合と結果が異なり得る。そこで**...**

□**scaling**

○演算前にオペランドに定数を乗じ、結果にその逆数を乗じる。

○scale ‘down and up’ または ‘up and down’

○レジスタ上で適切に underflow を起こし、2度丸めを防ぐ。

○2度丸め: 非正規化数がレジスタ上で正規化数として
扱われてしまい、メモリへのストア時に非正規化数に変換される。

○乗除算で必要。

実装

実装

- **JIT** コンパイラ **shuJIT** に **Golliver** の手法を実装。
- **strictfp** の文脈 **(**メソッド**,** クラス**)** に対して、仕様を満たすネイティブコードを生成させる。
- 何通りかの実装を行い、得失、性能を比較。
- 生成されるコードが行うべき処理
  - 丸め精度の設定。
  - 指数部に適切に overflow, underflow を起こさせる。
    - store-reload と scaling

丸め精度の設定

- **[**素直な実装**]** 演算に応じた精度に設定する。
  - × 効率が悪い。
    - 倍精度,単精度の演算が交互に行われるような場合、頻繁に設定し直す必要がある。
    - 丸め精度の設定はメモリ(キャッシュ)アクセスを伴う。
  - cf. SPARC や MIPS は精度ごとの演算命令を持つ。問題なし。
    - 例) fmuls と fmuld (SPARC), mul.s と mul.d (MIPS)

なんとかならないか...

実装 (contd.)

shuJIT への組み込み方

- 内部オペコードを追加。
  - コンパイル過程:
    バイトコード命令 → 内部オペコード → ネイティブコード
  - 丸め精度設定命令
    - fppc_double, fppc_save, fppc_restore
  - scale の pre-load 命令 (メソッドの先頭用)
    - strictenter, strictexit
  - strictfp 対応演算命令
    - fmul_strict, dmul_strict, fdiv_strict, ddiv_strict
- おまけ
  - shuJIT に次のオプションを用意:
    - (バイトコードによる) すべての演算に strictfp を強要する
    - strictfp 指定を無視する

丸め精度の設定 (contd.)

- **[**結論**]** **store-reload** が前提なら、倍精度のままでよい。
  そのまま単精度数の演算を行っても問題ない。
  - [Q] 2度丸めで結果が変わる危険はないのか？
    - 1. 結果がレジスタに格納される時、倍精度 (53 bit 精度) に。
    - 2. 続いてメモリにストアされる時、単精度 (24 bit 精度) に。
  - [A] 問題なし。
    - 1度で単精度まで丸めた値と、2度丸めの値は異なり得ない。
    - 乗算の場合: 24 bit × 24 bit はたかだか 48 bit、1度目の丸めの影響を受けない。
    - 除算の場合: (略: 技術概要参照)
      数学的に厳密な証明はできていない...

指数部の適切な **overflow, underflow**

store-reload ＆ scaling

□**store-reload**

○すでに実装されていた。

○...というか、shuJIT は (残念ながら) 常に行っている。

□**scaling**

○実装する上での種々の選択

◦scaling の方法。

◦scale をいつレジスタにロードするか。

◦どの scale をレジスタに置いておくか。

scaling

適切に指数部の underflow を起こし、2度丸めを防ぐ。

例) 倍精度数の a÷b は次の手順で計算する。

a × 2 ** -16382-(1022) ÷ b × 2** -(-16382-(-1022))

指数部が 15 bit あっても、11 bit しかない場合と同じ境界で underflow が起きる。

□**[選択 1] scaling** の方法

○(1) 2**n を乗じる。(もしくは 2**-n で割る。)

○(2) fscale 命令を用いる。(2の巾によるスケーリング命令)

○[実験] 両方を実装して比較。

scaling (contd.)

□**[選択 2]** いつ **scale** をレジスタにロードするか？

○(1) 乗じる時点。メモリ(キャッシュ)から直接乗じる。

○(2) strictfp なメソッドの先頭。

○(3) JIT コンパイラの初期化時。

○『メモリアクセスの頻度』と『レジスタ消費量 × 時間』のトレードオフ。

○[実験] (1), (2) を実装して比較。

scaling (contd.)

定数をレジスタに置いておく場合 ([選択 2] の (2),(3)) には...

□**[選択 3]** どの **scale** をレジスタに事前ロードするか？

○(1) 4つすべて。

倍精度用: 2 ** -(16383-1023), 2 ** (16383-1023)

単精度用: 2 ** -(16383- 127), 2 ** (16383- 127)

○(2) scale down 用の 2つ。

2 ** -(16383-1023) と 2 ** -(16383-127)

scale up 時にはこの逆数を乗じる必要があるので、

そのつど生成するか、代わりに除算を行うかする。

○(3) 倍精度、単精度、必要な方だけ。

(1), (2) と直行する。

○[実験] fscale 方式では逆数の算出コストが低い (fchs 命令) ため

◦scaling を乗算で行う場合 (1)

◦scaling を fscale 命令で行う場合 (2)

scaling (contd.)

scaling を fscale 命令で行う場合 ([選択 1] の (2)) には...

□**[選択 4]　scale のメモリ上の形式**

○倍精度数 (64bit) or 単精度数 (32bit) or 整数 (32bit)

○『メモリ - レジスタ間のトラフィック』と
『整数 → 浮動小数点数の変換コスト』に影響する。

○[実験]　倍精度

◦整数からの変換はオーバヘッドとなる。
130 ~ 140 msec / 10 ** 7 回　(Mobile Pentium II / 333MHz)

◦倍精度と単精度は性能変わらず。

実験

実験

何通りかの実装を比較。

□実験内容

○10回の乗算(除算) を含んだ式を 10**6 回計算。

○テストコードは技術概要に付属。

□環境

○MMX Pentium / 233 MHz と Mobile Pentium II / 333 MHz

○Linux 2.2.13pre7

○JDK 1.1.7 v3 と JDK 1.2 pre-release 2

□条件

○scaling 方法

◦fscale 命令

◦乗算命令

○scale の事前ロード

◦行わない。scaling の際にメモリ(キャッシュ)からロード。

◦行う。メソッド先頭でレジスタにロード。

実験結果

□**Pentium II, JDK 1.2**

scaling 方法	事前ロード	乗算 (時間, 比)		除算 (時間, 比)	
非 strictfp		3701	1.00	796	1.00
fscale 命令	有	5598	1.51	2590	3.25
fscale 命令	無	5680	1.53	2671	3.36
乗算命令	有	9855	2.66	1097	1.38
乗算命令	無	10674	2.88	1085	1.36

(ミリ秒)

□**Pentium, JDK 1.1.7**

scaling 方法	事前ロード	乗算 (時間, 比)		除算 (時間, 比)	
非 strictfp		4004	1.00	903	1.00
fscale 命令	有	5648	1.41	2749	3.04
fscale 命令	無	5727	1.43	2836	3.14
乗算命令	有	10434	2.61	1117	1.24
乗算命令	無	11712	2.93	1976	2.19

(ミリ秒)

考察

□ペナルティの大きさ
- ○JGNWG の見積もり:「Golliver の手法によるペナルティは 2 ~ 4倍」
- ○今回の実験では 1.43 ~ 3.36 倍 (fscale使用,事前ロード無)。
  - ○ただし、今回観測されたのは scaling のペナルティのみ。
    非strictfpでも store-reload が行われているので。(前述)

□事前ロードの効果 **(10 ** 7 回あたり)**
- ○fscale使用: 約 80 msec
- ○乗算使用: 約 800 ~ 1300 msec

□事前ロードの方法
- ○現実装は、scaleをメソッド先頭でpush、末尾でpopしている。
- ○strictfpメソッドがstrictfpメソッドを呼んでいった場合、
  FPUレジスタスタック (深さ 8) が溢れてしまう。
- ○現実装で妥当なのは『fscale命令でscaling & 事前ロード無し』か？

残されている？問題

□実装がほとんどない！
- ○Sun すら実装していない。
  - ○JDK 1.3beta でも未実装
- ○現在ある実装
  - ○BulletTrain: Ahead-of-Time コンパイラ
  - ○shuJIT: JIT コンパイラ

残されている？問題

残されている？問題 (contd.)

□**strictfp** ではない単精度数の演算
- ○言語仕様の改訂は、x86 用実行系の性能を妨げないことが目的。
- ○しかし... strictfp ではない文脈についても実装には注意を要す。
- ○また、ある程度のオーバヘッドが避けられない。
- ○次のどちらかを行う必要がある。
  (注: 場合に応じて使い分けてもよい)
  - ○演算に応じて丸め精度を設定し直す。
  - ○丸め精度は倍精度に固定、単精度数の演算では store-reload。
- ○理由
  - ○1. 新言語仕様では、15 bit (拡張精度) 指数部が許されたが、
    仮数部はきちんと倍精度: 53 bit、単精度: 24 bit が要求される。
  - ○2. x86 は精度に応じた演算命令を持たない。(cf. SPARC, MIPS)

改訂は目的を達したか？

残されている？問題 (contd.)

□言語仕様の改訂は目的を達したか？ **(contd.)**

- ○『仮数部の精度を strict に...』は JGNWG の主張？
  - ○最初のSun案は、x86 拡張精度 (仮数部 64 bit) を許していた？
  - ○JGNWG が対案を出した理由
    『Sun案は Java の predictability を下げてしまう』
- ○現仕様が妥当だとしたら、理由は...
  - ○単精度数の演算はまれなので、これで充分？
  - ○仮数部の strict な精度は predictability に大きく貢献する？？？

残されている？問題 (contd.)

□**JDK 1.1, 1.0 (**旧言語仕様**)** はどうあるべきだったか？

- ○新仕様で言うところの default か strictfp か？
- ○『常に **strictfp** の文脈』が妥当な解釈のようだ。　根拠:
  - ○JGNWG の文書はこれを前提として書かれている。
  - ○Sun は ‘exact reproducibility’ をうたっていた。
  - ○旧仕様いわく『The floating-point types are float, whose values are 32-bit IEEE 754 floating-point numbers, and double, whose values are 64-bit IEEE 754 floating-point numbers.』
- ○...しかし、Sun の意図はともかく、
  『**x86** の挙動そのまま **(15 bit** 指数部**)** が許されていた』
  という解釈はできないだろうか？
  - ○曖昧だった点 (15 bit 指数部) が改訂で明確になった。

残されている？問題 (contd.)

□**Java** コンパイラの **strictfp** 対応

- ○実行系だけでなく、Javaコンパイラも対応する必要がある。
  - ○Javaコンパイラ: `javac`,`Jikes`,`KJC(Kopi Java Compiler)`,`GCJ`,`JUMP`,`Guavac`, ...
- ○コンパイル時に式を計算する場合、
  その式の文脈を考慮せねばならない。
  - ○単純には、常に strictfp の文脈で式を評価すればよい。
- ○少なくとも JDK 1.3beta の `javac` は strictfp の文脈を考慮せず。
  - ○JDK 1.2 pre-relase2, Linux で、JDK 1.3beta の `javac(in tools.jar)` を用いて確認。

strictfp に対応した **Java**コンパイラはまだない？

まとめ

技術概要 提出後...

- □世界初の実装ではなかった...残念。
  - ○BulletTrain (Ahead-of-Time コンパイラ) が先。
    - ○BulletTrain: 99/4/27 ~ 99/8/13の間のいつか。shuJIT: 99/8/18。
    - ○常に strictfp の文脈？ (ver.1.1.2 の試用版)
    - ○NaturalBridge LLC 社の製品。$999。
- □**Java for computational science and engineering (java-for-cse) ML**でアナウンス、コメントをもらった。
  - ○BulletTrain について。 by Joseph D. Darcy (UCB)
  - ○非 strictfp 文脈での単精度数の演算法。 by Roger A. Golliver (Intel)
    - ○Pentium III なら SSE 命令を使うことで、store-reload も丸め精度の頻繁な設定も避けられる。
- □技術概要を **web**に掲載、**NetNews**でアナウンス。
  - ○反応なし...

まとめ

- □**strictfp** とは
- □背景
  - ○言語仕様改訂 (strictfp 導入) の経緯
  - ○x86 の挙動, 問題
  - ○strictfp の実現手法
- □実装
  - ○丸め精度の設定回数を減らす工夫
  - ○scaling の各種実装法
- □実験
  - ○各種実装法を比較
- □残されている？問題
  - ○実装の少なさ
  - ○strictfp でない単精度数の演算について
  - ○Java コンパイラの strictfp 対応
  - ○JDK 1.1, 1.0 (旧言語仕様) のセマンティクス